17-20 July 2018
NiPS Summer School 2018
University of Perugia, Italy

Co-Funded by the H2020 Framework Programme of the European Union

Energy aware transprecision computing

FPGA programming using arbitrary precision data-types

OPRECOMP
Open Transprecision Computing

Dionysios Diamantopoulos
IBM Research - Zurich

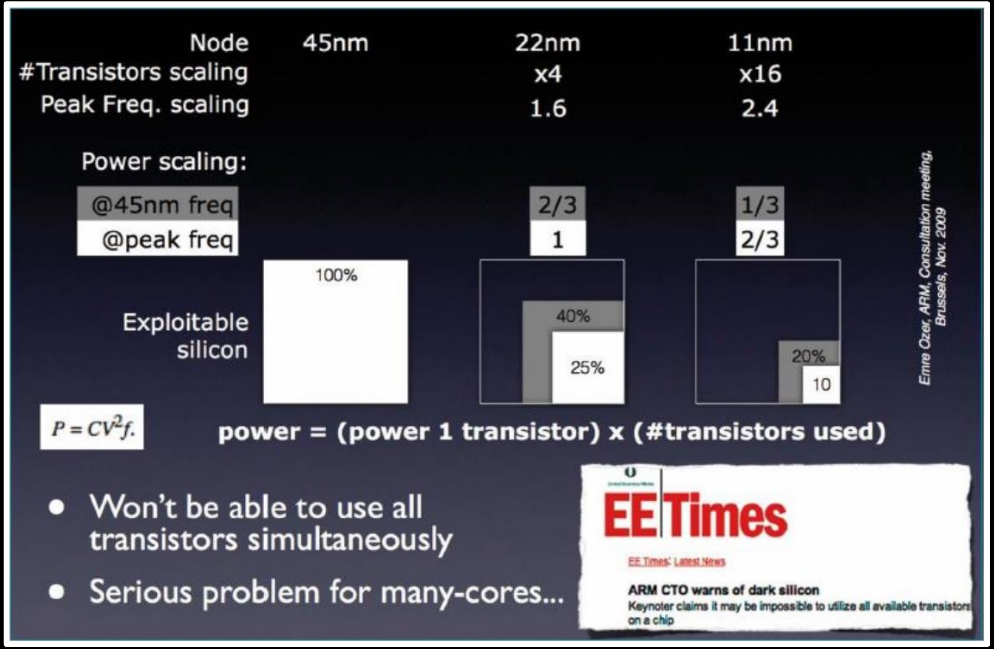
Disclaimer

The views and opinions expressed in this presentation are those of the author and do not necessarily reflect the official policy or position of IBM.

The “**Dark Silicon**” era

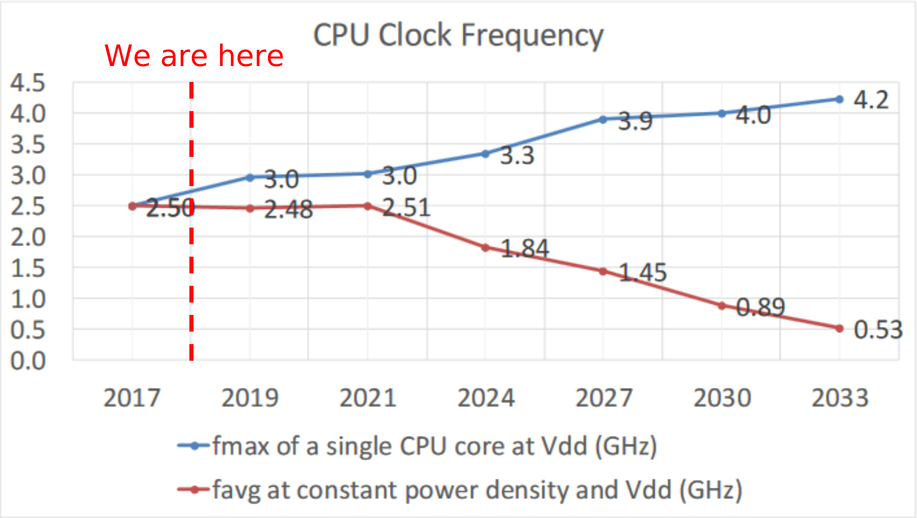
91%

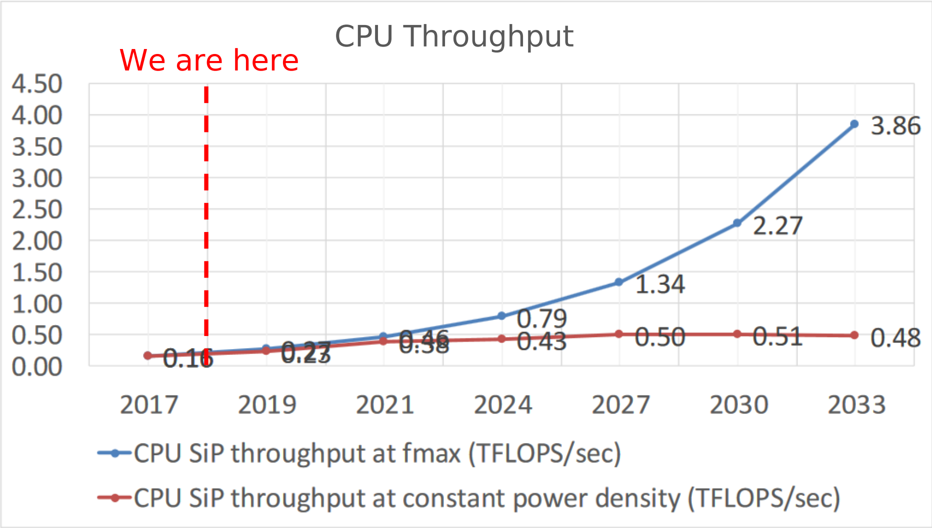
At 11nm, more than 91% of silicon area is “dark”.



A look into the next 15 years

-7.9x



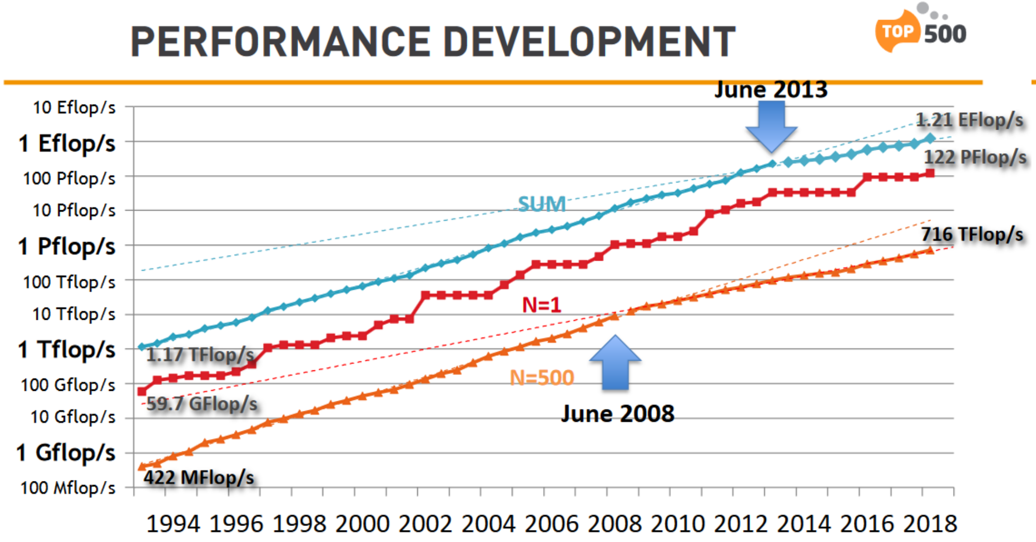


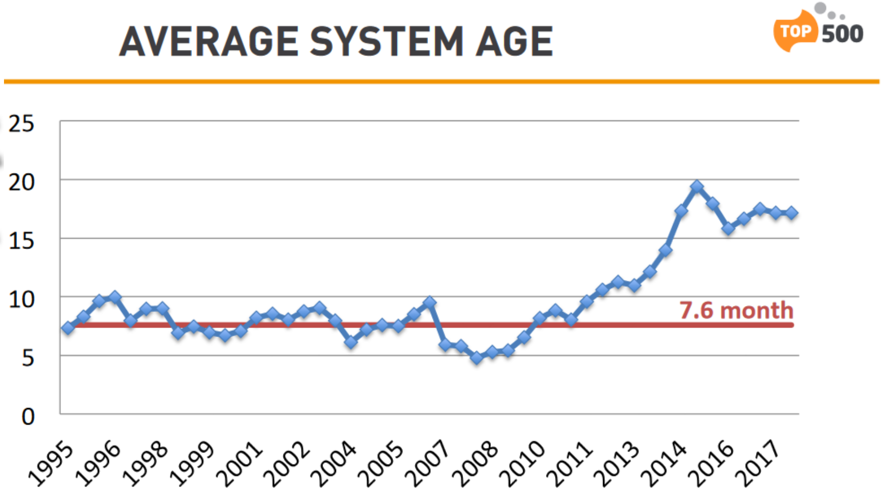
Source: The International Roadmap for Devices and Systems: 2017

