OPENCL ON INTEL FPGA

Thinks Efficienty !!!

Marc Gaucheron, Intel PSG

JDEV2017

Agenda

What Is FPGA ?

- Anatomy of FPGA – 15mn
- FPGA development flow – 15mn
- High Level Synthesis Tools – 15mn
- OpenCL for FPGA – 2h00
  - OpenCL SDK & Execution Model
  - Optimizing Kernel For FPGA
- Conclusion – 15mn

Acronyms

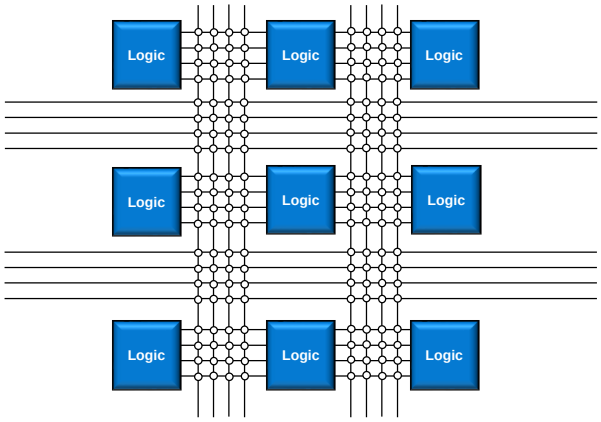
LUT- LookUp Table

LE – Logic Element

ALM – Adaptive Logic Module

LAB – Logic Array Block

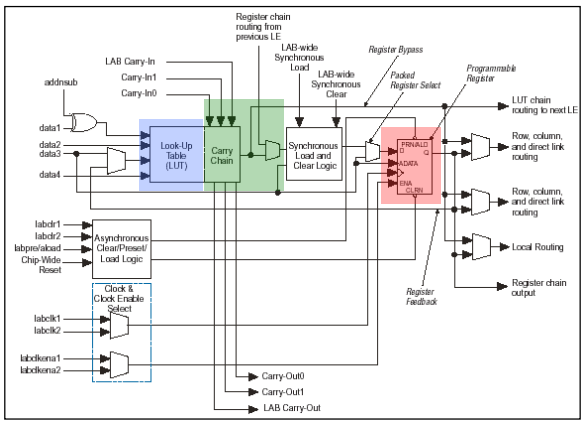
Field Programmable Gate Array (FPGA)



Programmable
Switch Fabric

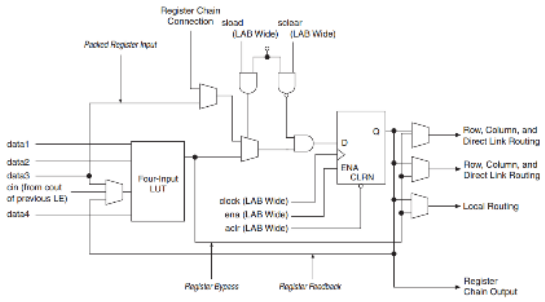
FPGA Logic blocks

FPGA logic is made up of Logic Elements (LEs) or Adaptive Logic Modules (ALMs)

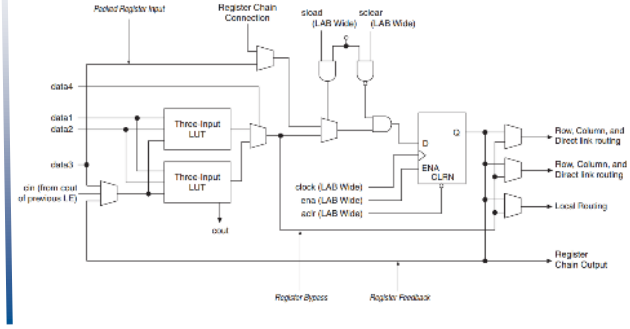


Logic Element (LE)

Normal Mode – Suited for general logic applications and combinational functions



Arithmetic Mode – Implementing adders, counters, accumulators, & comparators



The Quartus Prime software automatically chooses the appropriate operation mode for individual functions for optimal performance

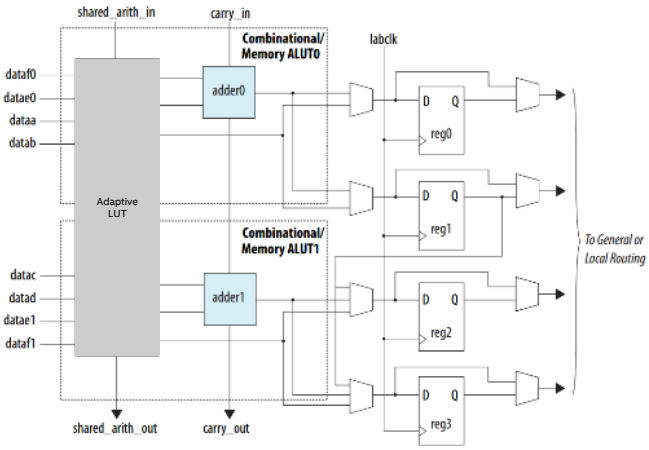
Adaptive Logic Module (ALM)

One ALM contains 4 programmable registers, each with the following ports:

- Data
- Clock
- Synchronous & asynchronous clear
- Synchronous load

Backward compatible with 4-input LUT architectures

Quartus Prime automatically configures ALMs for optimized performance

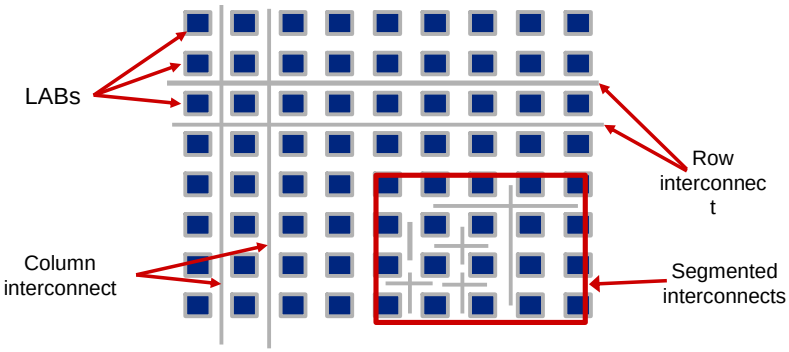


Logic Array Block (LAB)

LABs are group of LEs or ALMs

Row and column programmable interconnect

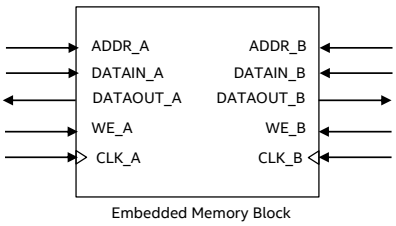
Interconnect may span all or part of the array



FPGA Embedded Memory

Memory blocks

- Create on-board memory structures to support design
  - Single/dual-port RAM
  - ROM
  - Shift registers or FIFO buffers
- Initialize RAM or ROM contents on power-on
  - M9K, M10K, M20K
- Memory LABs (MLABs)
- eSRAM



Embedded Memory Block

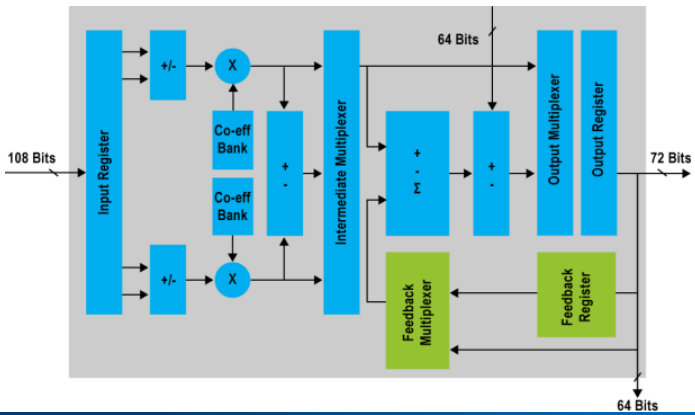
Arria 10 Embedded Memories

Feature		MLABs	M20K
Maximum Performance		700 MHz	730 MHz
Total RAM Bits per Block		640	20,480
Total M20K Memory Bits (Mb) per Device		2-13	13-54
Port Width Configurations		32 x 16 32 x 18 32 x 20	16K x 1, 8K x 2 4K x 4, 5 2K x 8, 10 1K x 16, 20 512 x 32, 40
Parity		✓	✓
Byte Enable		✓	✓
Packed Mode		✓	✓
Address Clock Enable			✓
Mixed Clock		✓	✓
Mixed Width (for Dual Port modes)		✓	✓
ECC Support		✓	Hard
Memory Modes	Single Port	✓	✓
	Simple & True Dual-Port	✓	✓
	Shift Register, ROM, FIFO	✓	✓

DSP Block

Useful for DSP functions

High-performance multiply/add/accumulate operations

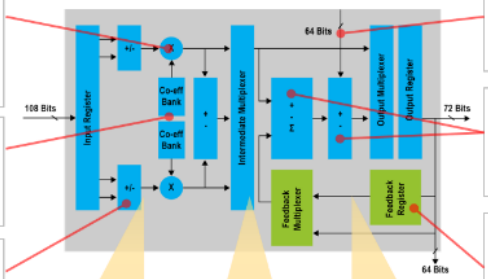


Arria 10 DSP block

Multiplier Modes for Flexibility
- Two 18x19 multipliers, or
- One 27x27 multiplier per block
- 36x36, 54x54 modes using multiple DSP blocks

Integrated Coefficient Registers
- Save memory and routing resources
- Built-in timing closure

Hard Pre-Adders
- Reduce multiplier usage
- Save routing resources

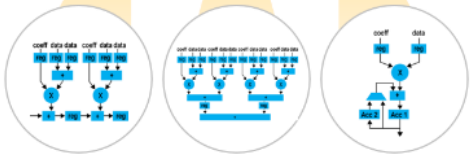


64-bit Cascade Path
- Supports systolic FIR
- Sum of products

Up to 64-bit Adder/ Subtractor/Accumulator
- 1,024-tap filters
- 2,048-tap symmetric filters

Feedback Register and Multiplexer
- Implement two independent filter channels per DSP block

Introduced for Arria V FPGAs, standard for all 20 nm FPGAs



Systolic FIR **Direct FIR** **Serial FIR**

Up to 1.5 TFlops

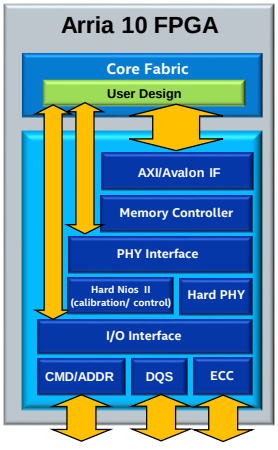
Arria 10 Hard Memory Controller & PHY

Hard memory controller

- Saves logic and memory resources
  - 5K LEs and 29 M20K blocks per x72 DDR3 IF
- Up to x144 support
- Up to 4x72 DDR3 interfaces in a single device

Hard memory controller supports

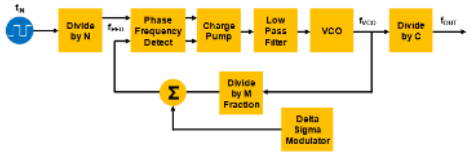
- DDR4, DDR3, LPDDR3
- DDR4 up to 2400 Mbps



Hard Memory Controller & PHY

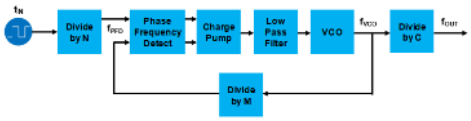
Core PLLs

Fractional PLLs	
Feature	**Description**
# Available	1 for every 3 transceivers on device
Location on die	In the core, adjacent to transceivers
Operating modes	Fractional-synthesis Integer (M/N where M,N=Integer)
FPGA clock network access	Transceiver reference clock GCLK (global clock) PCLK (periphery clock) RCLK (regional clock)



fPLLs Reduce Cost, Power and Circuit Board Space

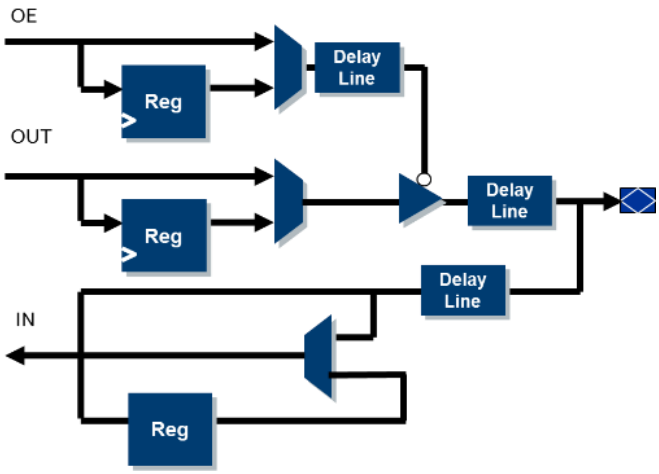
IO PLLs	
Feature	**Description**
# Available	1 for every IO bank (48 GPIOs)
Location on die	In the core, adjacent to IO banks
Operating modes	Integer (M/N where M,N=Integer)
FPGA clock network access	External Memory Interface LVDS SerDes Interface GCLK (global clock) PCLK (periphery clock) RCLK (regional clock)



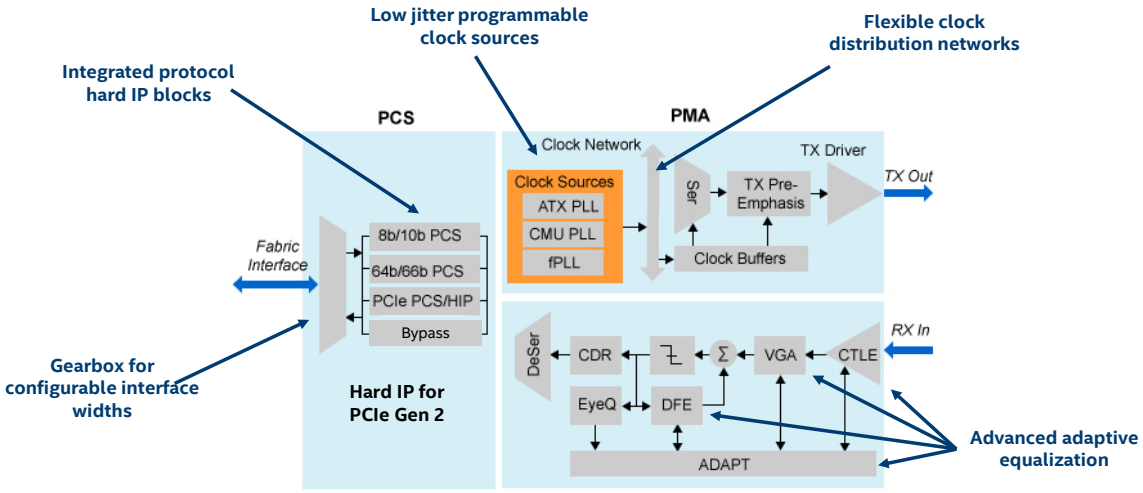
IO PLLs Enable High-Bandwidth IO Interfaces

FPGA IO Elements

- Input/output/bidirectional
- Multiple I/O standards
- Differential signaling
- Current drive strength
- Slew rate
- On-chip termination/pull-ups
- Open drain/tri-state

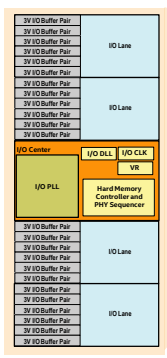


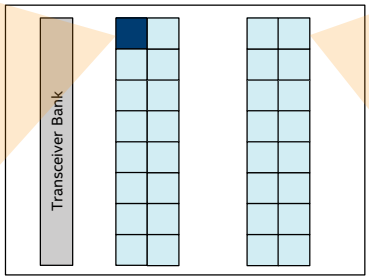
High Speed IO Transceivers



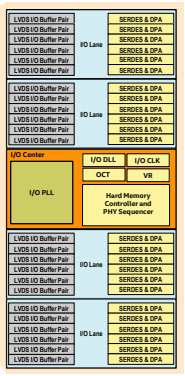
Cyclone® 10 GX: GPIO Blocks

3V I/O Block Structure*





LVDS I/O Block Structure

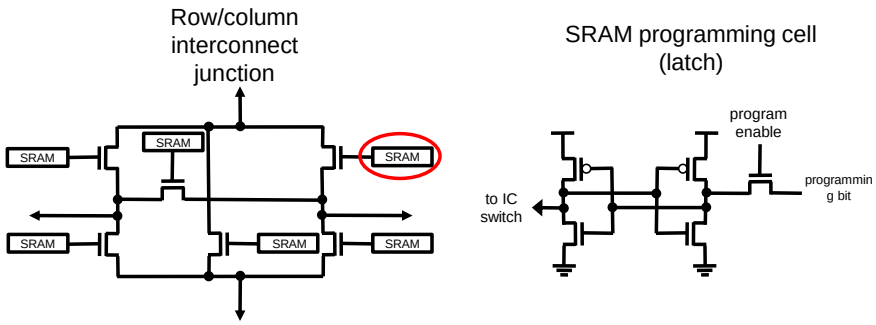


* Only a single 3V I/O Block in Cyclone 10 GX

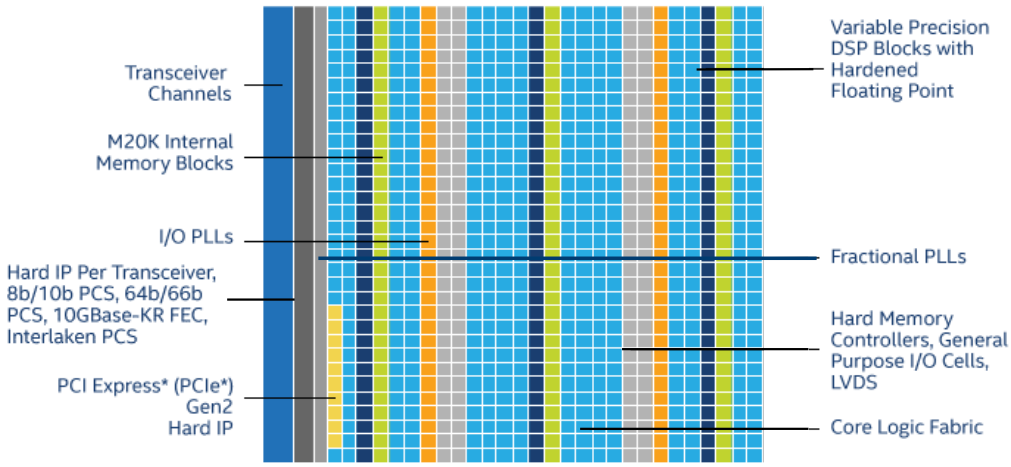
FPGA Programming

FPGAs use SRAM cell technology to program interconnect and LUT function levels

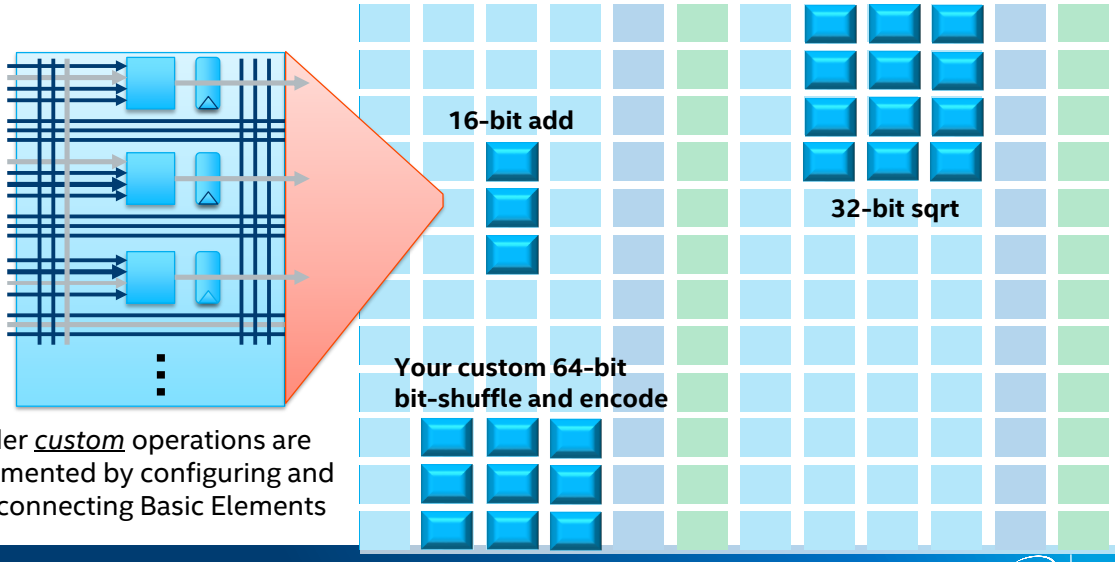
Volatile! Must be programmed at power-on!