Trends in HPC systems: Performance

Slow-down in performance growth since 2013 goes hand in hand with

- **Longer system usage (~2x) and**
- **Concentration of capabilities at the top (relatively larger top systems)**





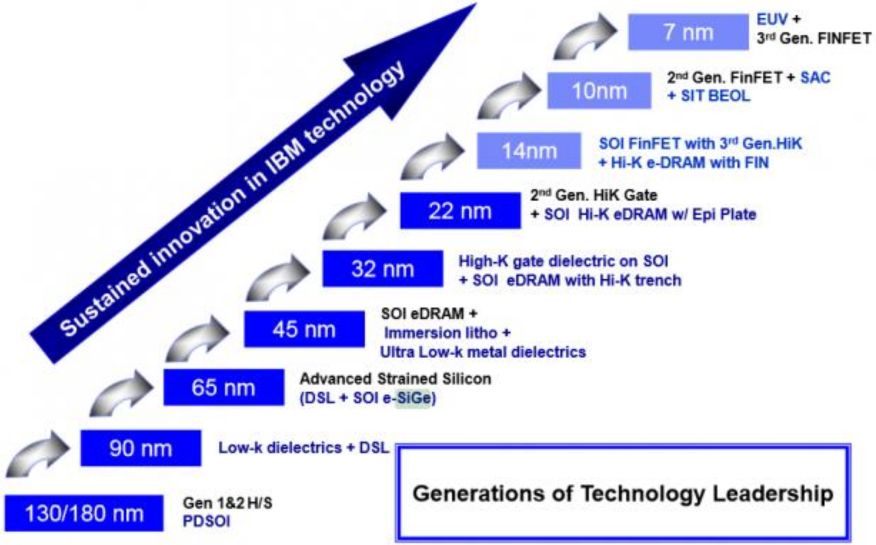
Source: top500.org, 2018

End of technology scaling

Ten years of IBM SOI Technology - Challenges:

- Leakage current
- EUV lithography
- Low yield
- High cost

- IBM's roadmap for 14nm, 10nm and 7nm

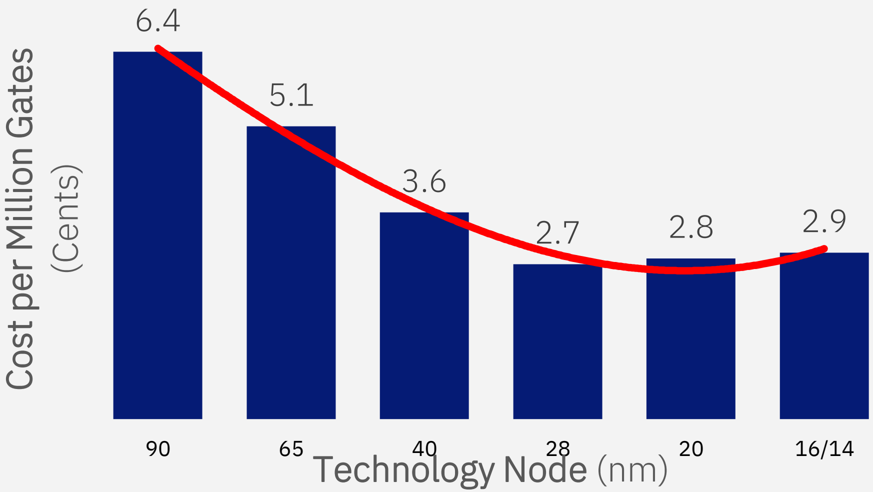


Cost per transistor rising – historic first

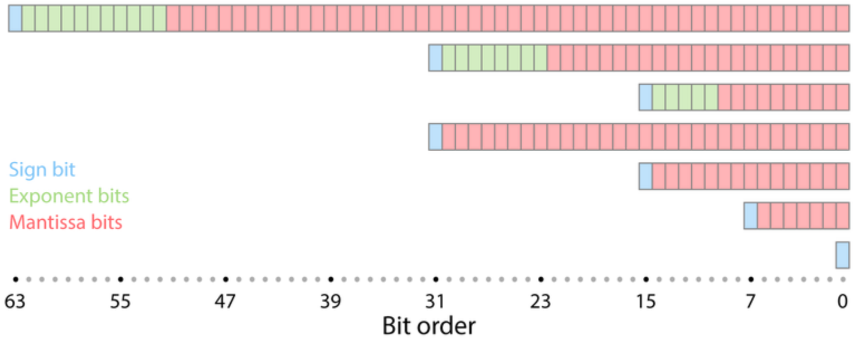
A "reasonably complex" SoC costs :

- $30 million at 28nm
- $271 million at 7nm
- $500 million at 5nm

- Gartner Research, semiengineering.com, 2016



Standard numerical formats and their hardware requirements



Format	Nonzero absolute range	Accuracy
float64	$2 \times 10^{-308} - 2 \times 10^{308}$	1×10^{-16}
float32	$1 \times 10^{-38} - 3 \times 10^{38}$	6×10^{-8}
float16	$6 \times 10^{-5} - 6 \times 10^{4}$	5×10^{-4}
int32	1 – 2,147,483,647	0.5
int16	1 – 32,767	0.5
int8	1 – 127	0.5
binary	1	1

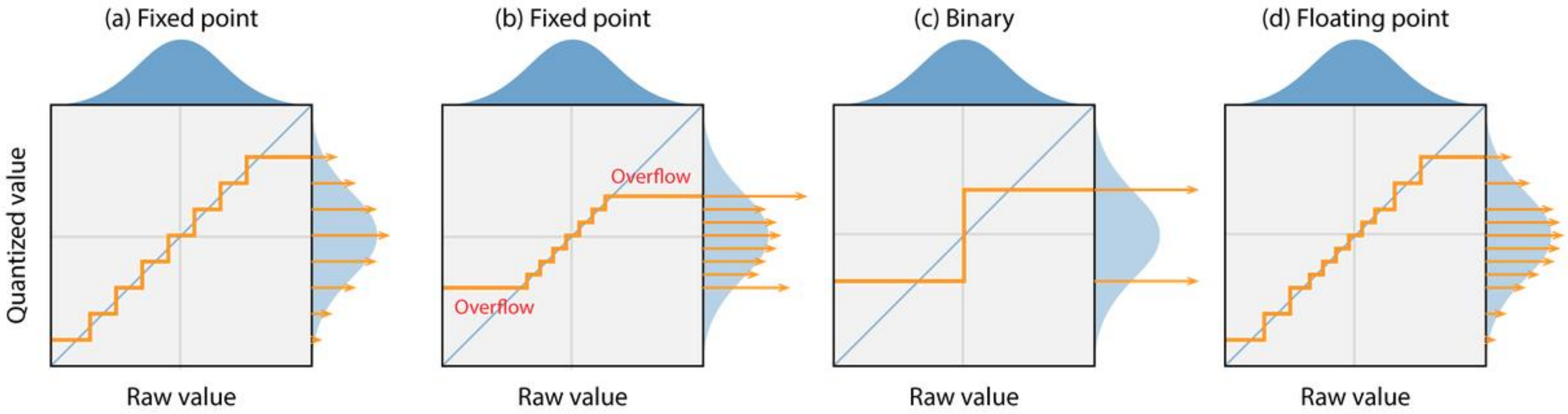
Operation	Energy (pJ)	Area (μm^2)
int8 addition	0.03	36
int16 addition	0.05	67
int32 addition	0.1	137
float16 addition	0.4	1,360
float32 addition	0.9	4,184
int8 multiplication	0.2	282
int32 multiplication	3.1	3,495
float16 multiplication	1.1	1,640
float32 multiplication	3.7	7,700

-18x *-27x*

@TSMC 45nm

Dr. Bill Dally's NIPS 2015 tutorial "High-performance hardware for machine learning"

Data types and quantization error

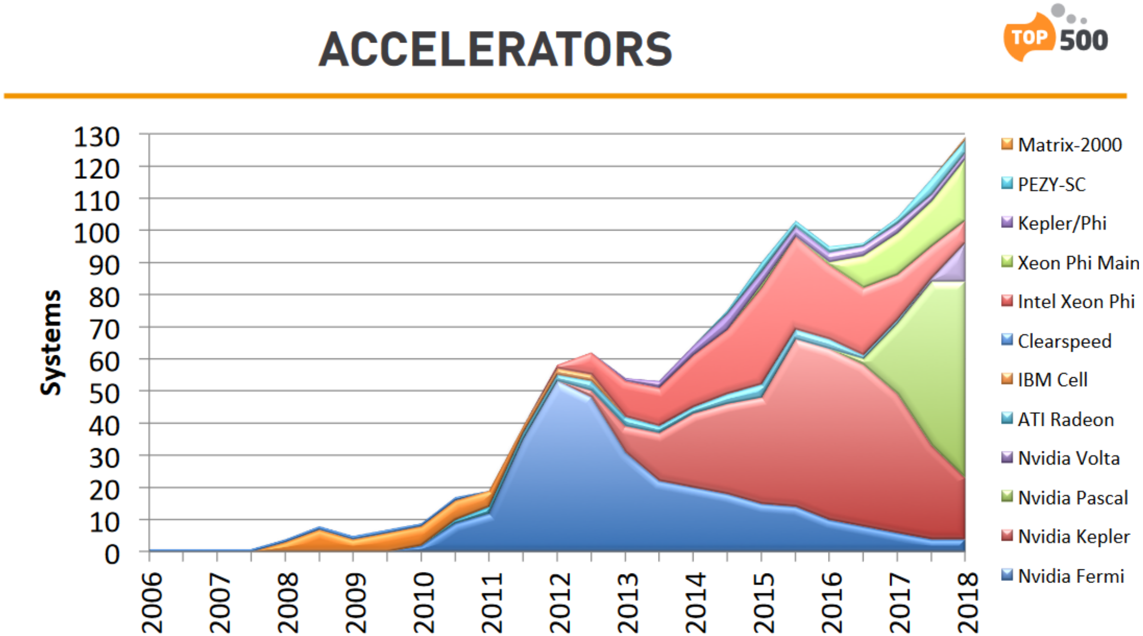


A hypothetical distribution of raw values (blue) and the corresponding discrete distributions resulting from quantization (orange)

https://ai.intel.com/flexpoint-numerical-innovation-underlying-intel-nervana-neural-network-processor/

Trends in HPC systems: Accelerators

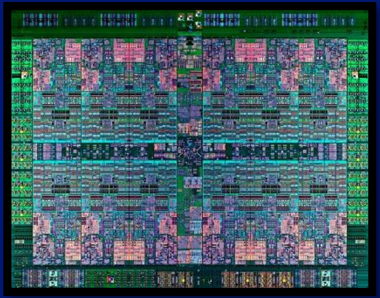
- **Accelerated systems get finally adopted by industrial users**
  - 25% of new TOP500 systems in November'17 + June'18
  - Accelerators can increase performance at lower TCO for targeted workloads

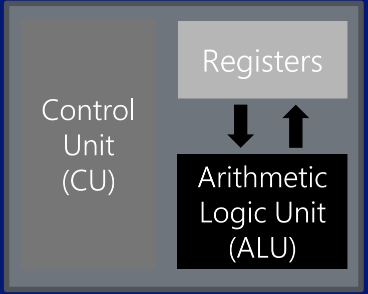


Source: top500.org, 2018

Silicon alternatives for accelerators

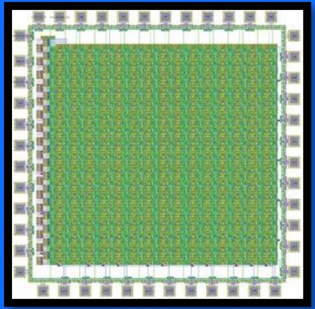
CPU

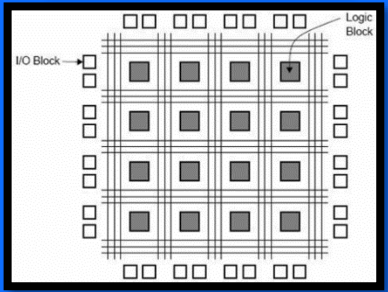




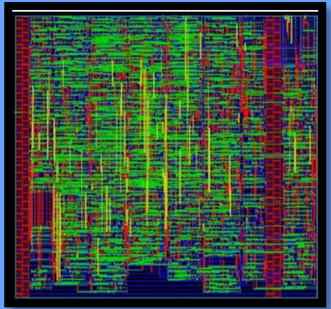
GPU

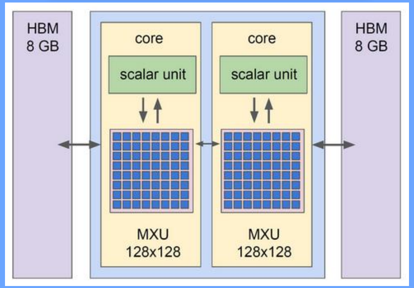
FPGA





ASIC





FLEXIBILITY ⟷ **EFFICIENCY**

Industry trends create new opportunities

Commercializing FPGAs from edge-to-cloud scale

datacenterknowledge.com, Apr 25, 2018


COMPANIES > MICROSOFT

Why Microsoft Has Bet on FPGAs to Infuse Its Cloud With AI

intel.com, April 11, 2018


ADOPTION OF INTEL FPGAS FOR ACCELERATION OF ENTERPRISE WORKLOADS GOES MAINSTREAM

datacenterdynamics.com, May 18, 2017


Google unveils second generation TPU, available as a service

18 May 2017 | By Sebastian Moss

eejournal.com, Sep. 7, 2017

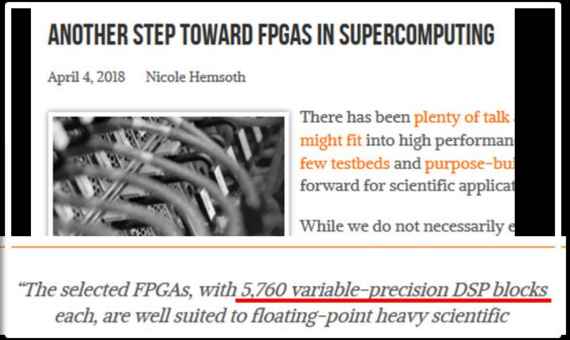

September 7, 2017

Xilinx Powers Huawei FPGA Accelerated Cloud Server

Delivers 10x acceleration for machine learning, data analytics and video processing

SHANGHAI, Sept. 6, 2017 /PRNewswire/

nextplatform.com, April 4, 2018


ANOTHER STEP TOWARD FPGAS IN SUPERCOMPUTING

April 4, 2018 Nicole Hemsoth

There has been plenty of talk ... might fit into high performan... few testbeds and purpose-bui... forward for scientific applicat...

While we do not necessarily e...

"The selected FPGAs, with 5,760 variable-precision DSP blocks each, are well suited to floating-point heavy scientific

venturebeat.com, March 26, 2018


Microsoft's Brainwave makes Bing's AI over 10 times faster

BLAIR HANLEY FRANK @BELRIL MARCH 26, 2018 9:00 AM

top500.org, August 23, 2017


Home / News / Microsoft Takes FPGA-Powered Deep Learning to the Next Level

Microsoft Takes FPGA-Powered Deep Learning to the Next Level

Michael Feldman | August 23, 2017 15:17 CEST

eetimes.com, March 19, 2018


designlines PROGRAMMABLE LOGIC

News & Analysis

New Xilinx CEO Touts 'Adaptive Computing'

Dylan McGrath
3/19/2018 06:01 AM EDT
4 comments

NO RATINGS
LOGIN TO RATE

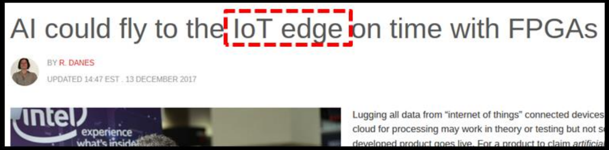
forbes.com, August 28, 2017


Microsoft: FPGA Wins Versus Google TPUs For AI

Moor Insights and Strategy, CONTRIBUTOR

Enterprise Mobile

siliconangle.com, December 13, 2017


AI could fly to the IoT edge on time with FPGAs

BY R. DANES
UPDATED 14:47 EST . 13 DECEMBER 2017

Lugging all data from "internet of things" connected devices ... cloud for processing may work in theory or testing but not s... developed product goes live. For a product to claim artificia...

nextplatform.com, August 22, 2017


AN EARLY LOOK AT BAIDU'S CUSTOM AI AND ANALYTICS PROCESSOR

August 22, 2017 Nicole Hemsoth

In the U.S. it is easy to focus on our native hy... (Google, Amazon, Facebook, etc.) and how the... deploy infrastructure at scale.