FPGA Floorplan



FPGA Architecture: Custom Operations Using Basic Elements

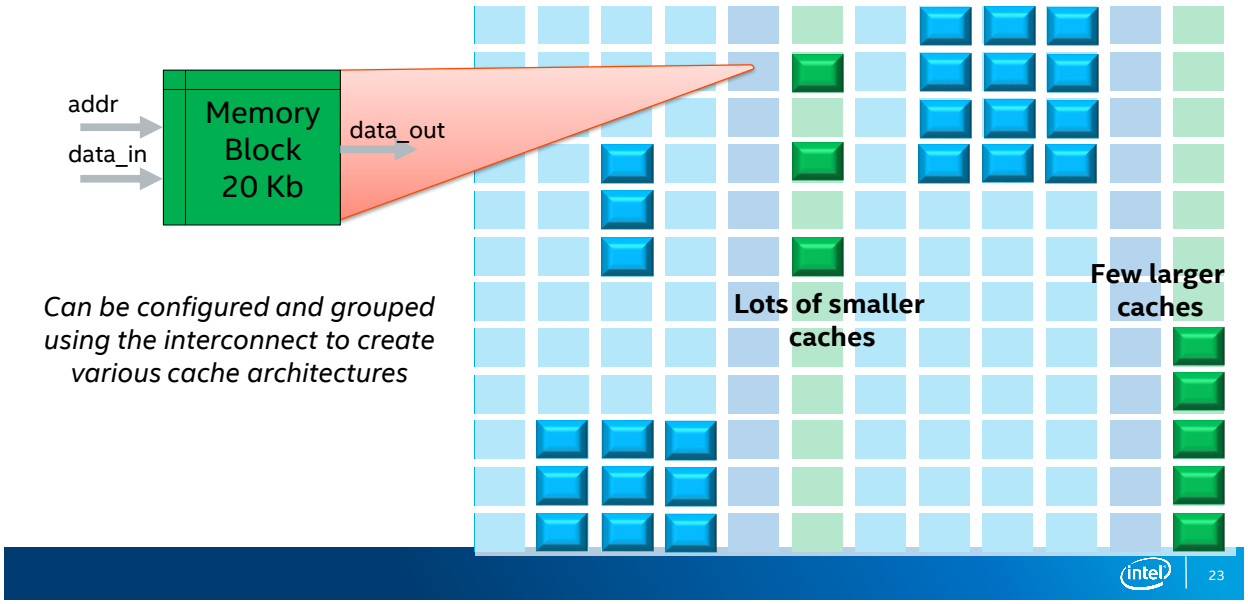


Wider *custom* operations are implemented by configuring and interconnecting Basic Elements

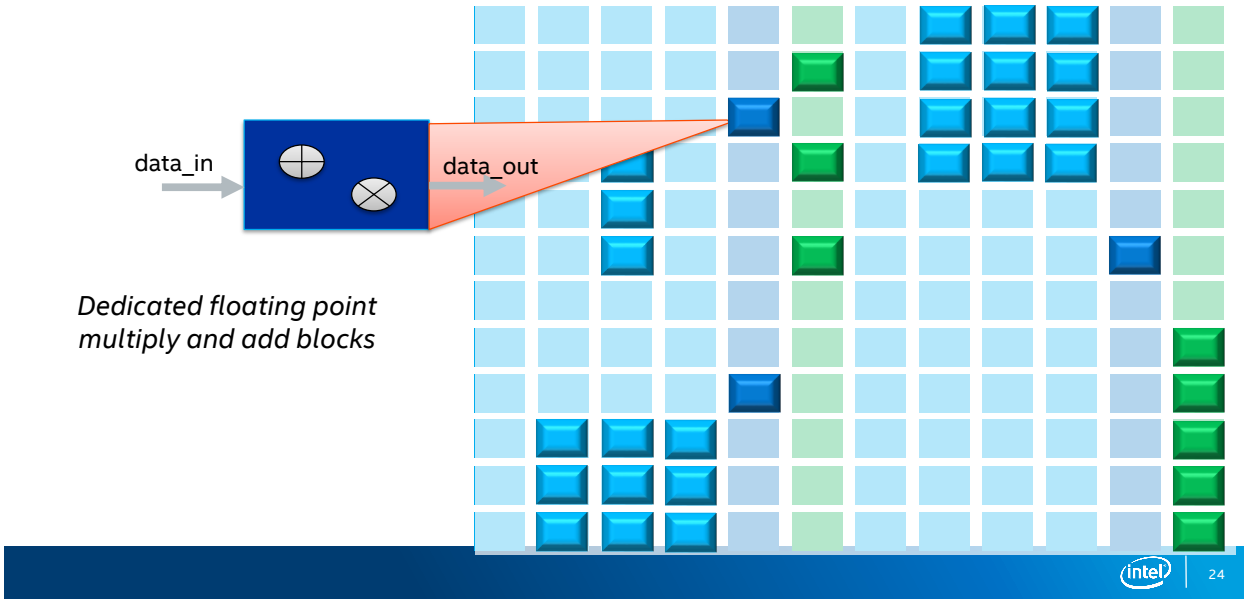
FPGA Architecture: Memory Blocks

Can be configured and grouped using the interconnect to create various cache architectures

FPGA Architecture: Memory Blocks

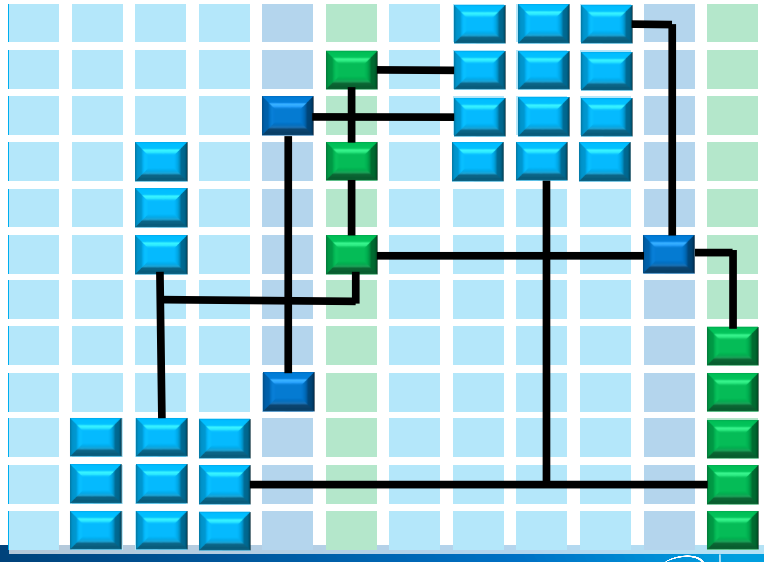


FPGA Architecture: Floating Point Multiplier/Adder Blocks



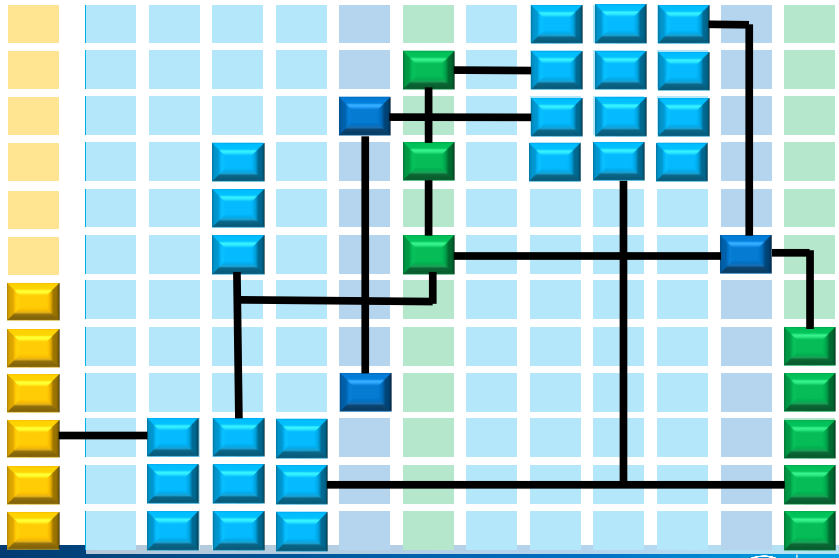
FPGA Architecture: Configurable Routing

*Blocks are connected into a **custom data-path** that matches your application.*



FPGA Architecture: Configurable IO

*The **Custom data-path** can be connected directly to **custom or standard IO interfaces** for inline data processing*



VHDL, VERILOG vs C

VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity parity_generator is
  generic( m          : integer);
  port( input_stream : in std_logic_vector
                       (m-1 downto 0);
        clk          : in std_logic;
        parity       : out bit);
end parity_generator1;

architecture odd of parity_generator is
begin
  P1: process
  variable odd : bit;
  begin
    wait until clk'event and clk = '1';
      odd := '0';
      for i in 0 to m-1 loop
        odd := odd xor input_stream(i);
      end loop;
      parity <= odd;
    end process;
end odd;
```

Verilog

```verilog
module parity_generator (
  input_stream,
  parity,
  clk
);

parameter M = 8;
input  logic [M-1:0] input_stream;
input  logic  clk
output logic  parity

always@(posedge clk) begin
  parity <= ^input_stream;
end

endmodule;
```

C++

```cpp
#include "HLS/ac_int.h"
#define M=8

bool parity(
  ac_int<M> input_stream
) {
  bool parity = true;
  for(int i=0; i<M; ++i)
    parity ^= ap_int[i];
  return parity
}
```

VHDL, VERILOG vs C

VHDL

```
entity ODD_PARITY_TB is
end;

library ieee;
use ieee.std_logic_1164.all;
use WORK.anu.all;
use textio.all

architecture OP_TB_ARCH of ODD_PARITY_TB is
  component Parity_Generator1
    port (
      input_stream : in input;
      clk          : in std_logic;
      parity       : out bit );
  end component;

  signal input_stream : input;
  signal clk    : std_logic;
  signal parity : bit;
  variable dbg  : line;
begin
    U1: Parity_Generator1
    port map(
      input_stream => input_stream,
      clk          => clk,
      parity       => parity
    );
    clk_process : process (clk)
    begin
      if clk <= 'U' then
        clk <= '0' after 1 ns;
      else
        clk <= not clk after 1 ns;
```

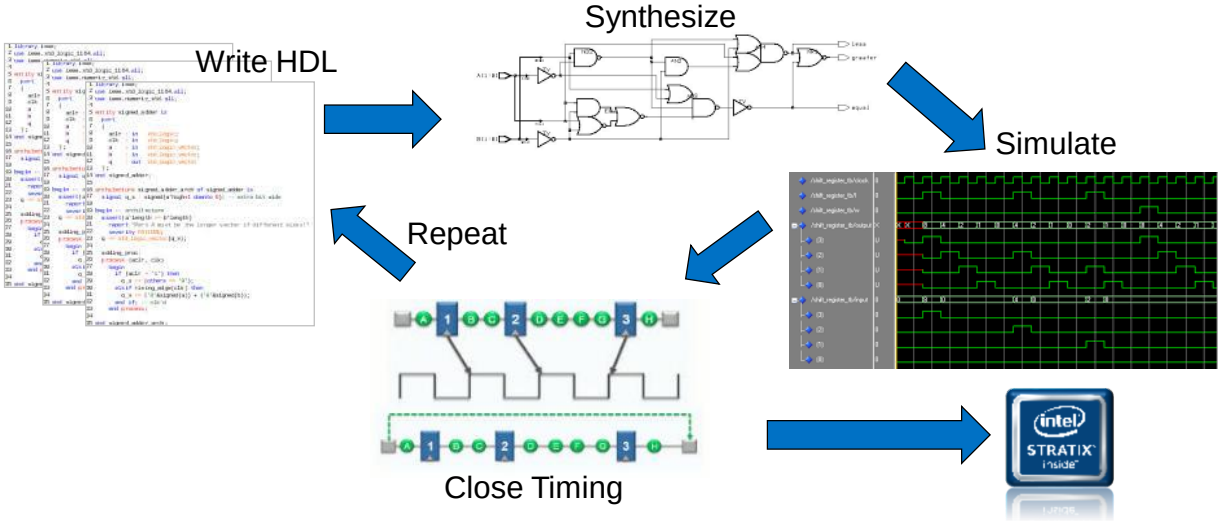
Verilog

```
module parity_tb;
  reg clk, input_stream, parity;

  parity_generator dut (
    .clk(clk),
    .input_stream(input_stream),
    .parity(parity)
  );

  initial
    clk = 0;
  always
    #5 clk = !clk;

  initial begin
    wait @(posedge clk);
    input_stream = 8'b10100110;
    wait @(posedge clk);
    $display("%b => %b",
      input_stream, parity);
    input_stream = 8'b01111100;
    wait @(posedge clk);
    $display("%b => %b",
      input_stream, parity);
  end
endmodule
```

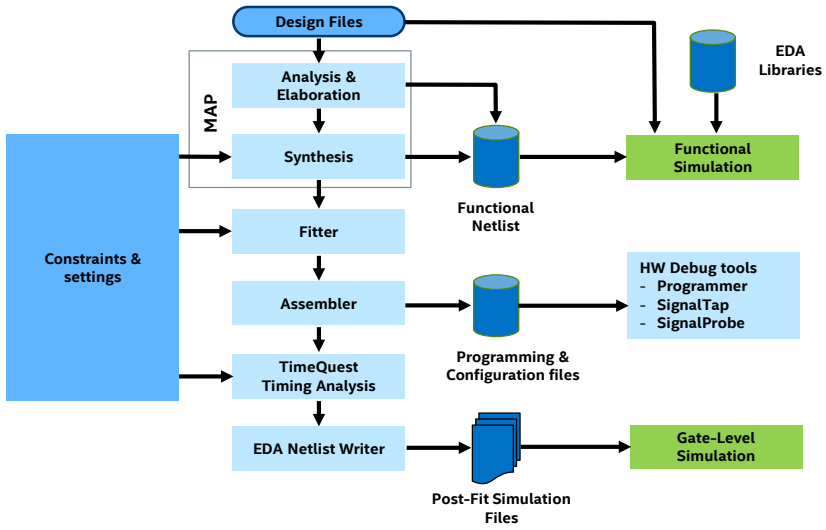
C++

```
#include "stdio.h"