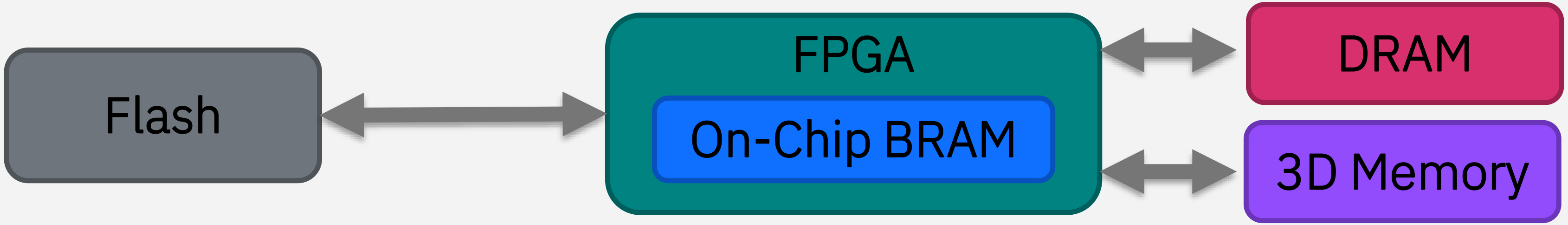
But as our regular readers understand well, t...

design-reuse.com, March 10, 2017


Alibaba Collaborating with Intel on an FPGA-based Solution to Help Customers Accelerate Business Applications

Memory Technology - Options for FPGAs

	BRAM	DRAM	3D Memory	Flash
Capacity	10MB	16 GB	4 GB	2 TB
Bandwidth	300 GB/s	20 GB/s	128 GB/s	2 GB/s
Access Latency	5 ns	40 ns	40 ns	50,000 ns
Power / MB	1000 mW	0.5 mW	0.2 mW	0.03 mW

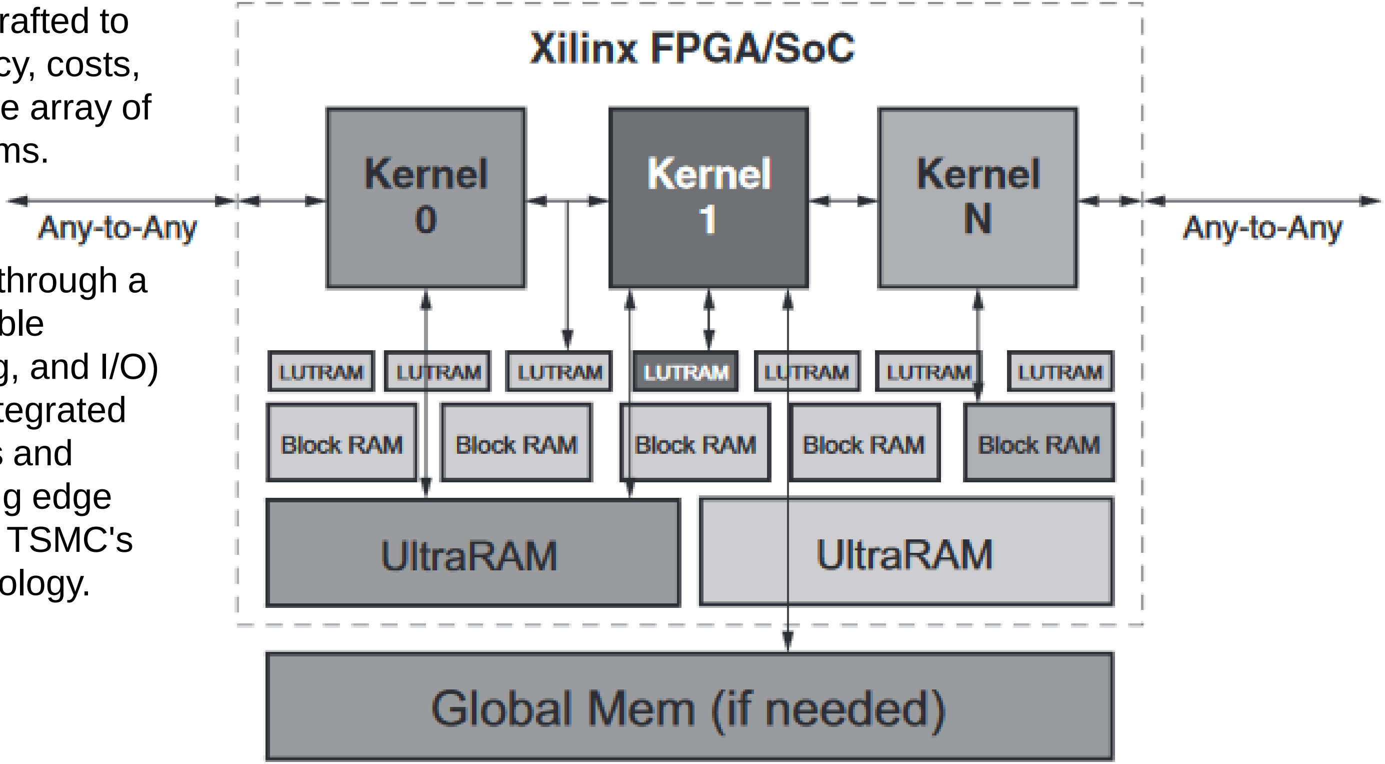


Source: The Era of Accelerators, V. Prasanna, FPL 2017

Xilinx FPGAs

Xilinx devices are carefully crafted to deliver the compute, efficiency, costs, and flexibility needs of a large array of high-performance end systems.

Xilinx achieves this balance through a mix of hardware programmable resources (e.g., logic, routing, and I/O) and flexible, independent, integrated core blocks (e.g., DSP slices and UltraRAM), all built on leading edge process technology, such as TSMC's 16nm FinFET process technology.



Xilinx All Programmable Devices: A Superior Platform for Compute-Intensive Systems, WP492 (v1.0.1) June 13, 2017

Xilinx FPGAs vs other acceleration platforms

Device	General Purpose Compute Efficiency	Tensor Operations Efficiency
NVidia Tesla P4	209 GOP/s/W[2] (INT8)	
NVidia Tesla P40	188 GOP/s/W (INT8)	
NVidia Tesla V100	72 GFLOP/s/W[3] (FP16)	288 GFLOP/s/W (FP16)
Intel Stratix 10	136 GOP/s/W (INT8)	
Xilinx Virtex® UltraScale+™	277 GOP/s/W (INT8) [Ref 27]	

Notes:

1. The numbers quoted are for comparison purposes only. Realizable device efficiency depends on the end application and the user.
2. Giga operations per second per watt of power consumed.
3. Giga floating point operations per second per watt of power consumed.

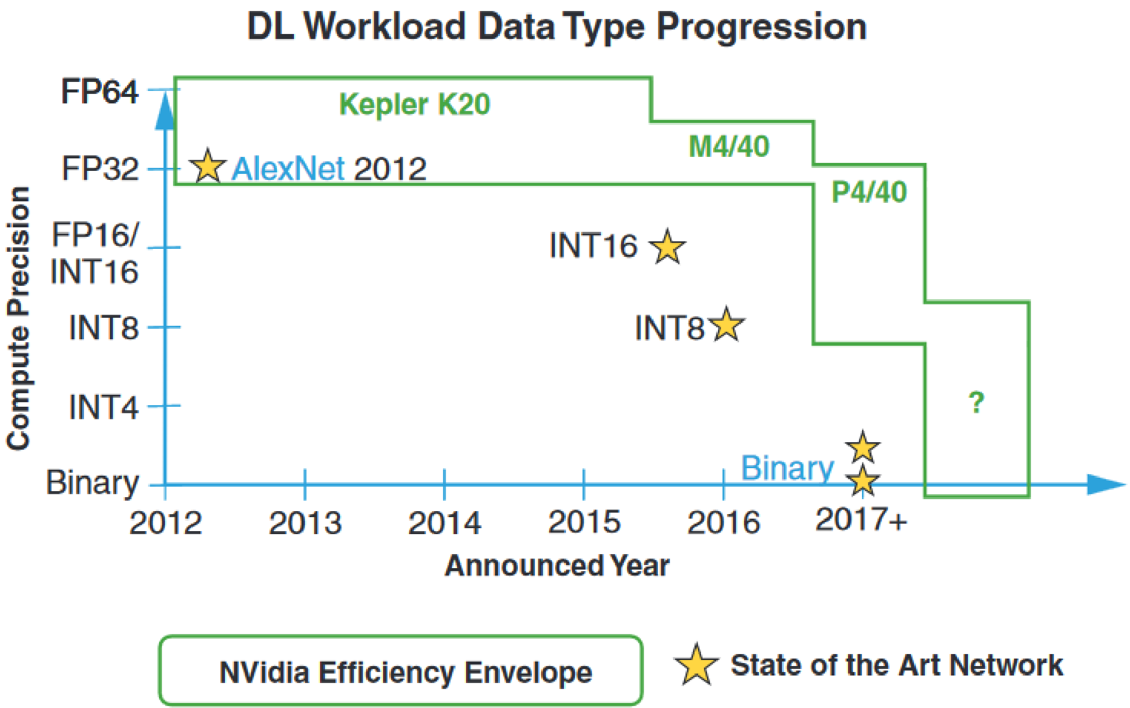
Xilinx devices offer the most efficient general-purpose compute platform from a raw compute perspective for fixed precision data types. This is primarily due to the lower overhead associated with processing in Xilinx FPGA-based architecture.

Xilinx All Programmable Devices: A Superior Platform for Compute-Intensive Systems, WP492 (v1.0.1) June 13, 2017

State-of-the-art DL and NVidia Reduced Precision Support

In an effort to keep pace with developments in the machine learning inference space, GPU vendors have been making the necessary silicon changes to support a limited set of reduced precision data types, e.g., FP16 and INT8. For example, the NVidia GPUs on Tesla P4 and P40 cards support INT8, providing four INT8 operations per ALU/Cuda core.