int main() {
  int input;
  input = 0b10100110;
  printf("%h => %d\n", input,
    parity_generator(input));
  input = 0b10100110;
  printf("%h => %d\n", input,
    parity_generator(input));
  return 0;
}
```

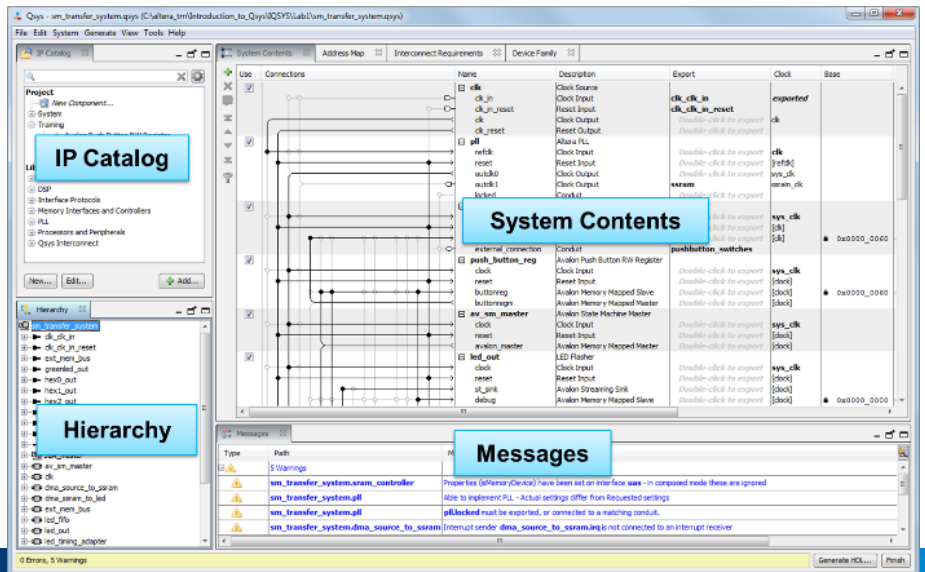

Traditional FPGA Design Process



Tools Flow Overview

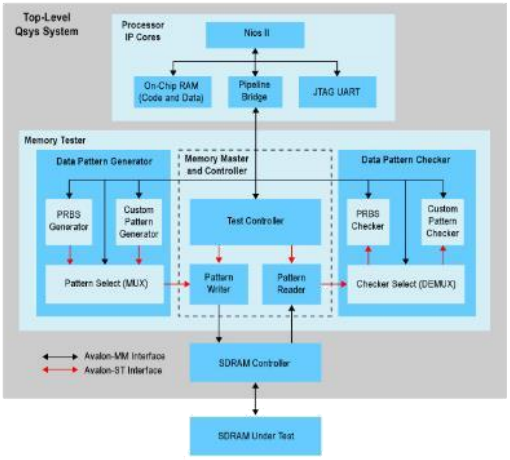


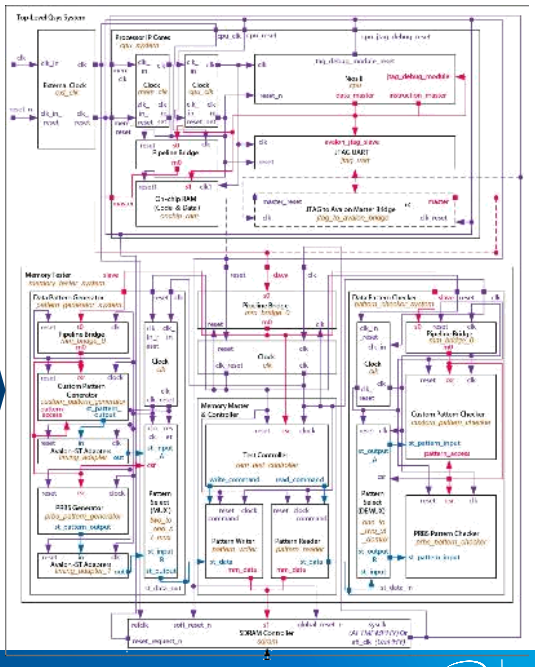
QSYS Prime



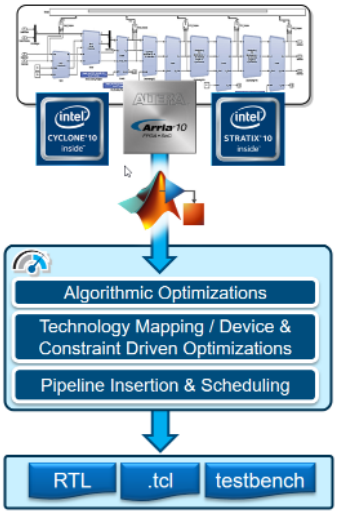
QSYS Example Design

Memory Tester





DSP Builder Flow

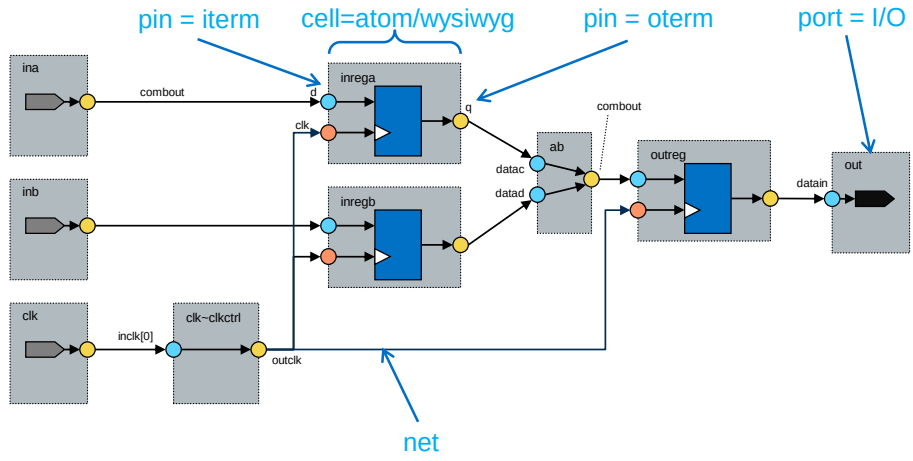


- **Simulink test-bench**
- **Device-independent, unpipelined design**
- **Configuration parameters**
  - Clock speed, data rate, input data width, channel count, Avalon-MM interface definition, …
- **Target device**
- **Cycle- and bit-accurate simulation**

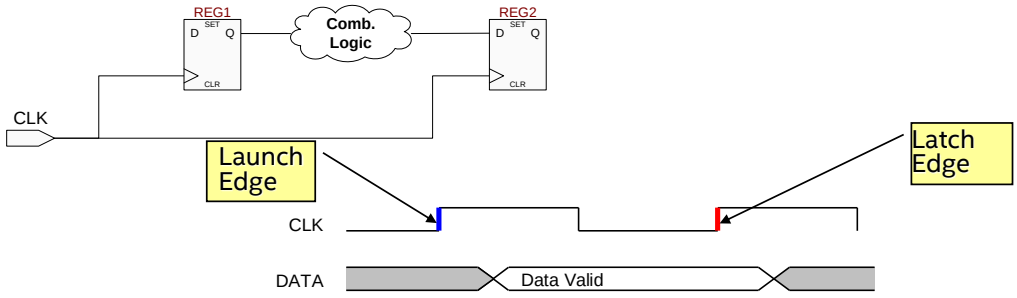
- **Optimize design**
- **Fixed and Floating Point**
- **Optimal mapping to device features**
- **Timing-aware pipeline insertion, balancing and scheduling**
- **Automated resource sharing in IP and folded subsystems**

- **Component ready for integration**
- **ModelSim testbenches for verification**

Logic Netlist Example



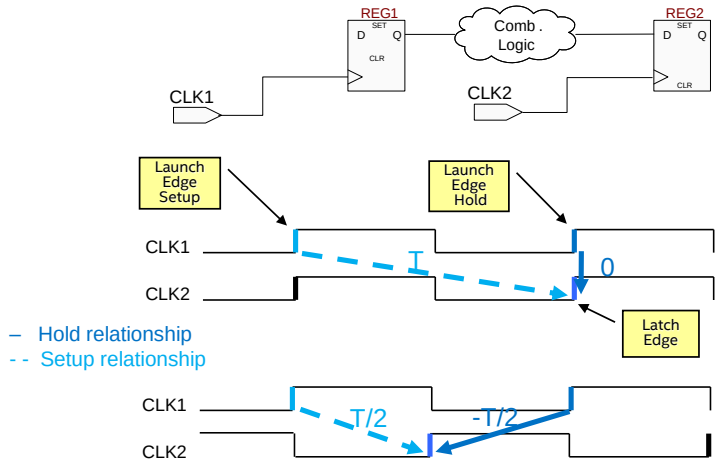
Launch & Latch Edges



Launch Edge: the edge which "launches" the data from source register

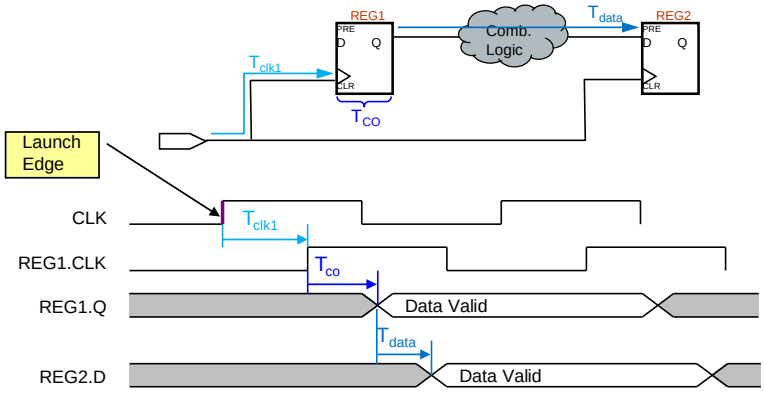
Latch Edge: the edge which "latches" the data at destination register (with respect to the launch edge)

Setup & Hold relationships



Data Arrival Time

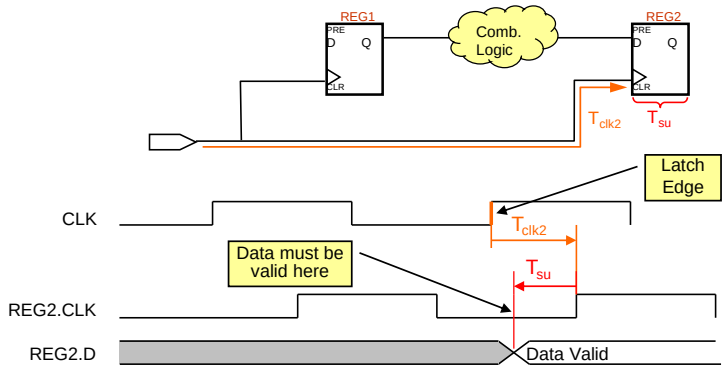
The time for data to arrive at destination register's D input



Data Arrival Time = launch edge + T_{clk1} + T_{co} +T_{data}

Data Required Time - Setup

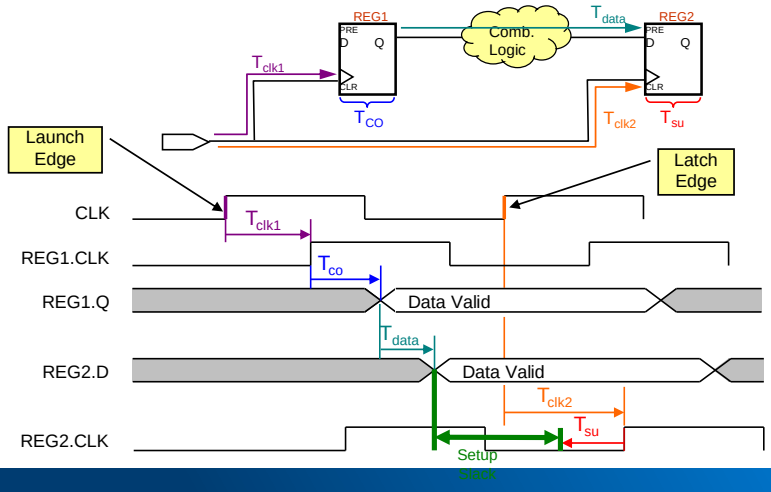
The minimum time required for the data to get latched into the destination register



Data Required Time = Clock Arrival Time - T_{su} - Setup Uncertainty

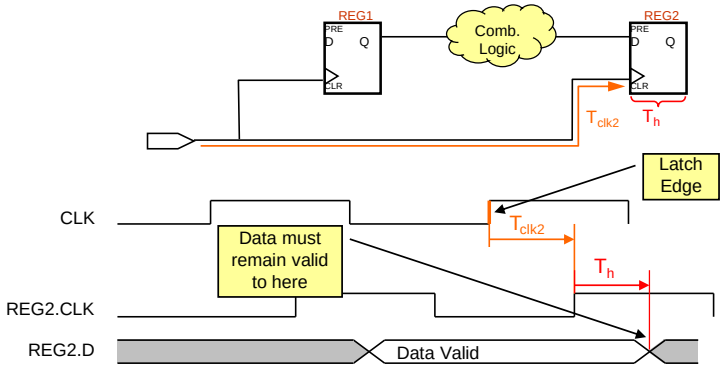
Setup Slack

The margin by which the setup timing requirement is met. It ensures launched data arrives in time to meet the latching requirement.



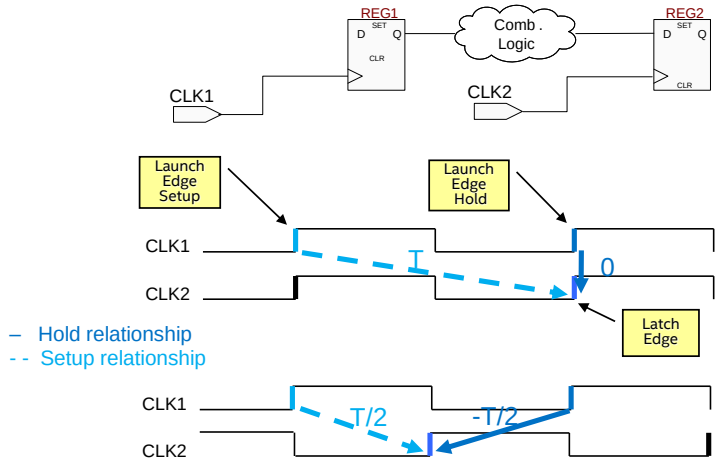
Data Required Time - Hold

The minimum time required for the data to get latched into the destination register



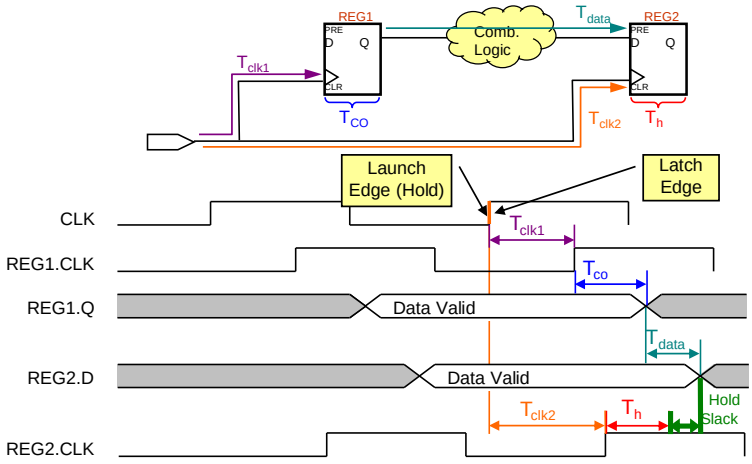
Data Required Time = Clock Arrival Time + T_h + Hold Uncertainty

Setup & Hold relationships



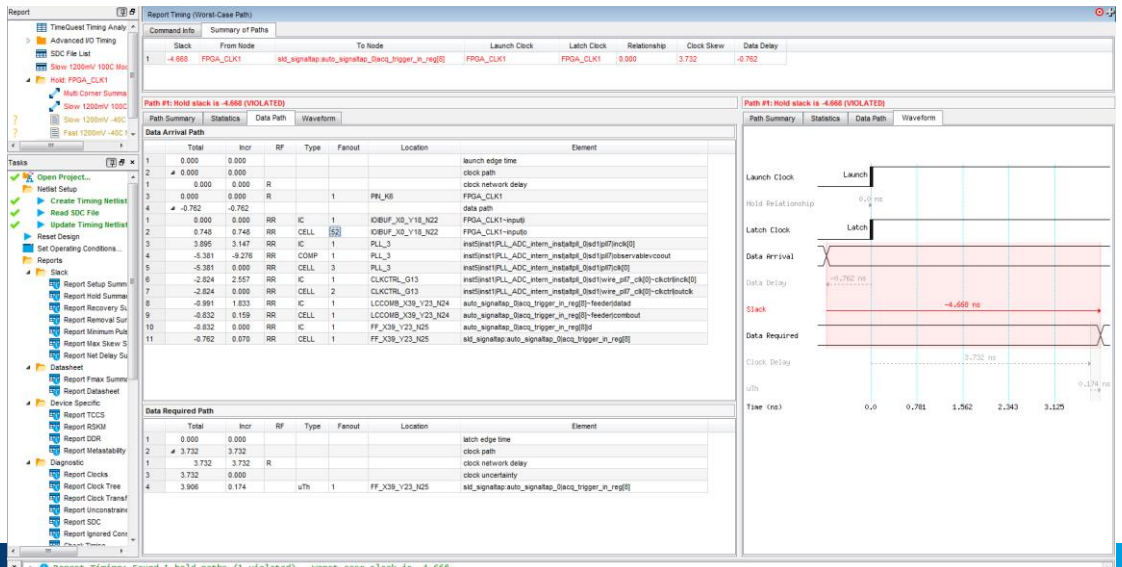
Hold Slack

The margin by which the hold timing requirement is met. It ensures latch data is not corrupted by data from another launch edge.



43

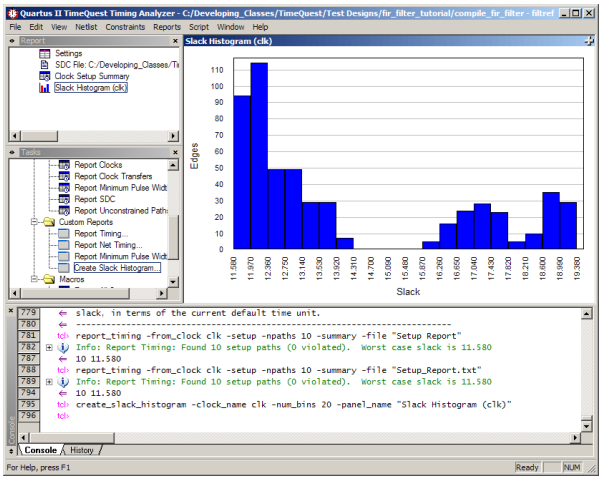
TimeQuest Prime



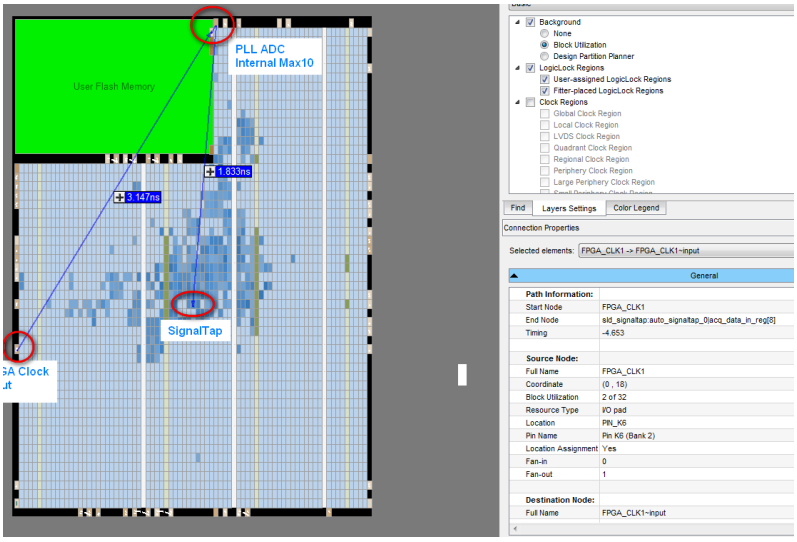
TimeQuest Timing Analyzer

Features

- Synopsys Design Constraints (SDC) support
  - Standardized constraint methodology
- Easy-to-use interface
  - Constraint entry
  - Standard reporting
- Scripting emphasis
  - Presentation focuses on using GUI



Chip Planner

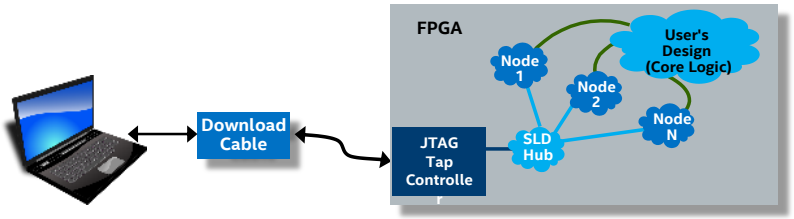


On-chip Debug Technology

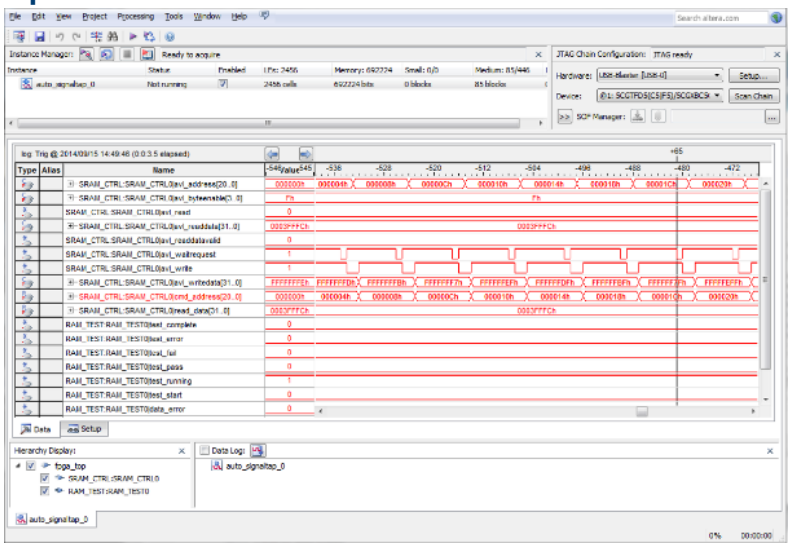
Debug tools communicate with the FPGA via standard JTAG interface

Multiple debug functions can share the JTAG interface simultaneously

- Altera's system-level debugging (SLD) hub technology makes this possible



SignalTap Window - Data



Elementary math functions supporting floating point

Coverage of ~70 elementary math functions

Patented & published efficient mapping to FPGA hardware

- Polynomial approximation, Horner's method, truncated multipliers, …

Compliant to OpenCL & IEEE754 accuracy standards

Rounding mode options for fundamental operators

Half- to Double-precision

Exp, Log and Power
- FPLn
- FPLn1px
- FPLog10
- FPLog2
- FPExp
- FPExpFPC
- FPExpM1
- FPExp2
- FPExp10
- FPPowr

Trig with argument reduction
- FPSinX
- FPCosX
- FPSinCosX
- FPTanX
- FPCotX

Trigonometrics of pi*x
- FPSinPiX
- FPCosPiX
- FPTanPiX
- FPCotPiX

Inverse trigonometric functions
- FPArcsinX\
- FPArcsinPi
- FPArccosX
- FPArccosPi
- FPArctanX
- FPArctanPi
- FPArctan2

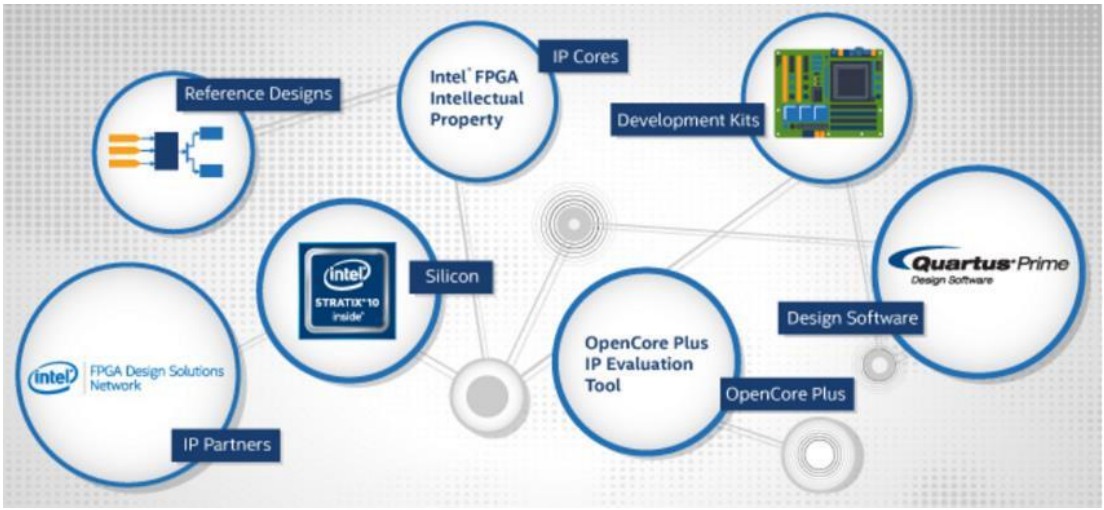
Conversion
- FXPToFP
- FPToFXP
- FPToFXPExpert
- FPToFXPFused
- FPToFP

Macro Operators
- FPFusedHorner
- FPFusedHornerExpert
- FPFusedHornerMulti
- FPFusedMultiFunction

Trigonometrics misc
- FPHypot
- FPRangeReduction

Basic Floating Point
- FPAdd
- FPAddExpert
- FPAddN
- FPSubExpert
- FPAddSub \
- FPAddSubExpert
- FPFusedAddSub
- FPMul
- FPMulExpert
- FPConstMul
- FPAcc
- FPSqrt
- FPDivSqrt
- FPRecipSqrt
- FPCbrt
- FPDiv
- FPInverse
- FPFloor
- FPCeil
- FPRound
- FPRint
- FPFrac
- FPMod
- FPDim
- FPAbs
- FPMin
- FPMax
- FPMinAbs
- FPMaxAbs
- FPMinMaxFused
- FPMinMaxAbsFused
- FPCompare
- FPCompareFused

Intel FPGA IP ecosystem



Intel FPGA IP Catalogue

https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/sg/product-catalog.pdf

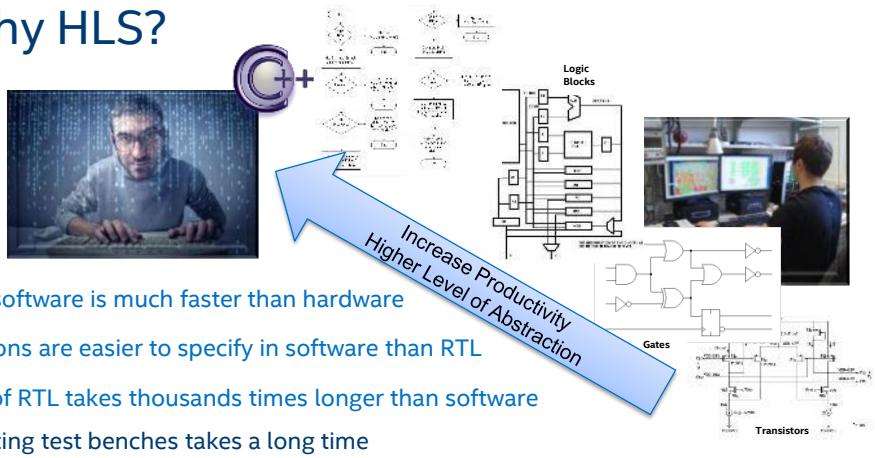
	PRODUCT NAME	VENDOR NAME
DSP	**ARITHMETIC**	
	Floating Point Megafunctions	Intel
	Floating Point Arithmetic Co-Processor	Digital Core Design
	Floating Point Arithmetic Unit	Digital Core Design
	ERROR DETECTION/CORRECTION	
	Reed-Solomon Encoder/Decoder II[1]	Intel
	Viterbi Compiler, High-Speed Parallel Decoder	Intel
	Viterbi Compiler, Low-Speed/ Hybrid Serial Decoder	Intel
	Turbo Encoder/Decoder	Intel
	High-Speed Reed Solomon Encoder/ Decoder	Intel
	BCH Encoder/Decoder	Intel
	Low-Density Parity Check Encoder/ Decoder	Intel
	Zip-Accel-C: GZIP/ZLIB/Deflate Data Compression Core	CAST, Inc.
	Zip-Accel-D: GUNZIP/ZLIP/Inflate Data Decompression Core	CAST, Inc.
	FILTERS AND TRANSFORMS	
	...Transform (FFT)/...	

	PRODUCT NAME	VENDOR NAME
DSP (CONTINUED)	**VIDEO AND IMAGE PROCESSING**	
	Video and Image Processing Suite[1]	Intel
	HD JPEG 2000 Encoders/ Decoders	IntoPIX
	TICO Lightweight Video Compression	IntoPIX
	Multi-Channel JPEG 2000 Encoder and Decoder Cores	Barco Silex
	VC-2 High Quality Video Decoder	Barco Silex
	VC-2 High Quality Video Encoder	Barco Silex
	MPEG-2 TS Encapsulator/ Decapsulator for SMPTE2022 1/2	IntoPIX
	JPEG Encoders	CAST, Inc.
	Ultra-fast, 4K-compatible, AVC/ H.264 Baseline Profile Encoder	CAST, Inc.
	Low-Power AVC / H.264 Baseline Profile Encoder	CAST, Inc.
	H.265 Main Profile Video Decoder	CAST, Inc.
	H.265 Encoders	Jointwave Group LLC
	H.264 Encoders	Jointwave Group LLC
	Video Processor and Deinterlacer with ...	... Inc.

Board Partners

ABSTRACTING THE FPGA DEVELOPMENT FLOW

So, Why HLS?



Debugging software is much faster than hardware

Many functions are easier to specify in software than RTL

Simulation of RTL takes thousands times longer than software

- Creating test benches takes a long time
- Updates to code requires changes to testbench to verify
- Simulation of PCIe at 1fs for 500ms = Enough said!

Design Exploration is much easier and faster in software

Mapping a Simple Program to an FPGA

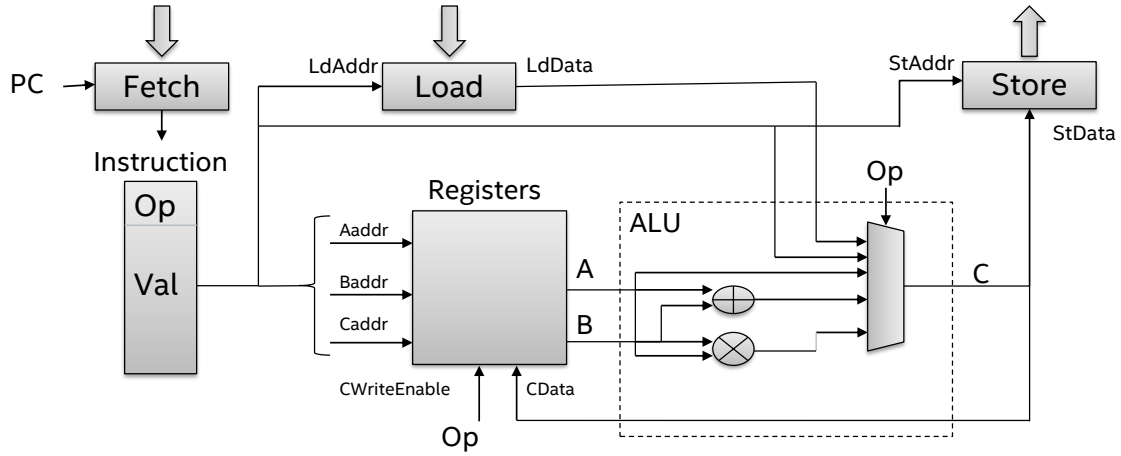
C/C++ instruction

```
Mem[100] += 42 * Mem[101]
```

CPU instructions

```
R0 ← Load Mem[100]
R1 ← Load Mem[101]
R2 ← Load #42
R2 ← Mul R1, R2
R0 ← Add R2, R0
Store R0 → Mem[100]
```

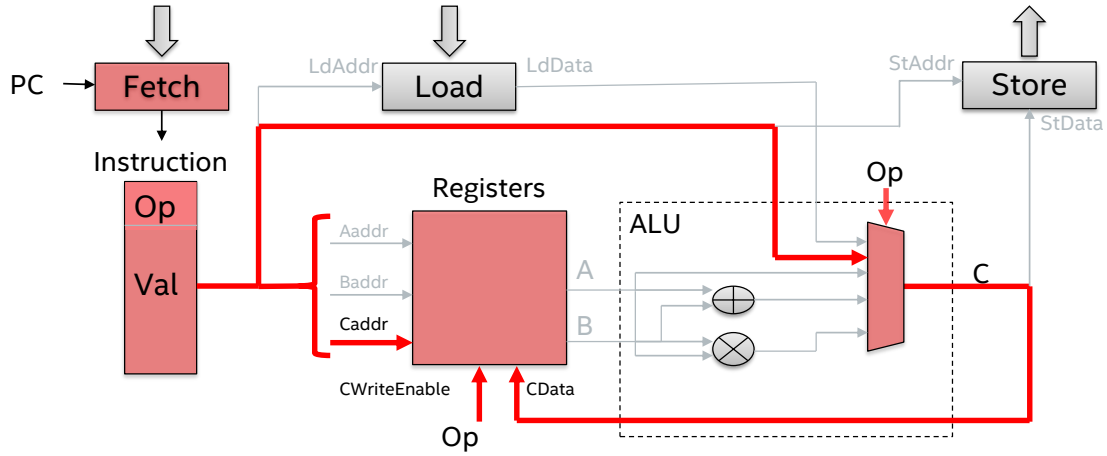
First let's take a look at execution on a *simple* CPU



Fixed and general architecture:

- **General "cover-all-cases" data-paths**
- **Fixed data-widths**
- **Fixed operations**

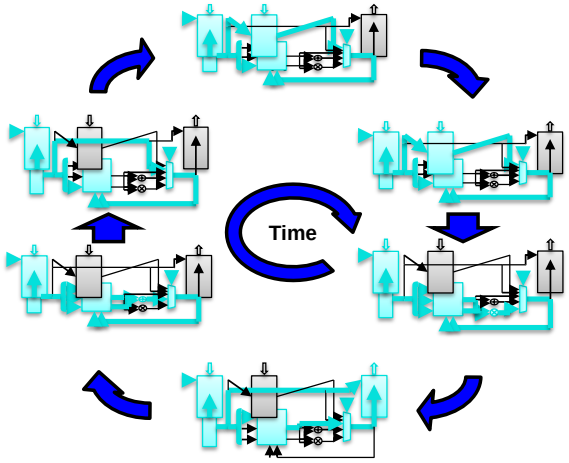
Looking at a Single Instruction



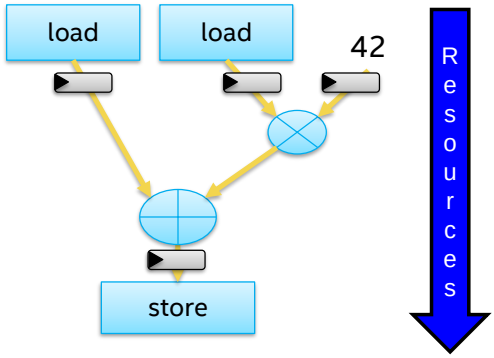
Very inefficient use of hardware!

Sequential Architecture vs. Dataflow Architecture

Sequential CPU Architecture



FPGA Dataflow Architecture

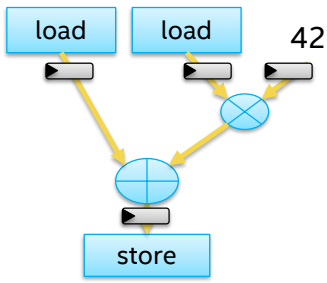


Custom Data-Path on the FPGA Matches Your Algorithm!

High-level code

```
Mem[100] += 42 * Mem[101]
```

Custom data-path



Build exactly what you need:

Operations

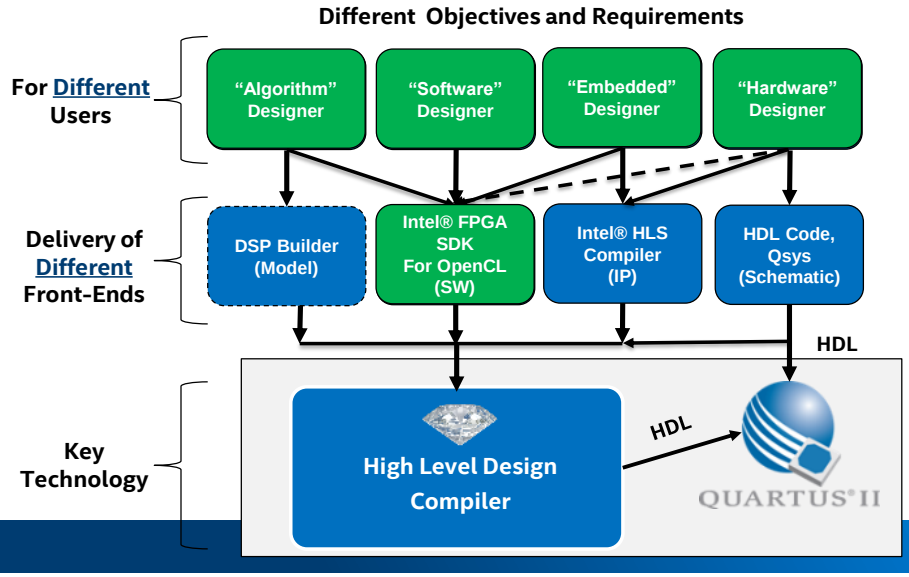
Data widths

Memory size & configuration

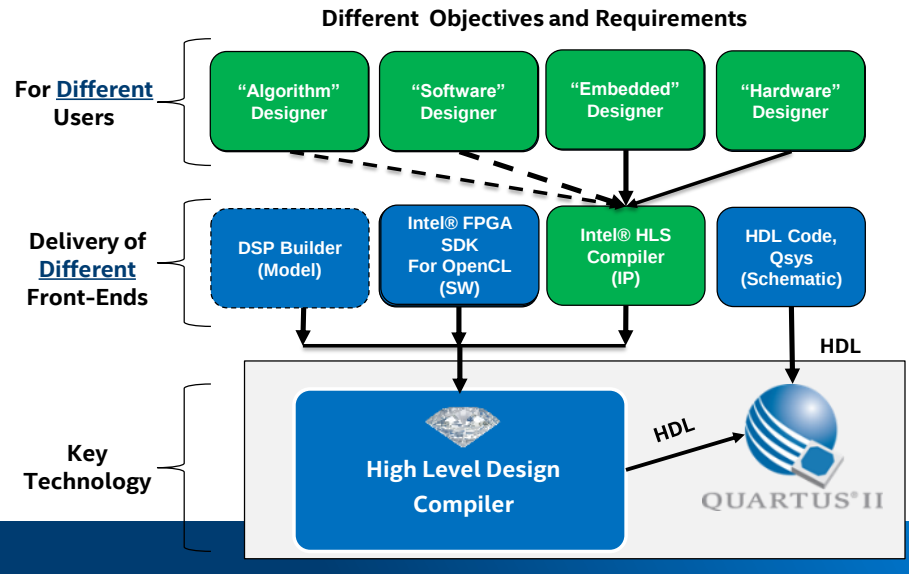
Efficiency:

Throughput / Latency / Power

Different Solutions for Different Users



Different Solutions for Different Users



What is OpenCL?

A software programming model for software engineers and a software methodology for system architects

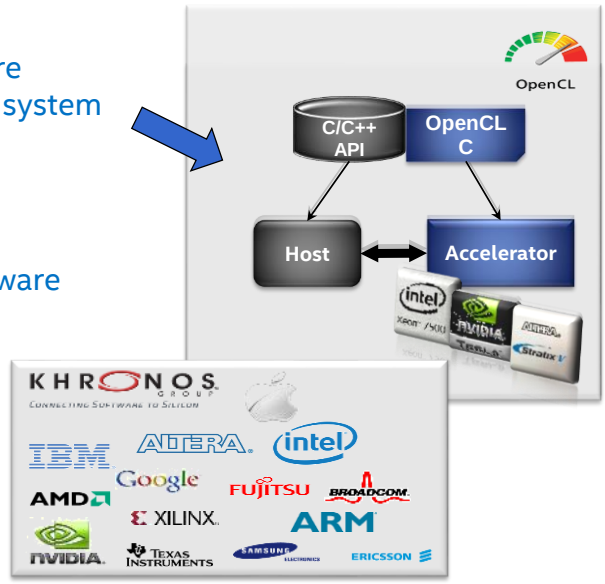
- First industry standard for heterogeneous computing

Provides increased performance with hardware acceleration

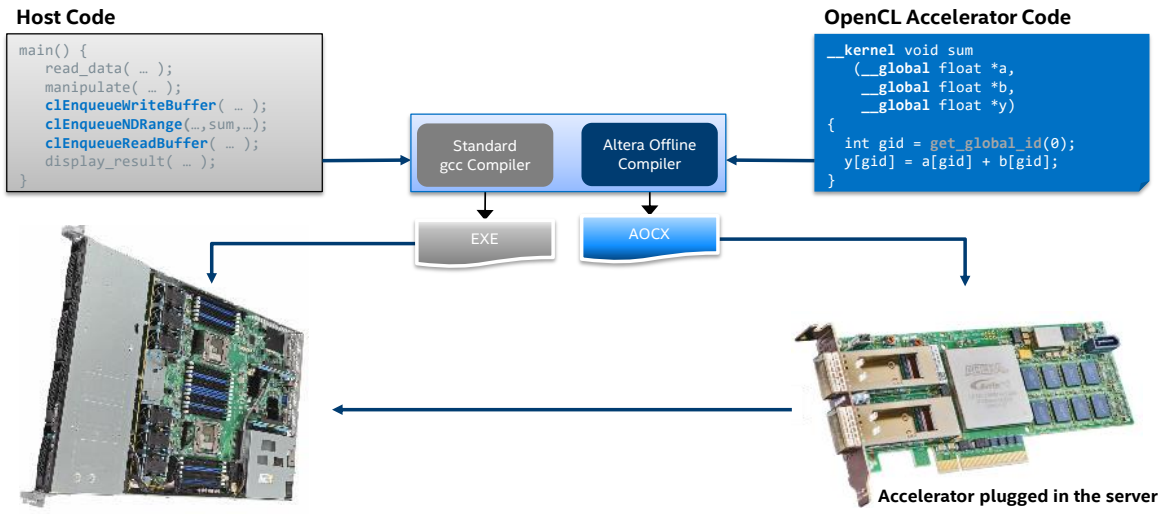
- Low Level Programming language
- C99

Open, royalty-free, standard

- Created by Apple
- Managed by Khronos Group
- Intel active member
- Conformant to the standard



OpenCL Use Model: Abstracting the FPGA away



Traditional OpenCL Host Program

Pure software written in standard C/C++ languages

Communicates with the accelerator devices via an API which abstracts the communication between the host processor and the kernels

Copy data from Host to FPGA

Tell the FPGA to run a particular kernel

Copy data from FPGA to Host