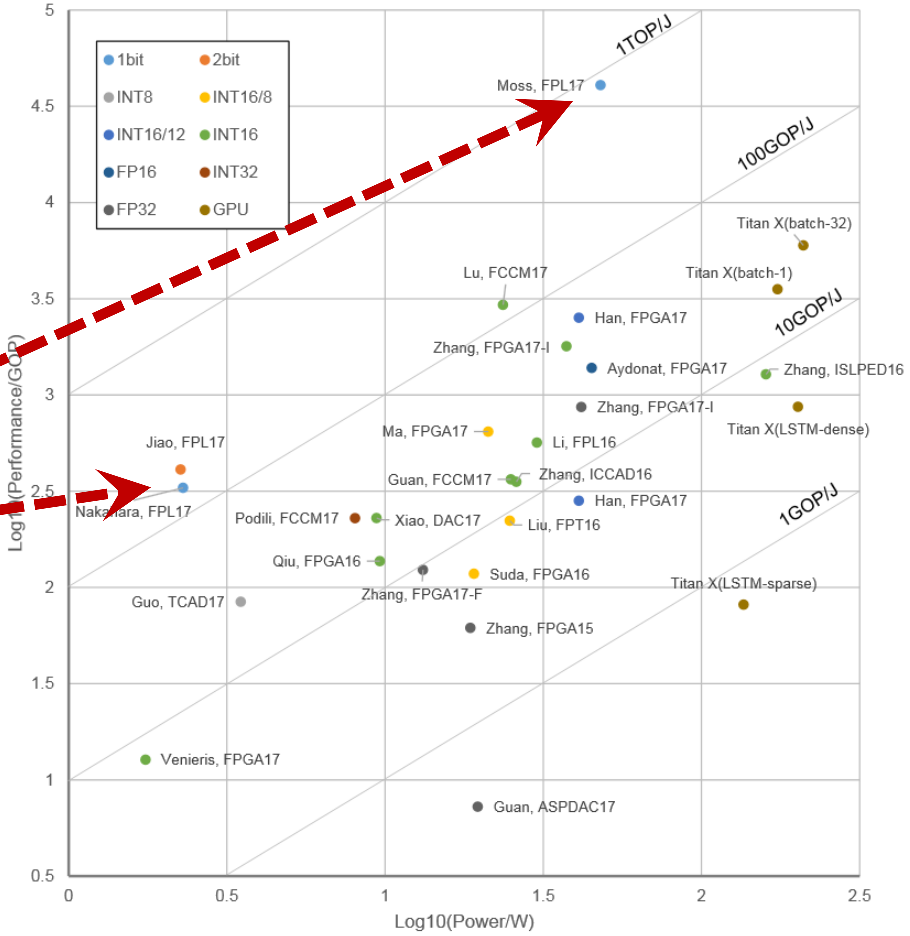
However, machine-learning inference benchmarks published by NVidia for GoogLeNet v1 inference on Tesla P40 show only a 3X improvement in efficiency for INT8 implementation vs. a FP32 implementation, illustrating the underlining challenges with squeezing reduced precision support into the GPU architecture and achieving efficient results



Xilinx All Programmable Devices: A Superior Platform for Compute-Intensive Systems, WP492 (v1.0.1) June 13, 2017

Precision tuning: mandatory for dealing with power & memory wall for modern NNs

- With software hardware co-design, FPGA is able to achieve **13× better energy efficiency** than state-of-the-art GPU while using **30% power** with conservative estimation.
- FPGA is a promising candidate for neural network acceleration.
- Acceleration **at bit-level** dominates on power and performance.
- Huge space for bit-width selection -> FPGAs can offer DSE & early prototyping for **arbitrary bit-widths**.



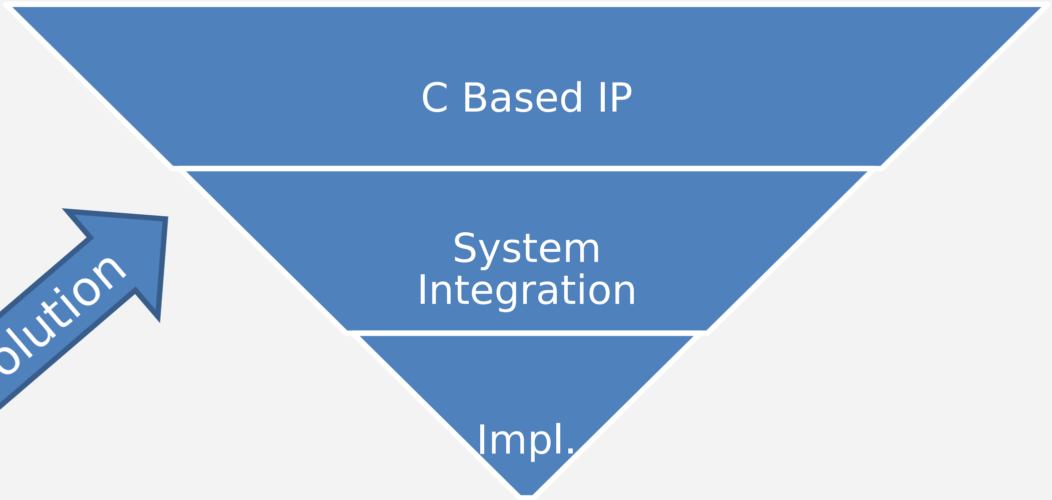
K. Guo, S. Zeng, J. Yu, Y. T. Wang, and H. Yang, "A survey of fpga based neural network accelerator," CoRR, vol. abs/1712.08934, 2017.

Challenges in using FPGA

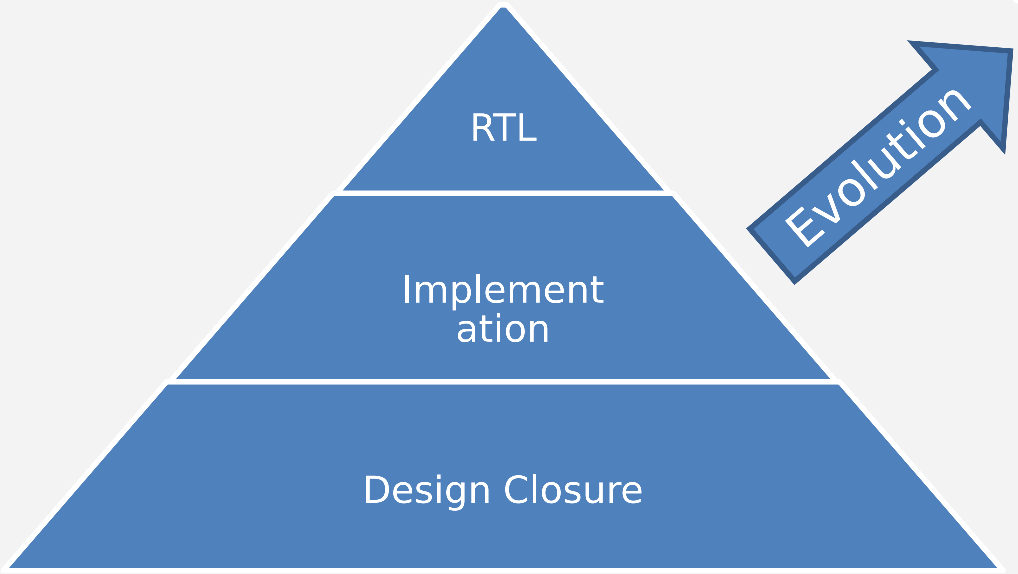
- Programming FPGAs
- Integrating FPGAs into applications
- Managing FPGAs in Cloud

Hardware Productivity – Time to switch !