```
main()
{
   read_data_from_file( … );
   manipulate_data( … );

   clEnqueueWriteBuffer( … );
   clEnqueueNDRange(…, sum, …);
   clEnqueueReadBuffer( … );

   display_result ( … );
}
```

OpenCL Kernels

Kernel: Data-parallel function

- Defines many parallel threads
  - Explicitly or inferred
- Keyword extensions to specify parallelism and memory hierarchy

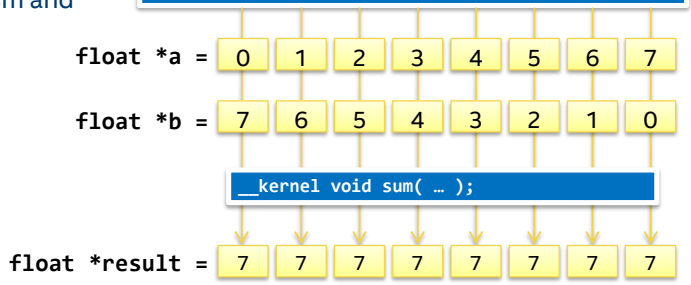
Executed by an OpenCL device

- CPU, GPU, FPGA, DSP

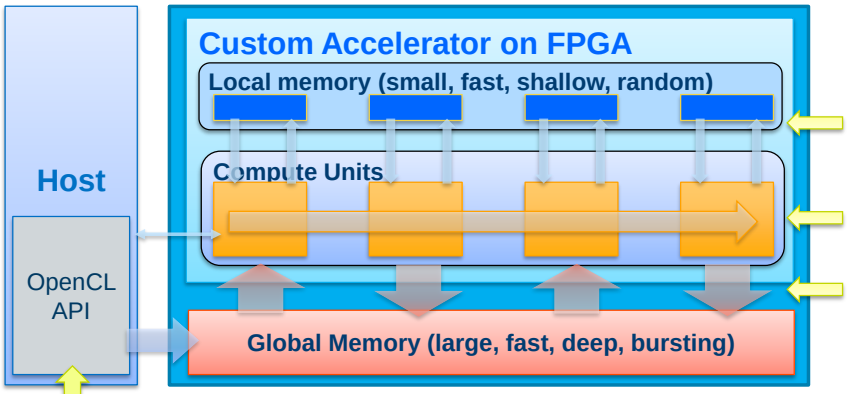
Code portable NOT performance portable

- **Between FPGAs it is!**

```
__kernel void sum(
    __global float *a,
    __global float *b,
    __global float *answer)
{
  int xid = get_global_id(0);
  result[xid] = a[xid] + b[xid];
}
```



Software Programmer's View of an OpenCL Platform



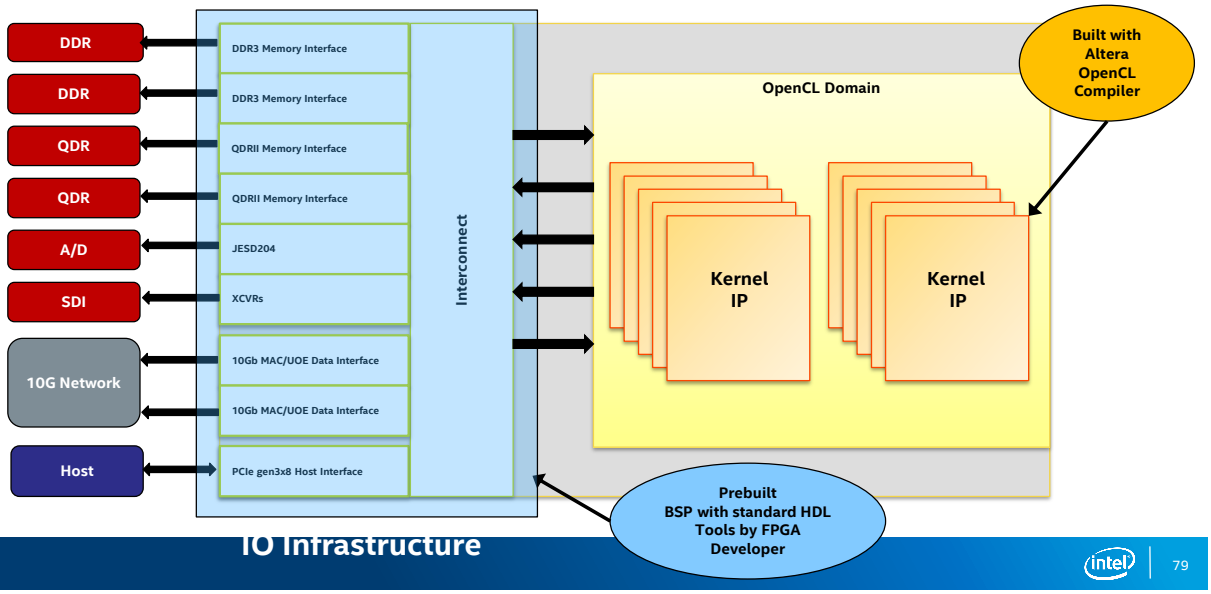
Custom local memory architecture: banking and arbitration

Custom compute units that run the kernels

Custom load/store units and interconnect

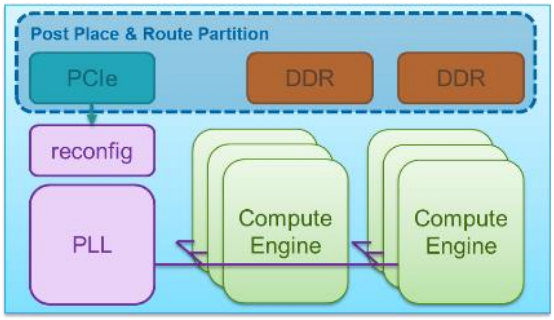
Host talks to global memory through OpenCL API routines

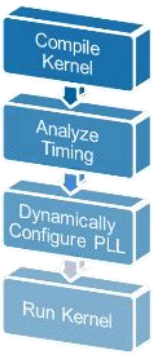
Board Support Package



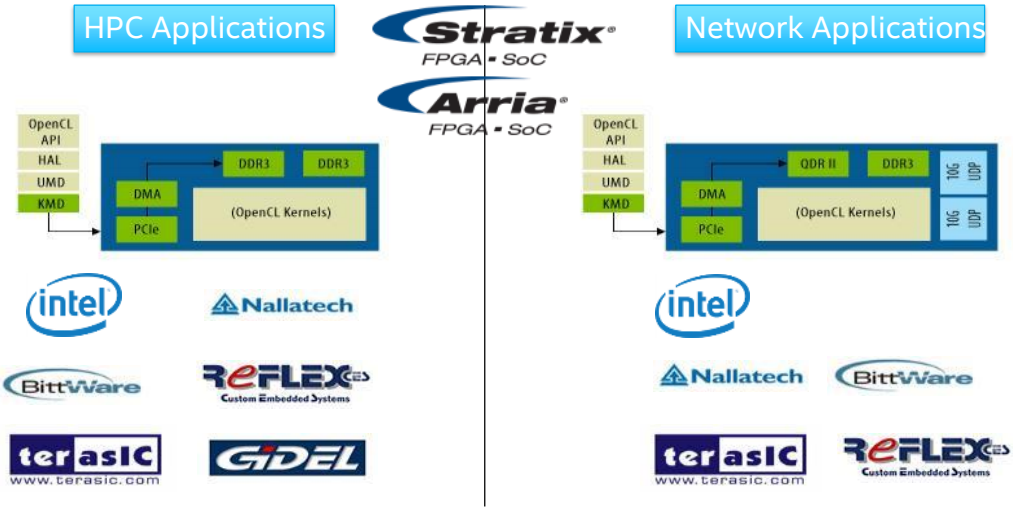
Guaranteed Timing Closure

- **Some interfaces have required clock frequencies**
  - **PCIe** 125 MHz / 250 MHz
  - **DDR3-1600** 800 MHz
  - **Kernel** ??





Start with OpenCL ready platforms 1/2



Bittware Arria 10 Gx Specifications

Intel Arria 10 GX FPGA - 10AX115N3F40E2SG

- Up to 1150K logic elements available
- Up to 53 Mb of embedded memory
- Up to 3,300 18x19 variable-precision multipliers

PCIe x8 interface supporting Gen1, Gen2, or Gen3

Dual QSFP cages for 2x 40GbE or 8x 10GbE

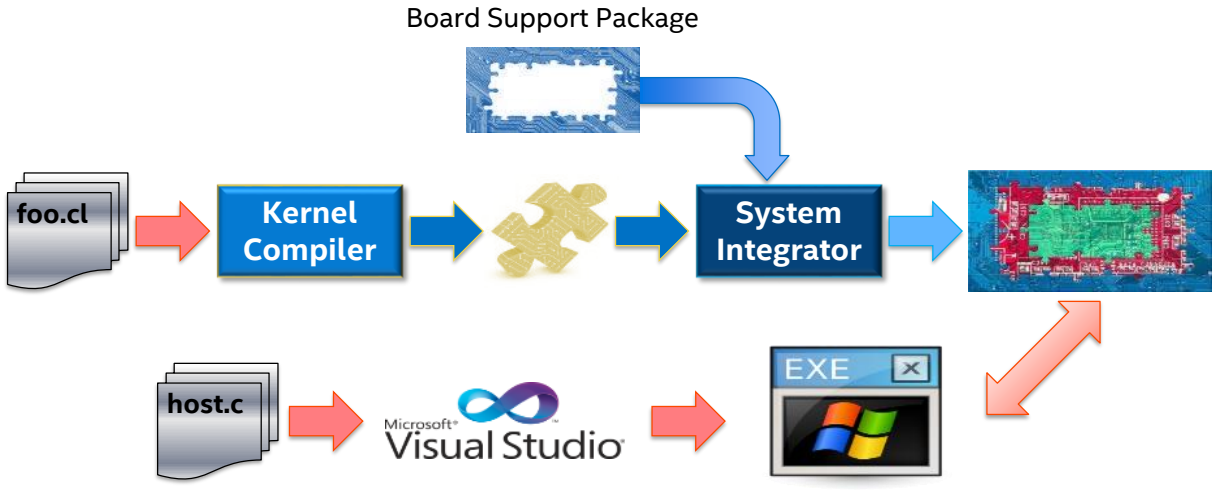
Up to 32 GBytes of DDR4 SDRAM with ECC (x72)

BMC for Intelligent Platform Management

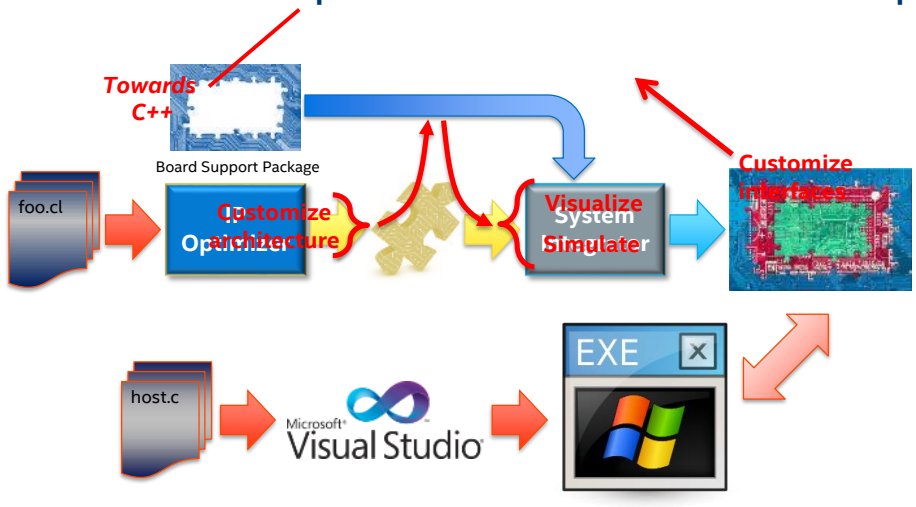
Precision clock and timing options

Utility I/O: USB 2.0

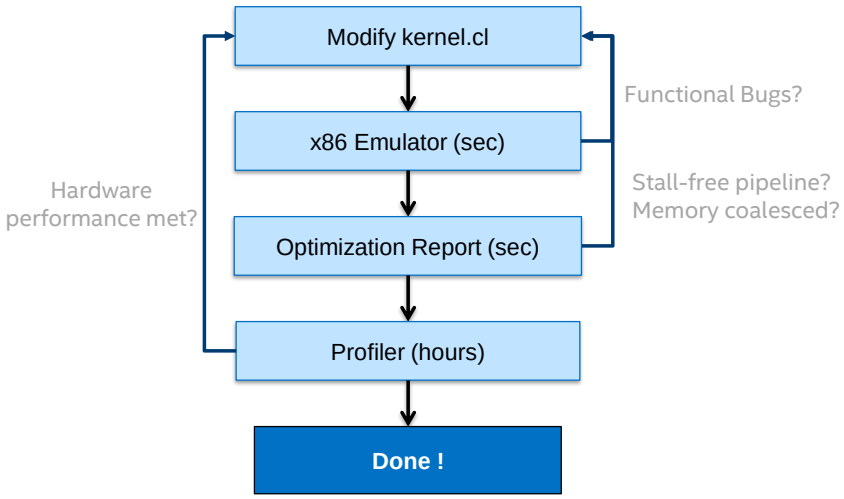
Development Flow for FPGA



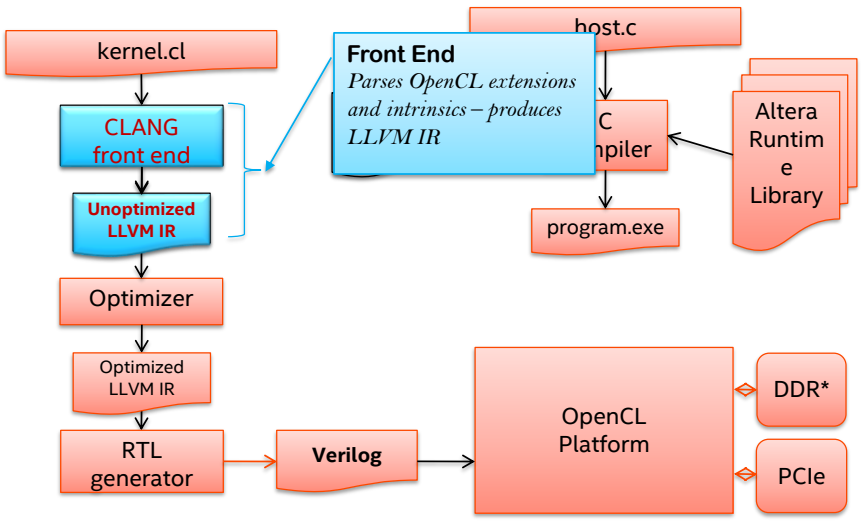
How Does i++ Compiler for HLS Differ From OpenCL?



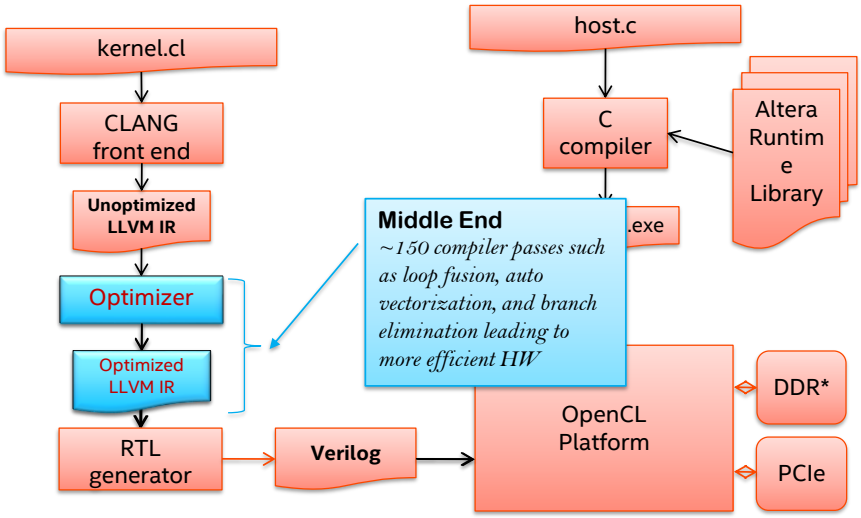
Multi-Step AOC Compilation



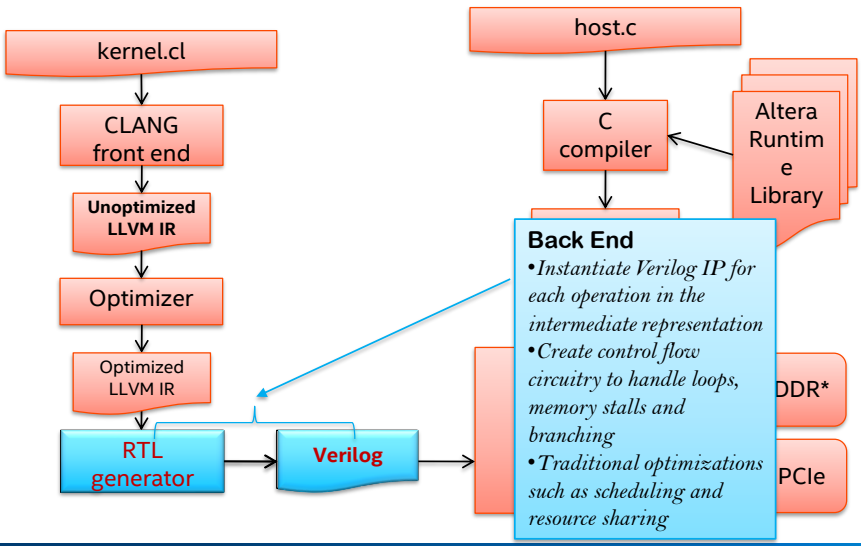
OpenCL Compiler Flow



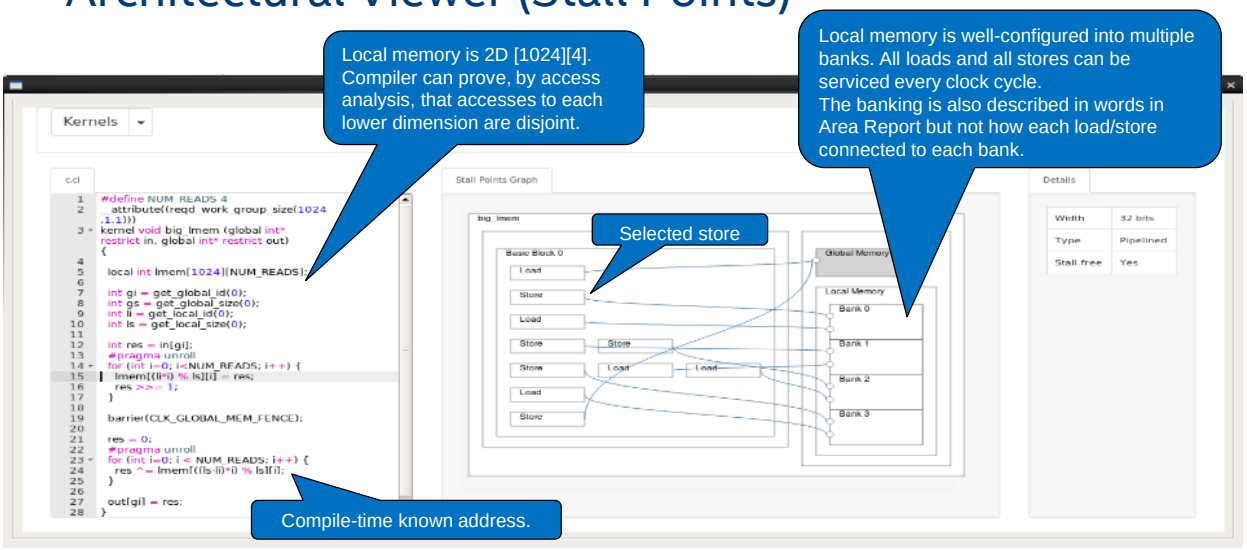
OpenCL Compiler Flow



OpenCL Compiler Flow



Architectural Viewer (Stall Points)



Area Report

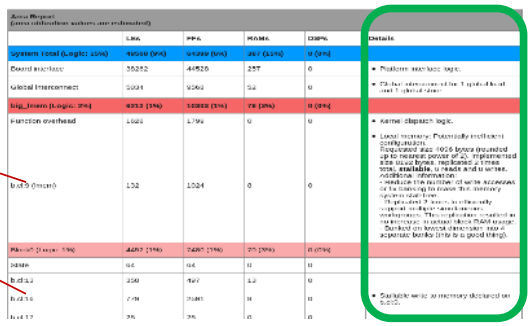
Shows estimated resource usage for each line of code

Provides details explaining reason for inefficiencies

Information on how to improve your design

Enhanced accuracy of estimates and relationship to code

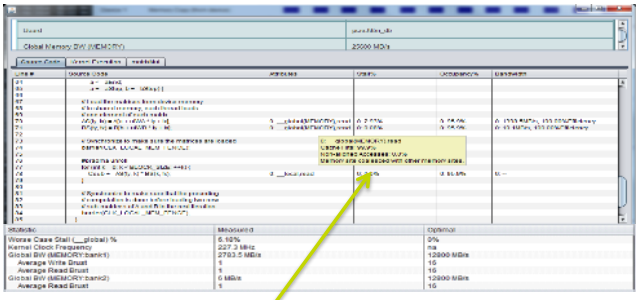
```
1 #define NUM_READS   8
2 #define NUM_WRITES  8
3 #define NUM_BARRIERS 2
4
5 __attribute((reqd_work_group_size(1024,1,1)))
6 kernel void big_lmem (global int* restrict in,
7                        global int* restrict out) {
8
9   local int lmem[1024];
10  int gi = get_global_id(0);
11  int gs = get_global_size(0);
12  int li = get_local_id(0);
13  int res = in[gi];
14  #pragma unroll
15  for (int i=0; i<NUM_WRITES;i++) {
16    lmem[li - i] = res;
17    res >>= 1;
18  }
19  barrier(CLK_GLOBAL_MEM_FENCE);
20  res = 0;
21  #pragma unroll
22  for (int i=0; i < NUM_READS; i++) {
23    res ^= lmem[li - i];
24  }
25  out[gi] = res;
26 }
```



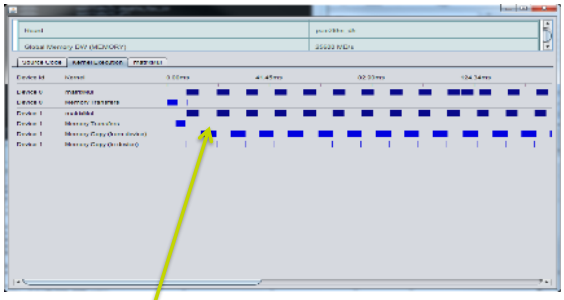
Dynamic Profiler

Intel FPGA SDK for OpenCL enables users to get runtime information about their kernel performance

Bottlenecks, bandwidth, saturation, pipeline occupancy



Performance Stats



Execution Times

Key Optimization Methodology is Still Data Localization

10 TFLOP floating point performance in Startix 10

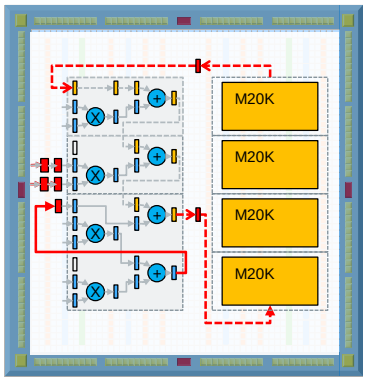
- **Use every DSP, every clock cycle compute spatially**

>8 TB/s memory bandwidth: keep state on chip!

- Exceeds available external bandwidth by **orders of magnitude**
- Random access, low latency (2 clks)
- **Place all data in on-chip memory compute temporally**

Avoid costly data movement

- Highest performance/W of Intel's programmable offerings

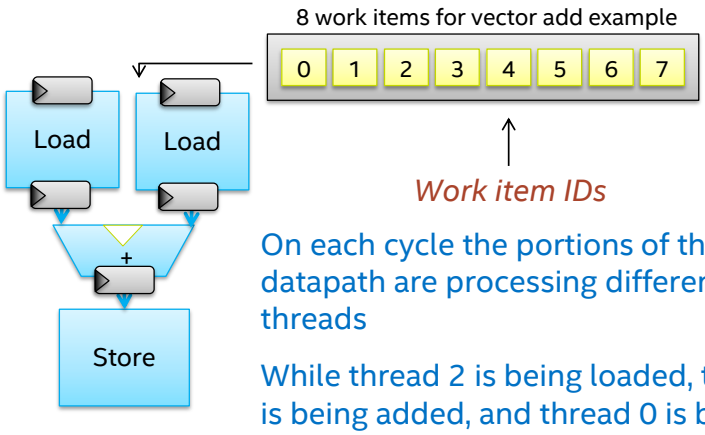


Fine-grained & low latency between compute and memory

Mapping Multithreaded Kernels to FPGAs

- **The most simple way of mapping kernel functions to FPGAs is to replicate the unrolled hardware for each thread**
  - Inefficient and wasteful
- **Better method involves taking advantage of *pipeline parallelism***
  - Attempt to create a deeply pipelined representation of a kernel
  - On each clock cycle, we attempt to send in input data for a new thread
  - Method of mapping coarse grained thread parallelism to fine-grained FPGA parallelism

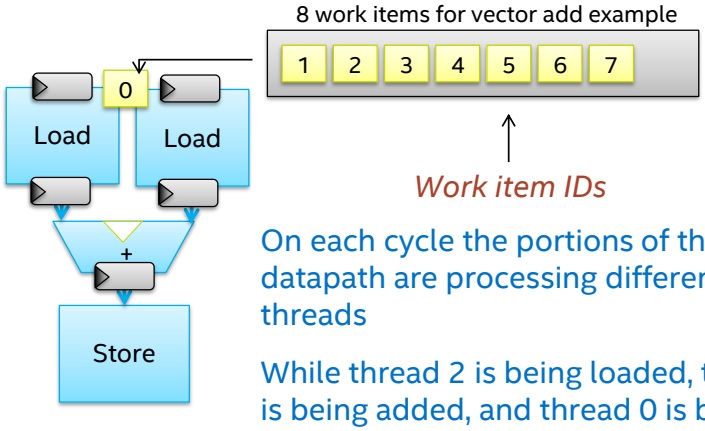
Example Datapath for Vector Add



On each cycle the portions of the datapath are processing different threads

While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

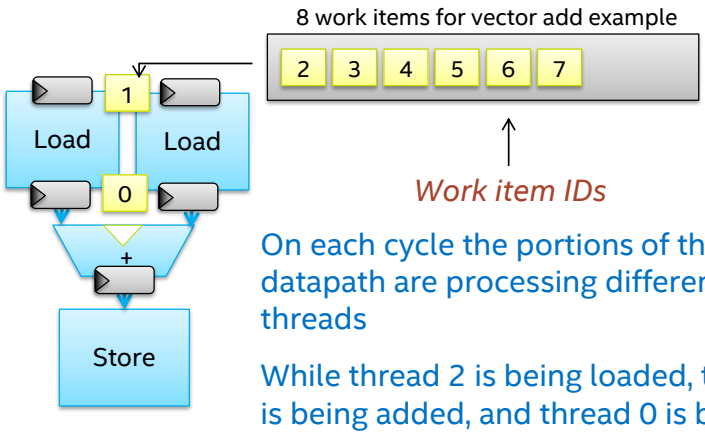
Example Datapath for Vector Add



On each cycle the portions of the datapath are processing different threads

While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

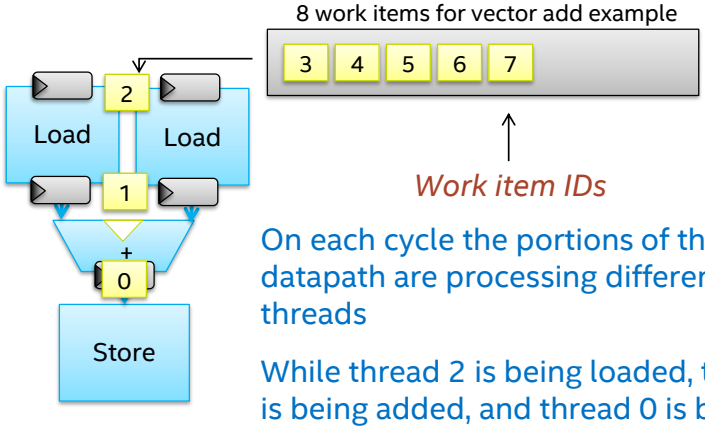
Example Datapath for Vector Add



On each cycle the portions of the datapath are processing different threads

While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

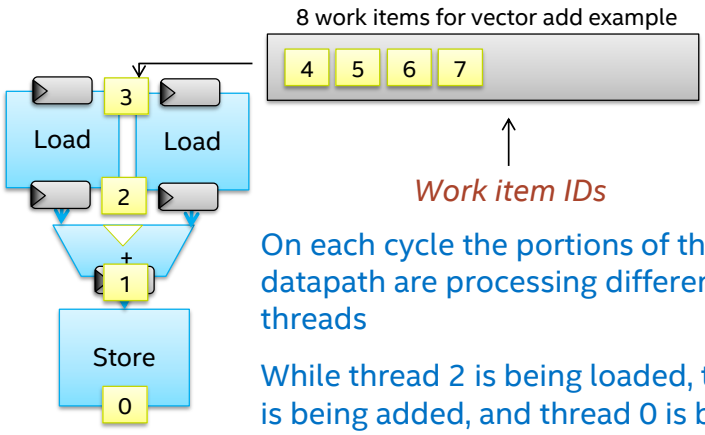
Example Datapath for Vector Add



On each cycle the portions of the datapath are processing different threads

While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

Example Datapath for Vector Add



On each cycle the portions of the datapath are processing different threads

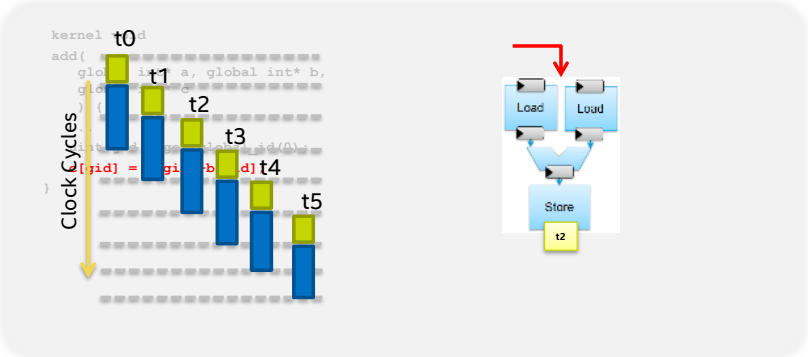
While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

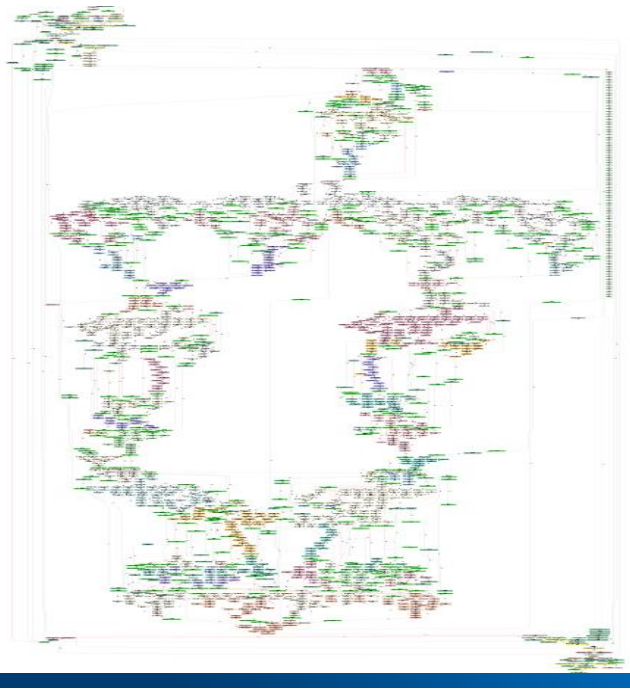
Silicon used efficiently at steady-state

Execution of Threads on FPGA

Better method involves taking advantage of *pipeline parallelism*

- ***Throughput = 1 thread per cycle***



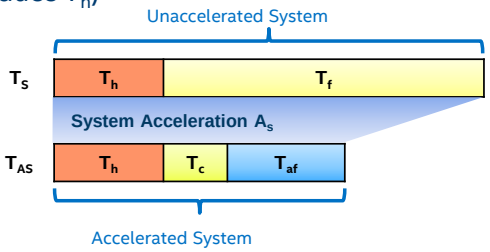


OPTIMIZING KERNEL FOR FPGA

Goals of OpenCL Optimization

Keep as much FPGA resources busy as possible doing useful stuff!

- Increase the number of parallel operations (Reduce T_{af})
- Reduce communication latency of the accelerated system (Reduce T_c)
- Increase efficiency of operations (Reduce T_{af})
- Minimize overhead (Reduce T_h)



Optimization technics

Loop Unrolling

Vectorization

Kernel Compute duplication

Task Kernel: Single Work-Item Kernels

Channels

NDRange Execution

Memory Optimizations: Global, Local & Private

Floating Point Optimizations

Loop Unrolling

By default, loops can hinder performance

Loop unrolling replicate hardware to execute multiple loop iterations at once

Increase performance by decreasing number of iteration through loop

- Structure of the compute units built will be significantly altered
- More resource consumption
- More local memory ports may be required

Simple loops will be unrolled automatically

- AOC will generate a warning

`unroll` kernel pragma

`#pragma unroll <N>` instructs AOC to attempt to unroll a loop <N> times

- Without <N>, AOC will attempt to unroll the loop fully
- Warning issued if AOC unable to unroll

```
#pragma unroll 2
for (size_t k=0; k<4; k++)
{
     mac += data_in[(gid*4)+k] * coeff[k];
}
```

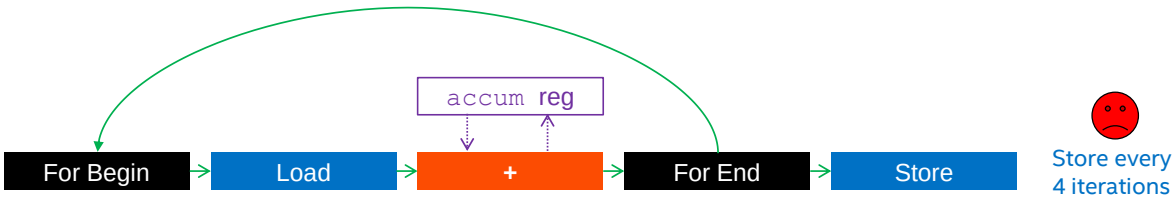
Control the amount of hardware used for loops

Loop Unrolling Example

Sum of 4 values for every work-item

Store a new result every 4 iterations

```
accum = 0;

for (size_t i=0; i<4; i++)
{
     accum += data_in[(gid*4)+i];
}
sum_out[gid] = accum;
```

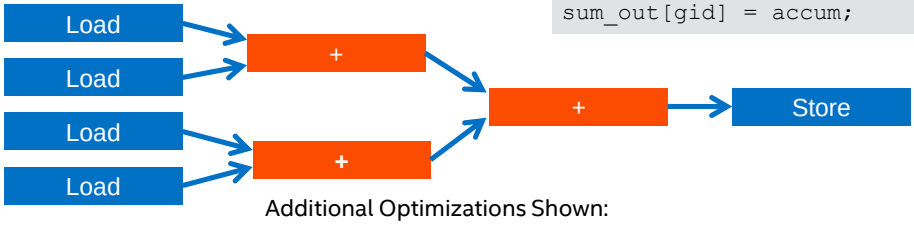


Loop Unrolling Example: Fully Unrolled

Unroll every iteration of the loop

Store a new result every clock cycle

```
accum = 0;
#pragma unroll
for (size_t i=0; i<4; i++)
{
    accum += data_in[(gid*4)+i];
}
sum_out[gid] = accum;
```



Store every cycle

Additional Optimizations Shown:

1. `accum` register removed
2. Order of operation optimization done if allowed
3. Operators removed if not needed
   - There would be 4 adders created if initial value of `accum` is not 0.

Loop Unrolling in the Optimization Report

Loop unrolling reported in the <kernel file>.log

Reported information

- Loop location
- Nesting relationship
- Requested unroll factor
- Achieved unroll factor

```
Loop Report:

 + Fully unrolled loop (file nd_full_nested.cl line 4)
 |  Loop was automatically and fully unrolled.
 |  Add "#pragma unroll 1" to prevent automatic unrolling.
 |
 |
 |-+ Fully unrolled loop (file nd_full_nested.cl line 6)
     Loop was fully unrolled due to "#pragma unroll" annotation.
```

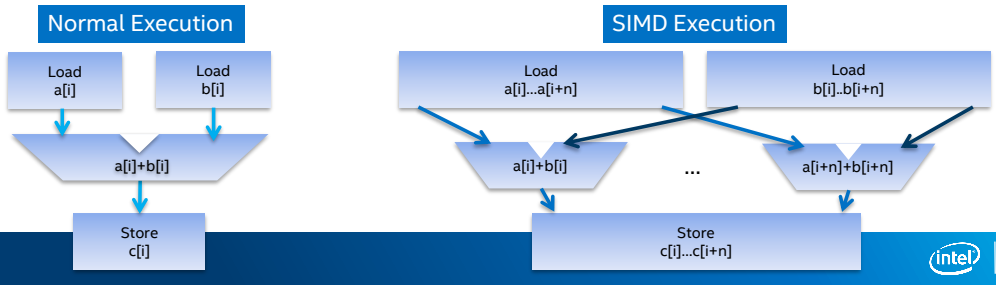
Kernel Vectorization

Widen the pipeline to achieve higher throughput

- Allow multiple work-items from the same workgroup to execute in Single Instruction Multiple Data (SIMD) fashion

Translate scalar operations into SIMD operations

- E.g. multiplication or addition
- Can be done manually or automatically



Vectorize Kernel Code Manually

Replicate operations in the kernel manually

Must also adjust NDRange in host application

```
__kernel void mykernel (…)
{
     size_t gid = get_global_id(0);
     result[gid] = in_a[gid] + in_b[gid];
}
```

Original Kernel

```
__kernel void mykernel (…)
{
     size_t gid = get_global_id(0);
     result[gid*4+0] = a[gid*4+0] + b[gid*4+0];
     result[gid*4+1] = a[gid*4+1] + b[gid*4+1];
     result[gid*4+2] = a[gid*4+2] + b[gid*4+2];
     result[gid*4+3] = a[gid*4+3] + b[gid*4+3];
}
```

Manually Vectorized Kernel

Automatic Kernel Vectorization

Use attribute to enable automatic kernel compute unit vectorization

- Memory accesses automatically coalesced
- No need to adjust NDRange in host application

num_simd_work_items attribute

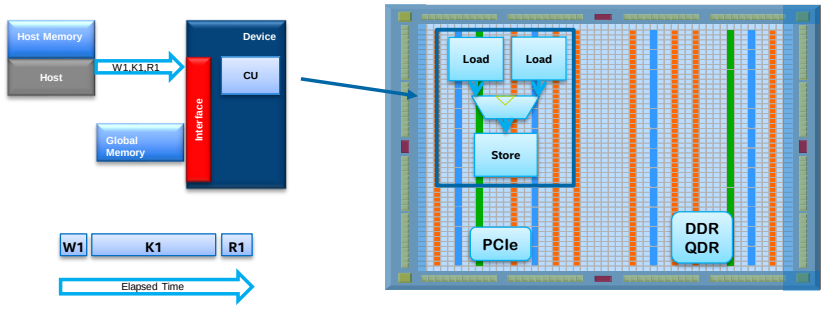
- Specify the number of work-items within a workgroup to be executed in a SIMD manner
  - Hardware operators automatically vectorized
- Vectorization only takes affect in the X dimension of the workgroup
- Must use with reqd_work_group_size attribute
  - reqd_work_group_size must be evenly divisible by num_simd_work_items in the X dimension
- Factor of 2,4,8,16