High Level Synthesis Based Design



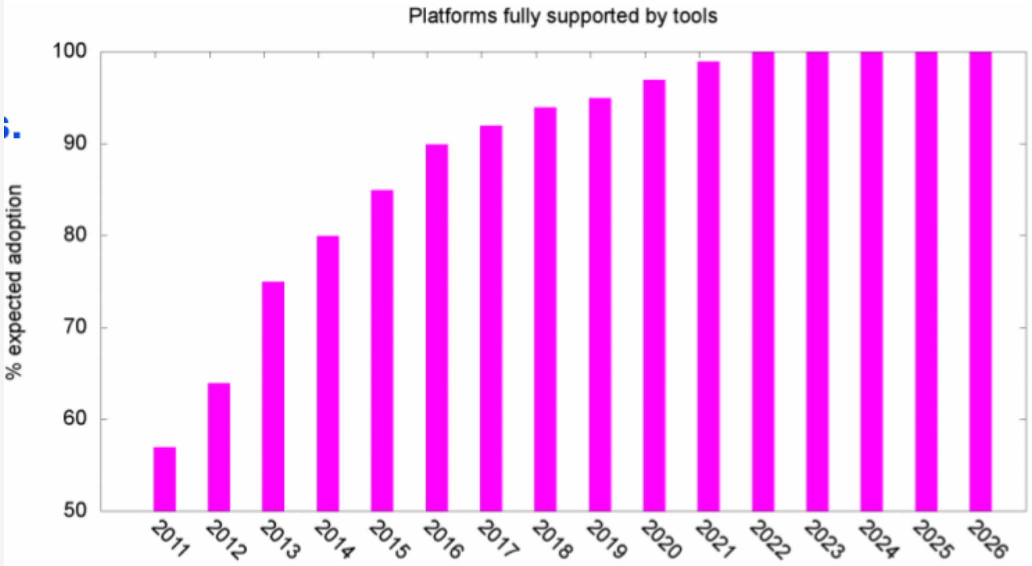
RTL Based Design



First Design	**15X Faster**
Derivative Design	>40X Faster
Typical QoR	0.7 – 1.2X

The 2020 Digital Platform … will be supported by ESL-HLS

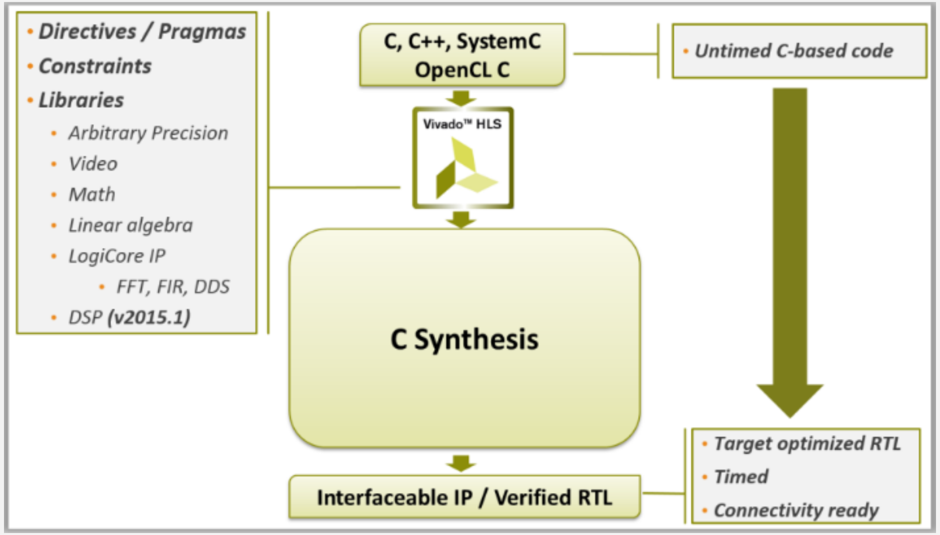
- Programmable and configurable aspects of the platform will be accessible via convenient layers of programmability. The current shift towards parallelism-aware languages including C/OpenMP, OpenCL, CUDA, AMP++, Matlab, SystemC and OpenACC is clearly visible and a vibrant reality among programmers.
- Within less than 10 years ALL computational platforms from the HPC realm to the autonomous, omnipresent, embedded systems will require full support by accessible ESL-HLS tools.



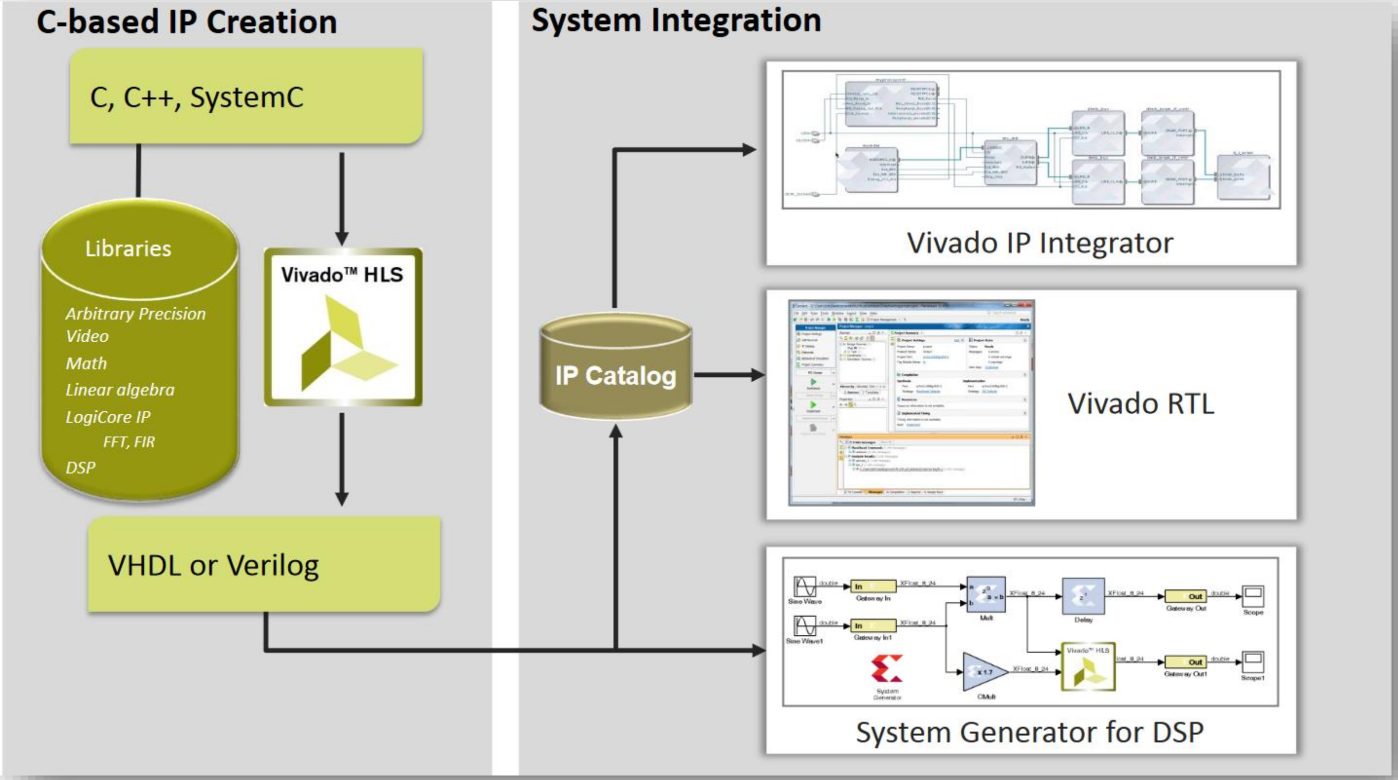
Source: ITRS

Vivado HLS: Framework for C-based IP Design

- C/C++ to optimized RTL IP
- C to hand-coded quality RTL–
  - In weeks not months…
- Accelerated verification
  - Over 100X over RTL
- Ideal for algorithmic designs
  - Excels at math (floating / fixed point)
  - Video, DSP…



Vivado HLS: System IP Integration Flow



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

Vivado HLS: Differentiations from RTL

- Code is untimed (C/C++)
- Loops are folded by default
- Pragmas play a crucial role in HLS for throughput
- Design control is added by HLS as a state machine
- RAMs are auto-generated based on arrays
- Simulation is tightly integrated
- Extensive architecture exploration
- Support for floating point

Arbitrary precision support in commercial HLS tools

Xilinx Vivado HLS

	Arbitrary Precision Integer	Arbitrary Precision Fixed-point	Floating-Point
C	[u]int<W> (1-1024 bits)	N/A	double, float, half
C++	ap_[u]int<W> (1-1024 bits)	ap_[u]fixed<W,I,Q,O,N> (1-1024 bits)	double, float, half
SystemC	sc_[u]int<W> (64 bits) sc_[u]bigint<W> (512 bits)	sc_[u]fixed<W,I,Q,O,N>	double, float, half

Intel HLS Compiler

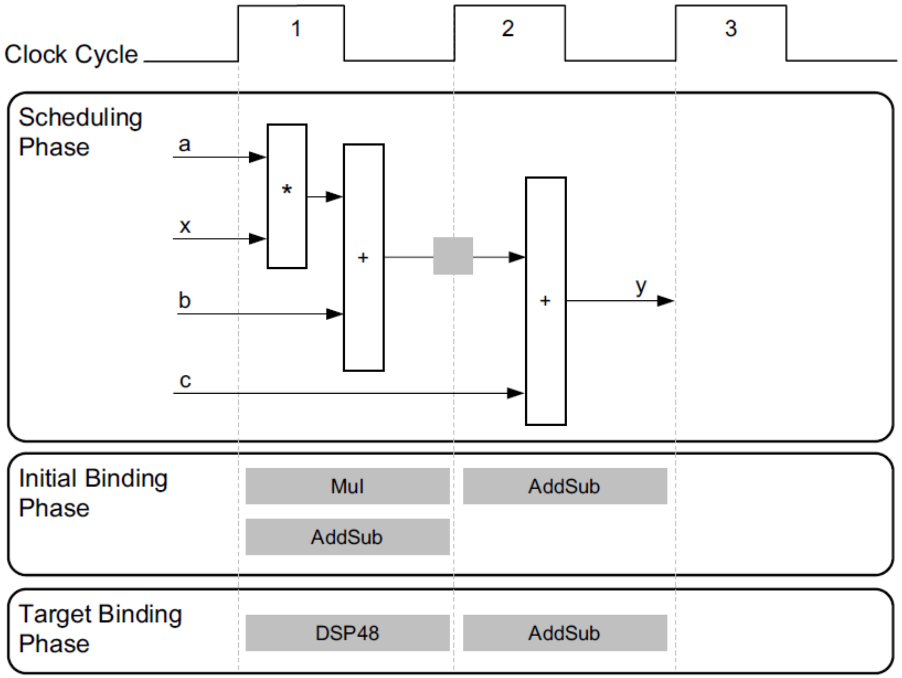
	Arbitrary Precision Integer	Arbitrary Precision Fixed-point	Floating-Point
C/C++	ac_int<N, true/false> (1-63 bits)	ac_fixed<N, I, true, Q, O>	double, float