```
__attribute__((num_simd_work_items(4)))
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void mykernel (…)
{…}
```

Default Compute Units

Only one compute unit per kernel created by default

Workgroups distributed to compute unit in sequence



Multiple Compute Units

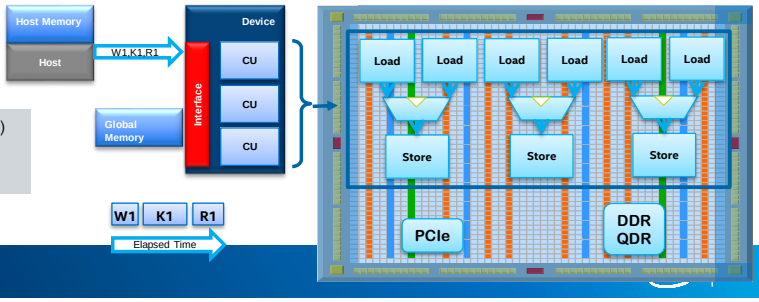
Use `num_compute_unit` attribute to specify number of kernel compute units to

- Entire compute unit including all local memory, control logic, and operators replicated

Workgroups from the same NDRange kernel launch are distributed to available compute units and processed in parallel

- Need at least three times as workgroups as compute units to effectively utilize all hardware

```
__attribute__((num_compute_units(3))
)
__kernel void …
```



118

Example: Combining Replication and Vectorization

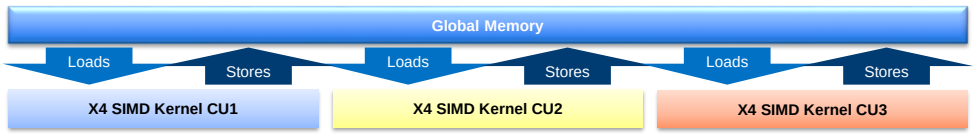
Resource estimates of 16 SIMD lanes indicate “no fit”

Resource estimates of 8 SIMD lanes suggest 12 lanes may fit

- Automatic vectorization only supports 2, 4, 8 and 16 lane configurations

Generate 12 lanes by combining num_simd_work_items and num_compute_units

```
__attribute__((num_simd_work_items(4)))
__attribute__((num_compute_units(3)))
__attribute__((reqd_work_group_size(8,8,1)))
__kernel void mykernel (…)
{ …
```



intel 119

Kernel Attributes in the Optimization Report

Effects of all kernel attributes are reported in the <kernel file>.log file

```
================================================================================
Kernel: MovAv
================================================================================
The kernel is compiled as an ND-Range.

The kernel was vectorized for 2 SIMD work-items along the lowest dimension.

The kernel was replicated 2 times due to num_compute_units attribute.
You should have at least three times as many work-groups as the number of compute units
to efficiently utilize all compute units.

The kernel has a required work-group size of (128, 1, 1).

================================================================================
```


intel experience what's inside™

TASK KERNEL: SINGLE WORK ITEM

JDEV2017

Single Work-Item Execution

Launching kernels with NDRange of (1,1,1)

- A kernel executed on a compute unit consisting of one work-item
- Defined as a Task in OpenCL

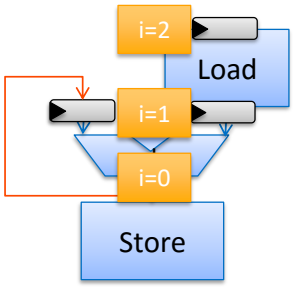
Why?

- Data parallel (multiple work-items) execution may not be suitable for certain situations
  - Difficulties partitioning data among parallel works-items
  - Dependency across work-items
  - Data not available prior to kernel execution
  - Data not easily divided into workgroups
- Sequential programming model of tasks more similar to C programming
  - Certain usage scenario more suited for sequential programming model
  - Easier to port

Tasks and Loop-pipelining

Allow users to express programs as a single-thread kernel

```
for (int i=1; i < n; i++) {
     c[i] = c[i-1] + b[i];
}
```



Compiler will infer parallel pipelined execution across loop iterations

- Pipeline parallelism still leveraged to efficiently execute loops in Altera's OpenCL solution
- Dependencies resolved by the compiler
- Values transferred between loop iterations with FPGA resources
  - No need to buffer up data
  - Easy and cheap to share data through feedbacks in the pipeline

Loop Pipelining

AOC will pipeline each iteration of the loop for acceleration

- Analyze any dependencies between iterations
- Schedule these operations
- Launch the next iteration as soon as possible

```
float array[M];

for (int i=0; i < n; i++)
{
   for (int j=0; j < M-1; j++)
     array[j] = array[j+1];
   array[M-1] = a[i];

  for (int j=0; j < M; j++)
    answer[i] += array[j] * coefs[j];
}
```

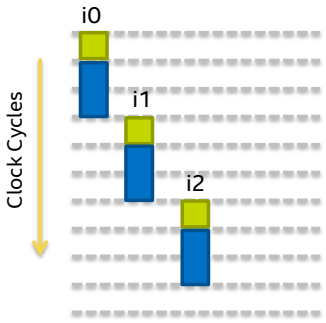
Shift Register `array`
(Dependency for next iteration)

At this point, launch the next iteration

Reduction on `array`
(Not a dependency)

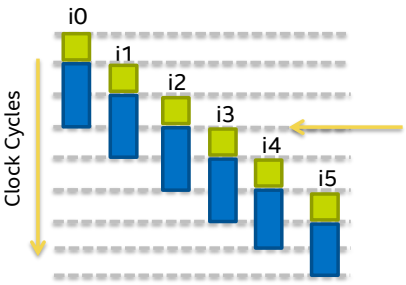
Loop Pipelining Example

No Loop Pipelining



No Overlap of Iterations!

With Loop Pipelining

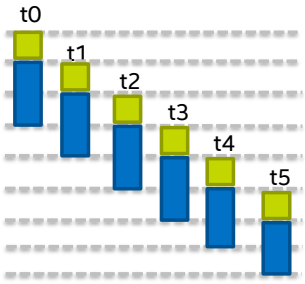


Looks like multi-threaded execution!

Finishes Faster because Iterations Are Overlapped

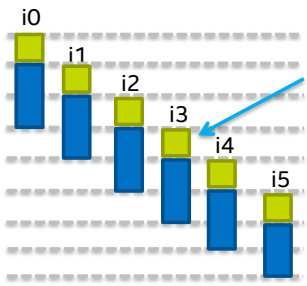
Parallel Threads vs Loop Pipelining

Parallel Threads



Parallel threads launch 1 thread per clock cycle in pipelined fashion

Loop Pipelining



Loop dependencies may not be resolved in 1 clock cycle

Loop Pipelining enables Pipeline Parallelism AND the communication of state information between iterations.

- If dependency can be resolved in 1 clock cycle, then the resulting computational throughput is the same

Communication: Channels/Pipes

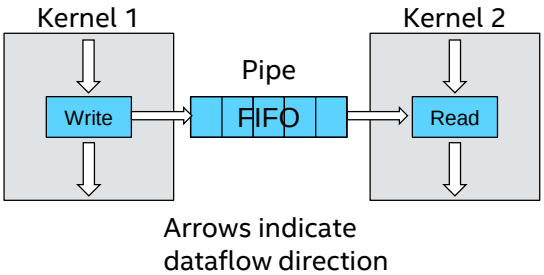
Key for systolic array architectures

Enables much lower latency data movement through data processing paths (result reuse)

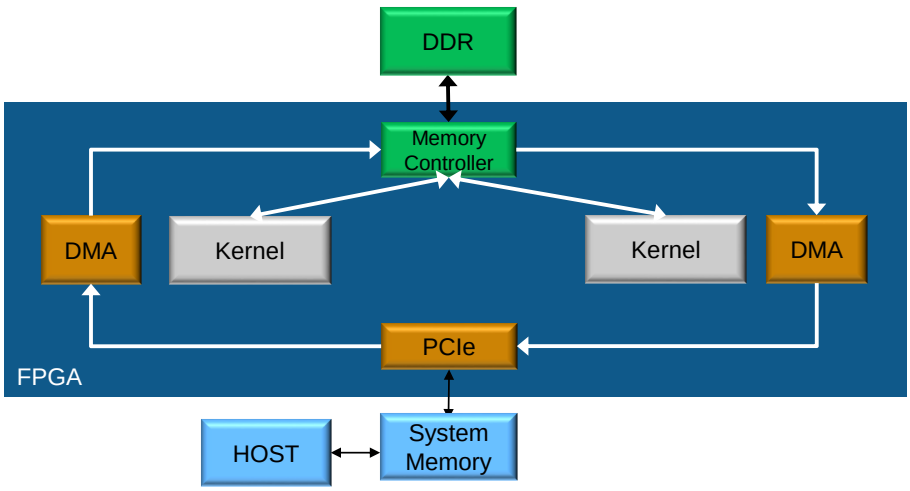
- No need to move data back and forth to external memory or cache

Provide unidirectional, FIFO-like communication into or out of kernels, or into or out of FPGA

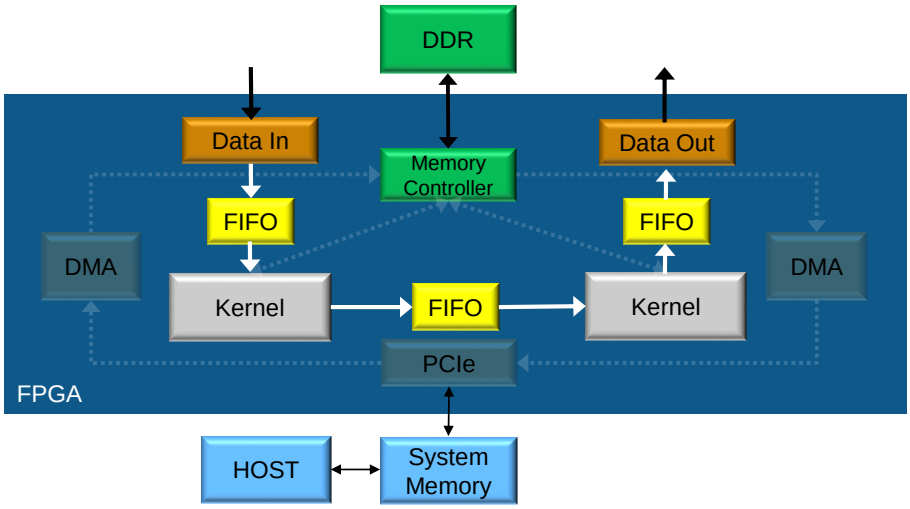
- Pipes are built on top of channels and only support kernel to kernel data movement



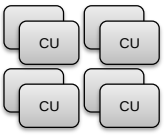
Traditional Data Movement Without Channels



Data Movement Using Channels



Systolic Array of Compute Units



Replicate kernel hardware with num_compute_units(X,Y,Z) attribute

- Creates X*Y*Z copies of kernel pipeline
  - Increases throughput
  - When applied to NDRange kernels, these copies used to execute multiple workgroups in parallel
    - More on this in the Optimizing NDRange kernels section
  - Consumes X*Y*Z times more resources for that kernel compute unit

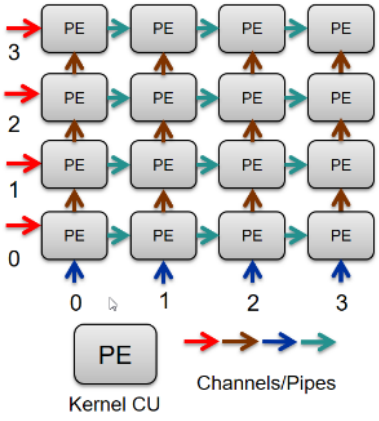
With single work-item kernels, AOC allows customization of kernel compute units using the get_compute_id() function

- Create compute ID dependent logic

Example with num_compute_units

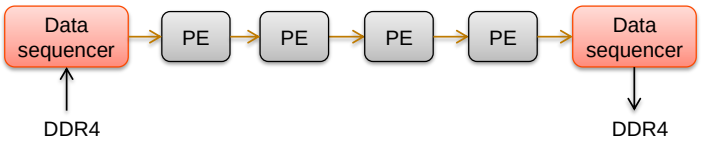
Using compute ID to determine channel usage

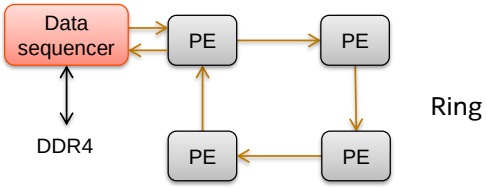
```
channel float4 ch_PE_row[3][4];
channel float4 ch_PE_col[4][3];
channel float4 ch_PE_row_side[4];
channel float4 ch_PE_col_side[4];

__attribute__((autorun))
__attribute__((max_global_work_dim(0)))
__attribute__((num_compute_units(4,4)))
kernel void PE() {
      float4 a,b;
      if (get_compute_id(0)==0) //First PE of row
            a = read_channel(ch_PE_col_side[col]);
      else
            a = read_channel(ch_PE_col[row-1][col]);

      if (get_compute_id(1)==0)
      …
}
```



Other Topologies Possible in FPGA Too

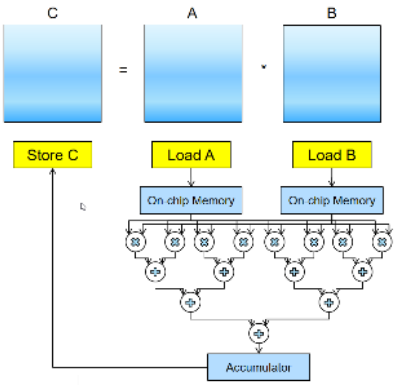
Linear Pipeline / Daisy chain

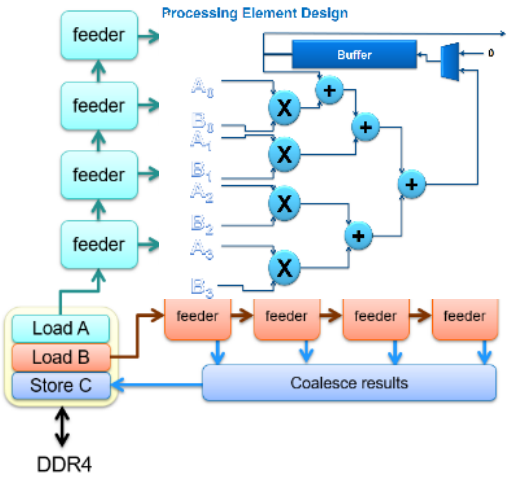




Ring

Use Case: Matrix Multiplication





Performance: ~1 TFLOPs

intel

MEMORY OPTIMIZATIONS

Global Memory & Local Memory

JDEV2017

OpenCL Global Memory

Global Memory Overview

- Programmers view of global memory in OpenCL
- Compiler view of global memory

Global Memory Architecture

- Load-Store Units
- LSU Arbitration
- Const Cache

Optimizations

Global Memory in OpenCL

'global' address space

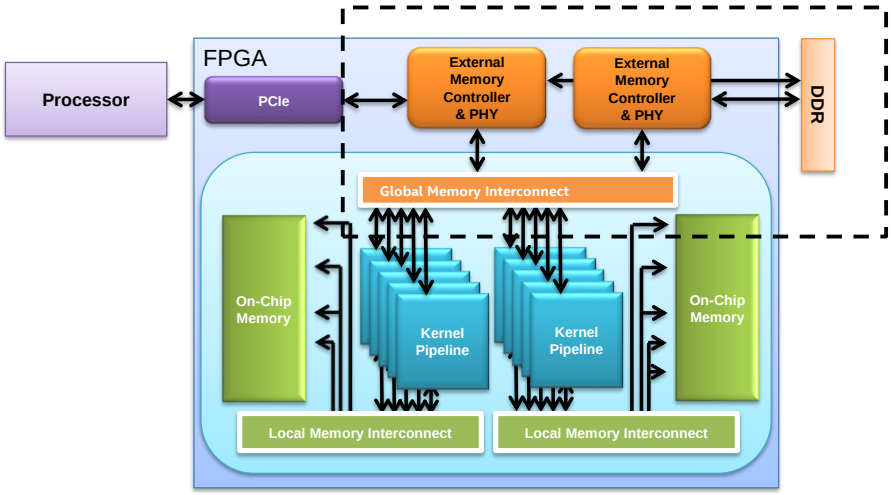
- Used to share data between host and device
- Shared between workgroups