High-Level-Synthesis

Scheduling & Binding Example

- In the scheduling phase of this example, HLS schedules the following operations to occur during each clock cycle:
  - First clock cycle: Multiplication and the first addition
  - Second clock cycle: Second addition and output generation
- In the initial binding phase of this example, HLS implements the multiplie operation using a combinational multiplier (Mul) and implements both add operations using a combinational adder/subtractor (AddSub).
- In the target binding phase, HLS implements both the multiplier and one of the addition operations using a DSP48 resource.

```
int foo(char x, char a, char b, char c) {
  char y;
  y = x*a+b+c;
  return y
}
```

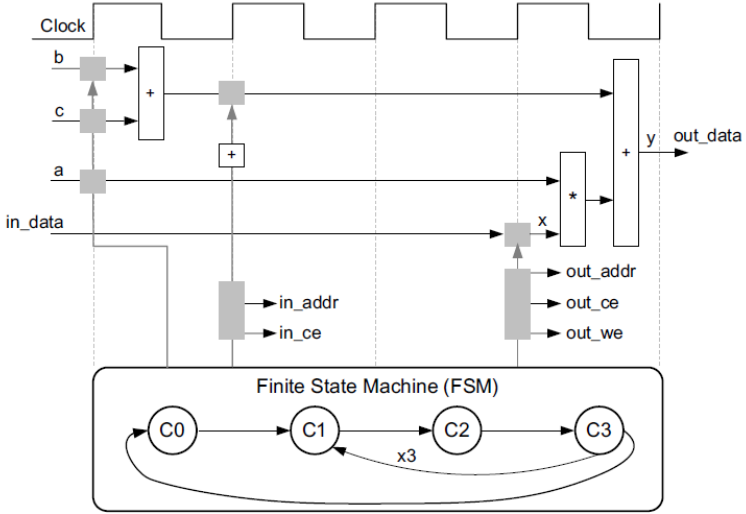


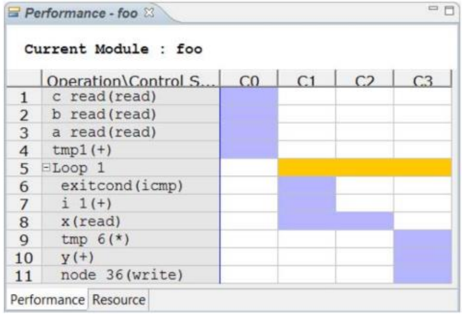
High-Level-Synthesis

Extracting Control Logic and Implementing I/O Ports

- HLS automatically extracts the control logic from the C code and creates an FSM in the RTL design to sequence these operations.
- HLS implements the top-level function arguments as ports in the final RTL design. The scalar variable of type char maps into a standard 8-bit data bus port.
- Arrays are synthesized into block RAM by default, but other options are possible, such as FIFOs, distributed RAM, and individual registers.
- HLS reads the data from port a with other values to perform the calculation and generates the first y output. The FSM ensures that the correct address and control signals are generated to store this value outside the block.

```
void foo(int in[3], char a, char b, char c, int out[3]) {
  int x,y;
  for(int i = 0; i < 3; i++) {
    x = in[i];
    y = a*x + b + c;
    out[i] = y;
  }
}
```





Advantages of Hardware Efficient Data Types

```
typedef char dinA_t;
typedef short dinB_t;
typedef int dinC_t;
typedef long long dinD_t;
typedef int dout1_t;
typedef unsigned int dout2_t;
typedef int32_t dout3_t;
typedef int64_t dout4_t;
```

```
#include "types.h"

void apint_arith(dinA_t  inA, dinB_t  inB, dinC_t  inC, dinD_t  inD,
          dout1_t *out1, dout2_t *out2, dout3_t *out3, dout4_t *out4
  ) {

  // Basic arithmetic operations
  *out1 = inA * inB;
  *out2 = inB + inA;
  *out3 = inC / inA;
  *out4 = inD % inA;

}
```

```
typedef int6 dinA_t;
typedef int12 dinB_t;
typedef int22 dinC_t;
typedef int33 dinD_t;
typedef int18 dout1_t;
typedef uint13 dout2_t;
typedef int22 dout3_t;
typedef int6 dout4_t;
```

C-based native data types
8-bit boundaries (8, 16, 32, 64 bits)

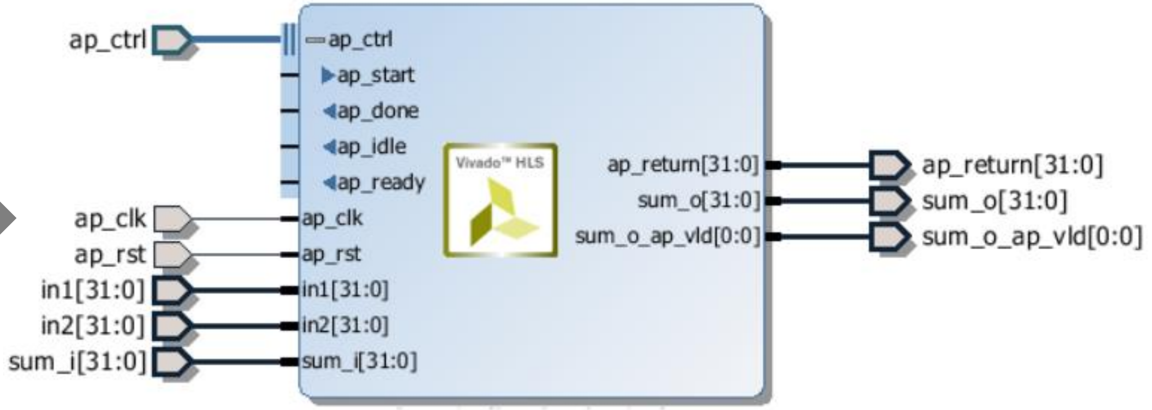
```
+ Latency (clock cycles):
    * Summary:
    +-----+-----+-----+-----+---------+
    |  Latency  |  Interval | Pipeline|
    | min | max | min | max |   Type  |
    +-----+-----+-----+-----+---------+
    |   66|   66|   67|   67|   none  |
    +-----+-----+-----+-----+---------+
* Summary:
+-----------------+---------+-------+--------+--------+
|       Name      | BRAM_18K| DSP48E|   FF   |  LUT   |
+-----------------+---------+-------+--------+--------+
|Expression       |        -|      -|       0|      17|
|FIFO             |        -|      -|       -|       -|
|Instance         |        -|      1|   17920|   17152|
|Memory           |        -|      -|       -|       -|
|Multiplexer      |        -|      -|       -|       -|
|Register         |        -|      -|       7|       -|
+-----------------+---------+-------+--------+--------+
|Total            |        0|      1|   17927|   17169|
+-----------------+---------+-------+--------+--------+
|Available        |      650|    600|  202800|  101400|
+-----------------+---------+-------+--------+--------+
|Utilization (%)  |        0|   ~0  |       8|      16|
+-----------------+---------+-------+--------+--------+
```

Arbitrary precision data types (1…1024 bits)

```
+ Latency (clock cycles):
    * Summary:
    +-----+-----+-----+-----+---------+
    |  Latency  |  Interval | Pipeline|
    | min | max | min | max |   Type  |
    +-----+-----+-----+-----+---------+
    |   35|   35|   36|   36|   none  |
    +-----+-----+-----+-----+---------+
* Summary:
+-----------------+---------+-------+--------+--------+
|       Name      | BRAM_18K| DSP48E|   FF   |  LUT   |
+-----------------+---------+-------+--------+--------+
|Expression       |        -|      -|       0|      13|
|FIFO             |        -|      -|       -|       -|
|Instance         |        -|      1|    4764|    4560|
|Memory           |        -|      -|       -|       -|
|Multiplexer      |        -|      -|       -|       -|
|Register         |        -|      -|       6|       -|
+-----------------+---------+-------+--------+--------+
|Total            |        0|      1|    4770|    4573|
+-----------------+---------+-------+--------+--------+
|Available        |      650|    600|  202800|  101400|
+-----------------+---------+-------+--------+--------+
|Utilization (%)  |        0|   ~0  |       2|       4|
+-----------------+---------+-------+--------+--------+
```

Interface synthesis

```
#include "sum_io.h"

dout_t sum_io(din_t in1, din_t in2, dio_t *sum) {

     dout_t temp;

     *sum = in1 + in2 + *sum;
     temp = in1 + in2;

     return  temp;
}
```



This example above includes:

•Two pass-by-value inputs in1 and in2.