```
__kernel void Add(__global float* a,
                  __global float* b,
                  __global float* c)
{
   int i = get_global_id(0);
   c[i] = a[i] + b[i];
}
```

Generally allocated on host as cl_mem object

- Created with clCreateBuffer
- Data transferred with clRead/clWrite Buffer
- cl_mem object assigned to global pointer argument in kernel

OpenCL BSP Global Memory



Compiler's View of Global Memory

Agnostic to the memory technology itself

- DDR, QDR, HMC, QPI

Only a few pertinent parameters (provided by BSP)

- How many interfaces
- Width of the bus
- Burst size (affinity for linear access)
- Latency
- Bandwidth

Exposed as Avalon Memory Mapped Slave

- Compiler builds datapath and interconnect to communicate to fixed IP
- BSP developer can create BSP variants for different memory configurations

OpenCL Global Memory

Global Memory Overview

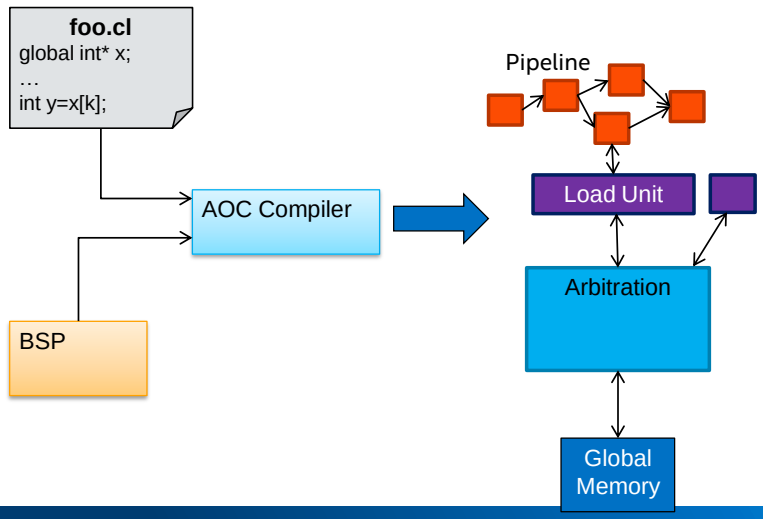
- Programmers view of global memory in OpenCL
- Compiler view of global memory

Global Memory Architecture

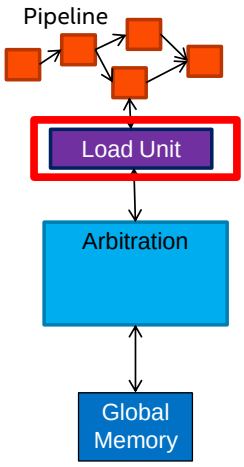
- Load-Store Units
- LSU Arbitration
- Const Cache

Optimizations

Compiler Generated Hardware



Global Load/Store Unit (LSU)



Width Adaptation

- User data (32-bit int) **to** memory word (512-bit DRAM word)
- Coalesces to avoid wasted bandwidth

Burst coalescing

- Coalesces consecutive memory transactions into large burst transaction

LSU Types (1)

Pipelined

- Used for local memory

Simple

- Passes transactions to interconnect from pipeline
- Used for loads/stores used very infrequently

Burst-Coalesced

- Most common global memory LSU
- Specialized LSU to groups loads/stores into bursts
- Load LSU can cache/re-use data
  - Private caching is applied heuristically

Streaming

- Simplified LSU used if compiler can determine access pattern is completely linear

LSU Types (2)

Semi-streaming

- Load LSU Specialized for nearly linear accesses
- Instantiated heuristically

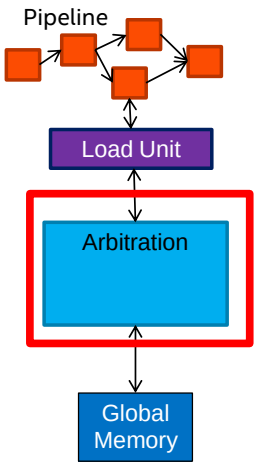
Burst-non-aligned

- Wrapper around Burst-coalesced LSU, instantiated if access alignment may not be aligned to LSU width
- Tries to minimize overhead of non-aligned accesses by coalescing

Wide LSU

- Wrapper around other LSU types, converts wider than global memory accesses into global memory sized accesses
- More area efficient than multiple memory width LSUs

Arbitration



Arbitrate to physical interfaces

- Tree interconnect (high bandwidth)
- Ring interconnect (high fmax)
  - Increase reliance on large bursts
- Arbitration type chosen base on # of LSUs

Distribute (load balance) across physical interfaces

Const Cache

Constant buffer resides in global memory but accessed via on-chip cache shared by all work-groups

- Constant cache optimized for high cache hit performance

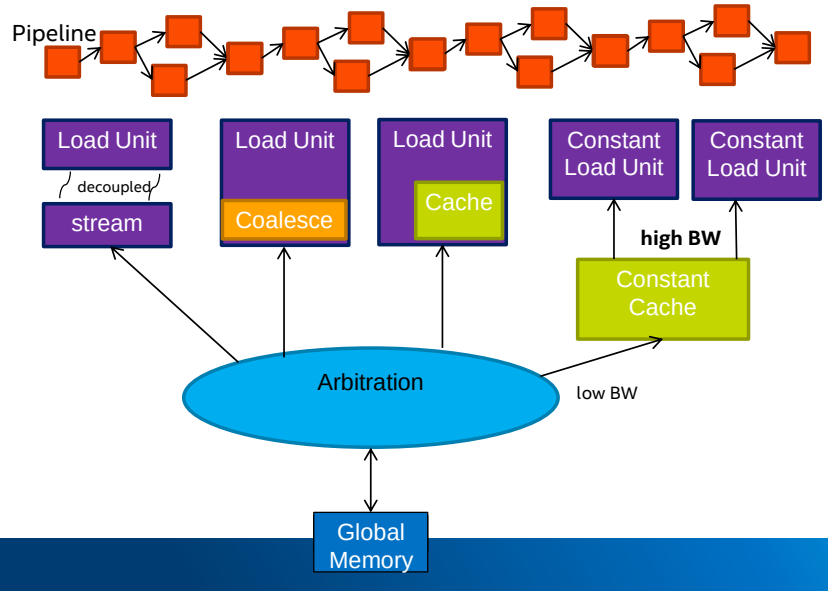
Use for read-only data that all work-groups access

- E.g. high-bandwidth table lookups

Constant cache default size is 16kB

- Uses on-chip RAM blocks that are shared with local memory

Complete Picture



OpenCL Global Memory

Global Memory Overview

- Programmers view of global memory in OpenCL
- Compiler view of global memory

Global Memory Architecture

- Load-Store Units
- LSU Arbitration
- Const Cache

Optimizations

Contiguous Memory Accesses

Interleaved memory suitable for contiguous access

- Sequential accesses to global memory are the ideal access pattern to increase memory efficiency

AOC will analyze access pattern of load and stores

Looks for sequential load and store operations for the entire kernel and directs kernel to access consecutive locations in global memory

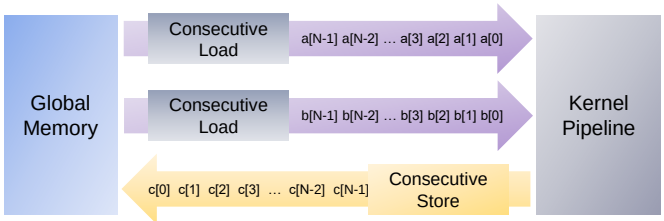
Leads to increased access speeds and reduced hardware resource needs

Array Indexing and Contiguous Memory Accesses

Basing array index on the work-item global ID leads to efficient contiguous load and store operations

- Data sent to and received from the kernel pipeline as needed
- Computation and memory accesses can happen simultaneously

```
__kernel void mykernel (…)  {
     size_t gid = get_global_id(0);
     c[gid] = a[gid] + b[gid];
}
```



Contiguous Memory Accesses (Tasks)

For Task kernels, memory should be indexed with an increasing loop counter for contiguous accesses

```
__kernel void mykernel (…)  {
     for(int i = 0; i<BUFFER_SIZE; i++)
     {
     c[i] = a[i] + b[i];
     }

}
```

Ensure 4-Byte Alignment for Data Structures

Struct alignments smaller than 4 bytes result in larger and slower hardware

```
typedef struct {
     char r,g,b,alpha;
} Pixel;
```

1-byte aligned struct

Force 4-byte alignment

Create a `union` of the structure and an integer

```
typedef struct {
     char r,g,b,alpha;
} Pixel_s;

typedef union {
     Pixel_s p;
     int not_used;
} Pixel;
```

or

Use aligned attribute with GCC or in the kernel

```
typedef struct {
     char r,g,b,alpha;
} __attribute__((aligned(4))) Pixel;
```


Using __constant Buffers

Constant cache default size is 16kB

- Uses on-chip RAM blocks that are shared with local memory

Manually specify cache size with AOC option

- Specify in bytes, AOC will round up to closest power of 2
- Have no effect if no kernels use __constant address space

Constants suffer huge penalties for cache misses

- Use __global const instead if constant argument can't fit in the cache

```
aoc --const-cache-bytes 32768 mykernel.cl
```

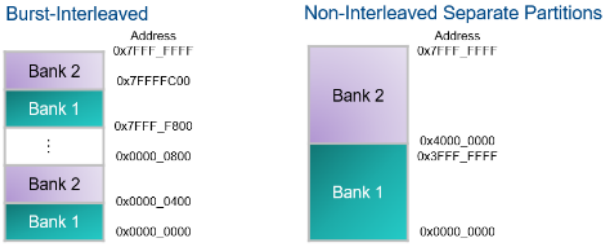
Global Memory Banking Optimizations

Address space can be partitioned between banks or finely interleaved

Default: Configures global memory as burst-interleaved

- Best for sequential traffic
- Best for load balancing between memory banks

Burst-interleaving granularity determined by board vendor



Manually Partitioned Global Memory

Turn off interleaving and assign data manually into memory banks

- Control memory bandwidth across a group of buffers

Advantages

- Better performance when accessing multiple pointers assigned to multiple banks
- Leads to more deterministic behavior
  - Designer knows the access pattern

In majority of use cases, manually partition of global memory leads to improved performance

Manual Partitioning Mechanism

`aoc --sw-dimm-partition <kernel file>.cl`

- Configure memory banks as non-interleaved address spaces

Use CL_MEM_BANK flags to allocate memory buffer to one of the banks

- Allocate each buffer to a single memory bank only

CL_MEM_BANK_1_ALTERA	Allocates to lowest available memory region
CL_MEM_BANK_2_ALTERA	Allocates to the second memory bank
CL_MEM_BANK_n_ALTERA	Allocates to the n^{th} bank, as long as the board supports it

```
clCreateBuffer(context, CL_MEM_BANK_2_ALTERA |
          CL_MEM_READ_WRITE, size, 0, 0);
```


Avoiding False Memory Dependencies

Avoid using pointers that alias other pointers

- Only one pointer variable is used to access the contents

Use the restrict keyword whenever possible

- Specify to the compiler no aliasing for a pointer
- Prevents AOC from creating memory dependencies between non-conflicting load and store operations
- May cause functional errors if used with pointers that aliases other pointers

```
__kernel void my_kernel ( __global int * restrict A,
                          __global int * restrict B)
{
                          …
}
```

Heterogeneous Memory

Some boards offer more than one type of global memory

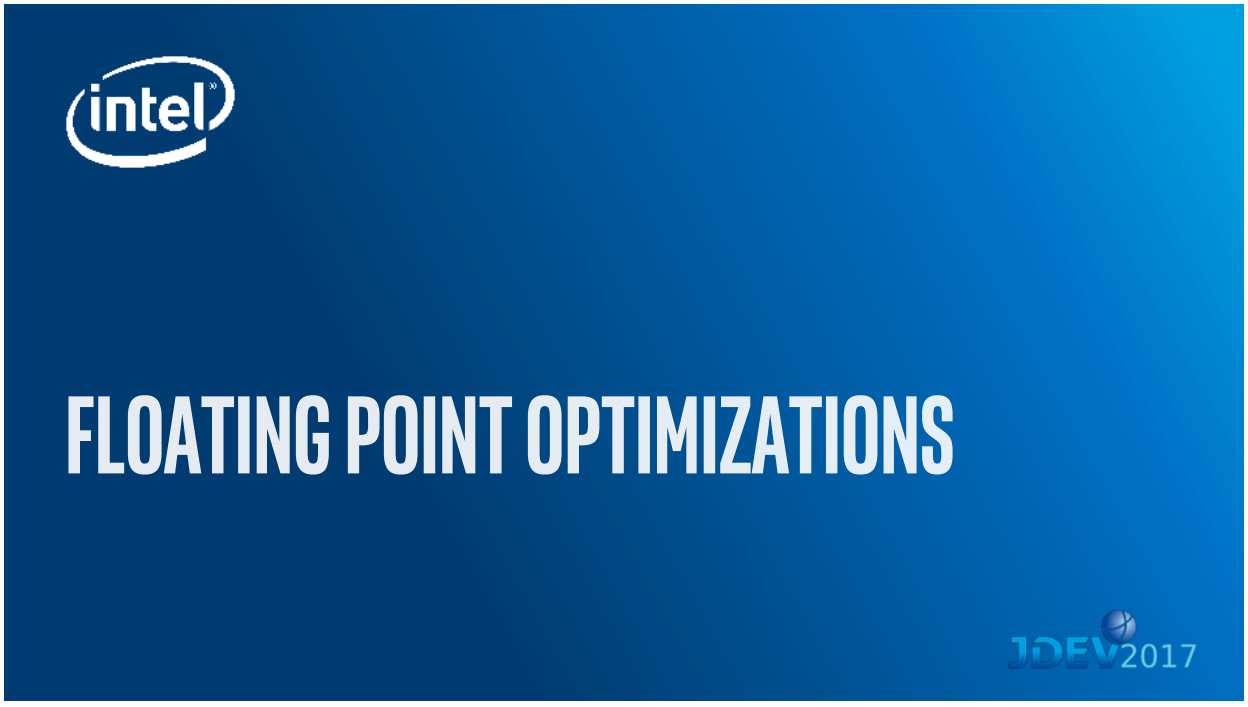
- Eg. DDR and QDR
- One memory will be used by default

Memory location can be assigned per kernel argument

- `__attribute((buffer_location("MEMORY_NAME")))`

Host will move memory to correct memory type upon kernel invocation

```
__kernel void foobar (
                        __global uint *src,
                        __global uint *dst,
                        __global __attribute((buffer_location("QDR"))) char* g_tables)
{ …
```

Floating-Point Optimizations

Apply to `float` and `double` data types

AOC has the ability to create more efficient hardware for floating-point operations

Optimizations will cause small differences in floating-point results

- **Not** IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008) compliant

AOC floating-point optimizations

- Needs to be manually enabled
- Tree Balancing
- Reducing Rounding Operations

Other optimizations

- Floating-point vs. fixed-point representations
- Use a device with hard floating point

Arithmetic Order of Operation Rules

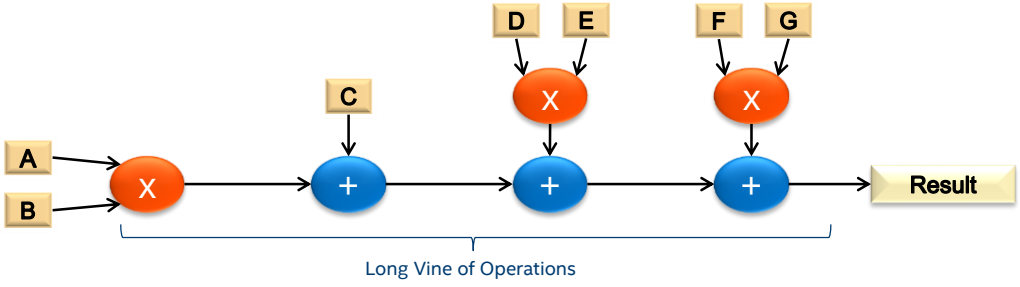
Strict order of operation rules apply in OpenCL

By default, AOC honors those rules

- May lead to long, unbalanced, slower, less-efficient floating-point operations

Example

```
Result = (((A * B) + C) + (D * E)) + (F * G)
```



Tree Balancing

Allow AOC to reorder arithmetic operations to convert into a tree pipeline structure

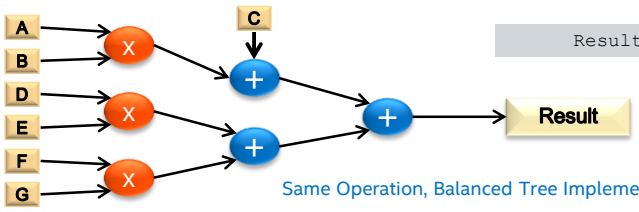
- Possibly affects the precision, not consistent with IEEE 754

Enable AOC tree balancing with `-fp-relaxed` option

- Design needs to tolerate the small differences in floating-point results

```
aoc  --fp-relaxed <kernel_file>.cl
```

```
Result = (((A * B) + C) + (D * E)) + (F * G)
```



Same Operation, Balanced Tree Implementation

Rounding Operations

For a series of floating-point operations, IEEE 754 require multiple rounding operation

Rounding can require significant amount of hardware resources

Fused floating-point operation

- Perform only one round at the end of the tree of the floating-point operations
- Leads to more accurate results
- Other processor architectures support certain fused instructions such as fused multiply and accumulate (FMAC)
- AOC can fuse any combination of floating-point operators

Reducing Rounding Operations

AOC will not reduce rounding operations by default

Enable AOC rounding reduction with `--fpc` option

- Not IEEE 754 compliant
- Use when program can tolerate these differences in floating-point results

```
aoc --fpc <kernel_file>.cl
```

1. Removes floating-point rounding operations whenever possible
   Round floating-point operation only once at the end of the tree of operations
   Applies to *, +, and -
2. Carry additional mantissa bits to maintain precision
   Carries additional bits through calculations, removes them at the end of the tree of operations
3. Changes rounding mode to round toward zero

Floating-Point vs. Fixed-Point Representation

Fixed-point implementation always use less logic compared to floating-point representation

- Save hardware by converting operations to fixed-point

No variable width fixed-point representation in OpenCL

- Use `char` (8-bit), `short` (16-bit), `int` (32-bit), or `long` (64-bit)

If data resolution required is not one of the default supported ones (8, 16, 32, or 64), use appropriate masking operations to save hardware

- Saving are relatively small but can be significant if applied across a large design

Fixed-Point Example

17-bit fixed-point data resolution needed

- Use 32-bit data type to store the value
- Avoid generating hardware for the upper 15 bits by using static bit masks
- Result: 17 bit addition implemented instead of 32-bit addition

32-bit data types

```
__kernel fixed_point_add(__global const unsigned int * restrict a,
                         __global const unsigned int * restrict b,
                         __global unsigned int * restrict result)
{
   size_t gid = get_global_id(0);
   unsigned int temp;
   result[gid] = 0x3_FFFF & ((0x1_FFFF & a[gid]) + (0x1_FFFF & b[gid]));
}
```

Mask away upper 14 bits of results

Mask away upper 15 bits of inputs

Want to learn more?

Developer Zone

- OpenCL online demos
- OpenCL design examples
- Application Developer Partners

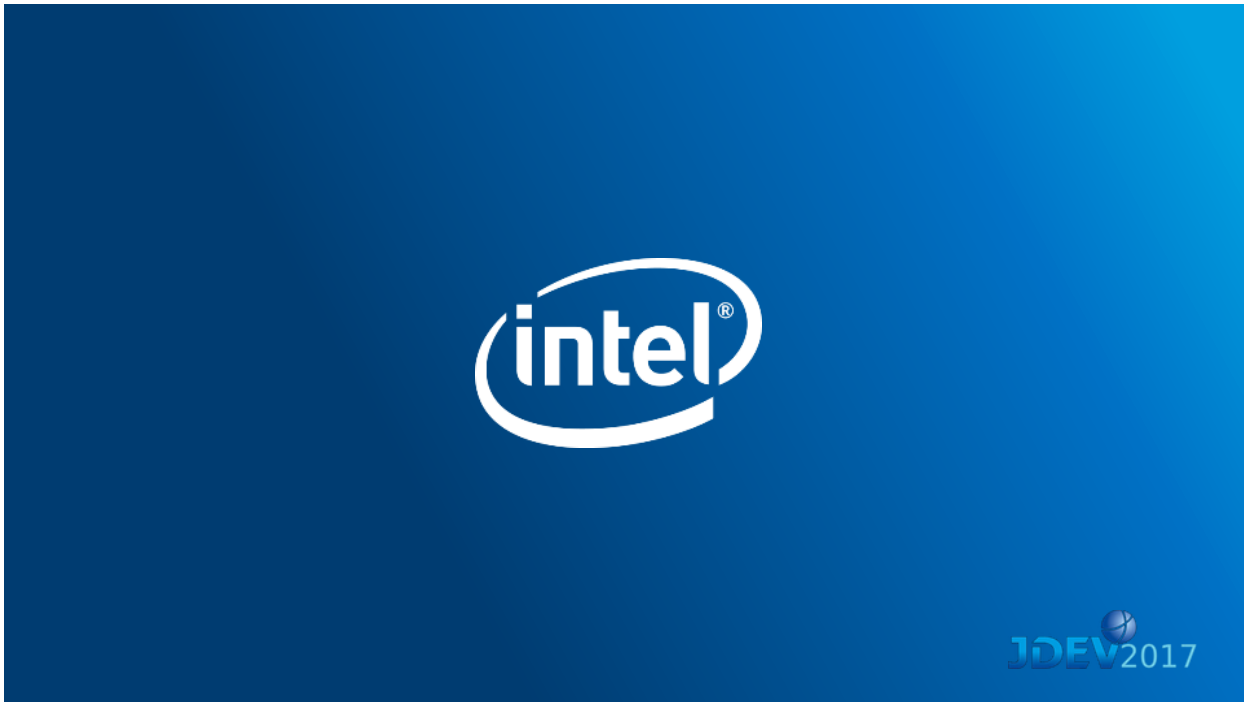
White papers, Publications and Optimization Tips for OpenCL

Instructor-Led training

- Parallel Computing with OpenCL Workshop by Altera
- Optimization of OpenCL for Altera FPGAs Training by Altera
- Building Custom Platforms

Online training

- Introduction to Parallel Computing with OpenCL
- Writing OpenCL Programs for Altera FPGAs
- Running OpenCL on Altera FPGAs
- FPGAs vs GPUs
- Single-Threaded vs. Multi-Threaded Kernels
- Building Custom Platforms for Altera SDK for OpenCL
- OpenCL Optimization Techniques: Secure Hash Algorithm (SHA-1) Example
- OpenCL Optimization Techniques: Image Processing Algorithm Example


intel
JDEV2017