•A pointer sum that is both read from and written to.

•A function return, the value of temp.

•Clock and Reset ports: ap_clk and ap_rst.

• Block-Level interface protocol. These are shown expanded in the preceding figure : ap_start, ap_done , ap_ready , and ap_idle .

• Port Level interface protocols. These are created for each argument in the top-level function and the function return (if the function returns a value). In this example, these ports are: in1 , in2 , sum_i , sum_o , sum_o_ap_vld , and ap_return.

Data Type and Interface Synthesis Support

The type of interfaces that are created by interface synthesis depend on the type of C argument, the default interface mode, and the INTERFACE optimization directive, using the following abbreviations:

- D: Default interface mode for each type.
- I: Input arguments, only read.
- O: Output arguments, only written.
- I/O: Input/Output arguments, both read and written.

Argument Type	Scalar		Array			Pointer or Reference			HLS:: Stream
Interface Mode	**Input**	**Return**	**I**	**I/O**	**O**	**I**	**I/O**	**O**	**I and O**
ap_ctrl_none	□	■	□	□	□	□	□	□	□
ap_ctrl_hs	□	■ D	□	□	□	□	□	□	□
ap_ctrl_chain	□	■	□	□	□	□	□	□	□
axis	■	□	■	□	■	■	□	■	■
s_axilite	■	■	■	■	■	■	■	■	□
m_axi	□	□	■	■	■	■	■	■	□
ap_none	■ D	□	□	□	□	■ D	■	■	□
ap_stable	■	□	□	□	□	■	□	□	□
ap_ack	■	□	□	□	□	■	■	■	□
ap_vld	■	□	□	□	□	■	■	■ D	□
ap_ovld	□	□	□	□	□	□	■ D	■	□
ap_hs	■	□	■	□	■	■	■	■	■
ap_memory	□	□	■ D	■ D	■ D	□	□	□	□
bram	□	□	■	■	■	□	□	□	□
ap_fifo	□	□	■	□	■	■	□	■	■ D
ap_bus	□	□	■	■	■	■	■	■	□

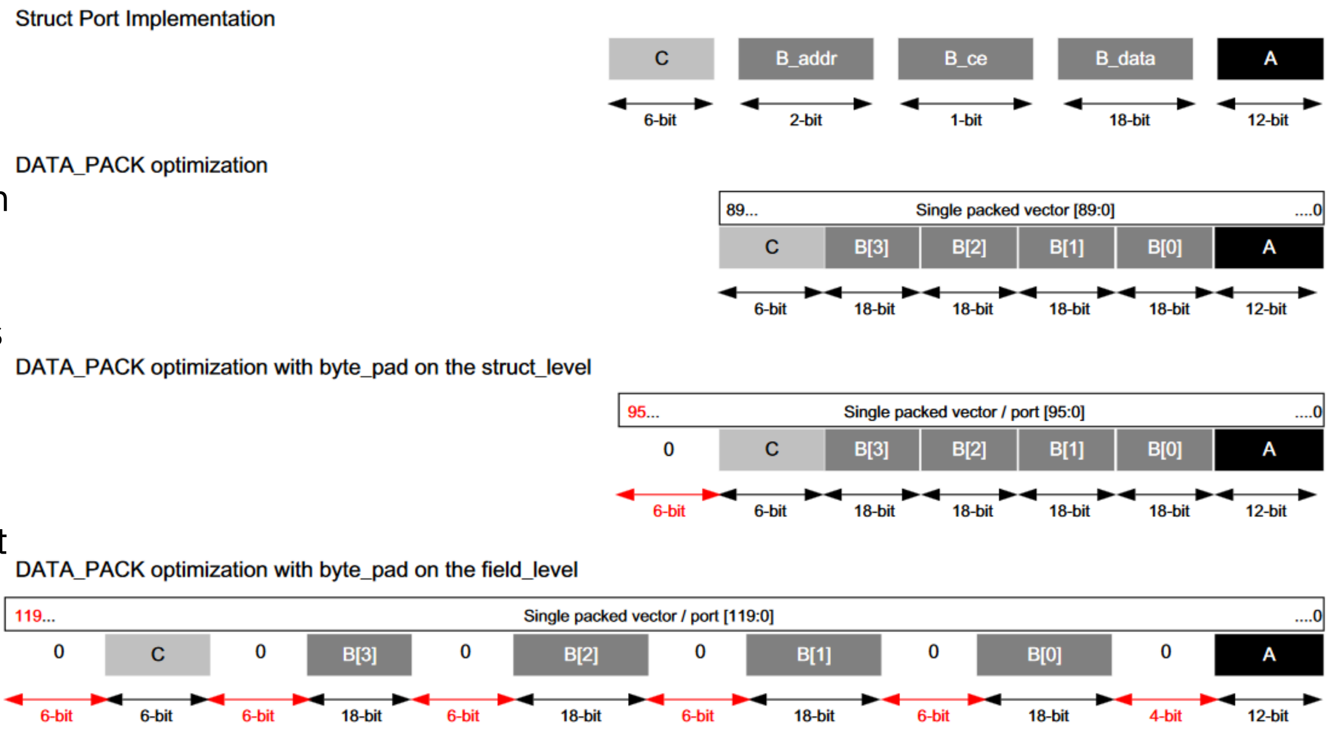
■ Supported D = Default Interface □ Not Supported

X14293

Interface Synthesis and Structs

```
typedef struct{
  int12 A;
  int18 B[4];
  int6 C;
} my_data;

void foo(my_data *a )
```

- The DATA_PACK optimization directive is used for packing all the elements of a struct into a single wide vector. This allows all members of the struct to be read and written to simultaneously.
- The first element of the struct is aligned on the LSB of the vector and the final element of the struct is aligned with the MSB of the vector.



AXI4-Stream Interfaces

An AXI4-Stream interface can be applied to any input argument and any array or pointer output argument.

AXI4-Stream interfaces are always implemented as registered interfaces to ensure no combinational feedback paths are created when multiple HLS IP blocks with AXI-Stream interfaces are integrated into a larger design.

Four types of register modes are provided to control how the AXI-Stream interface registers are implemented.

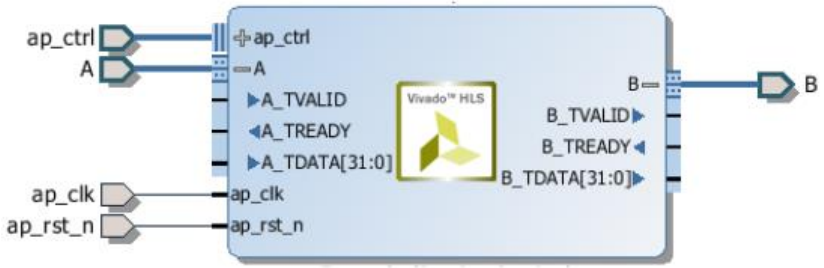
• **Forward**: Only the TDATA and TVALID signals are registered.

• **Reverse**: Only the TREADY signal is registered.

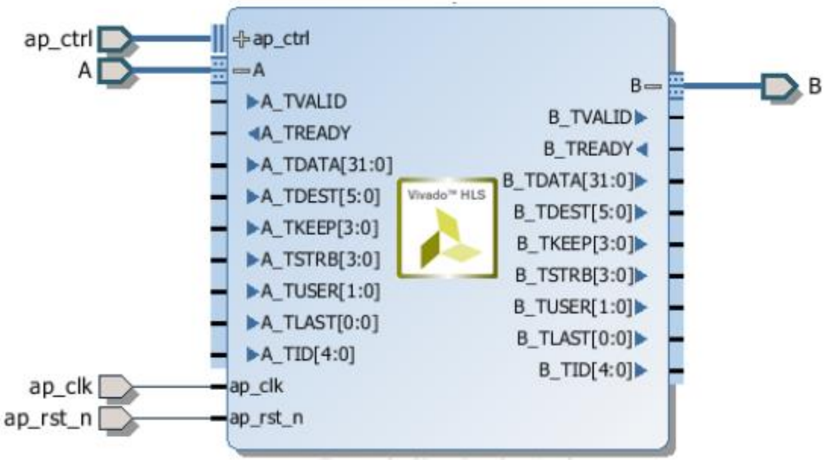
• **Both**: All signals (TDATA, TREADY and TVALID) are registered. This is the default.

• **Off**: None of the port signals are registered.

AXI4-Stream Interfaces Without Side-Channels



AXI4-Stream Interfaces With Side-Channels



AXI4-Lite Interface

You can use an AXI4-Lite interface to allow

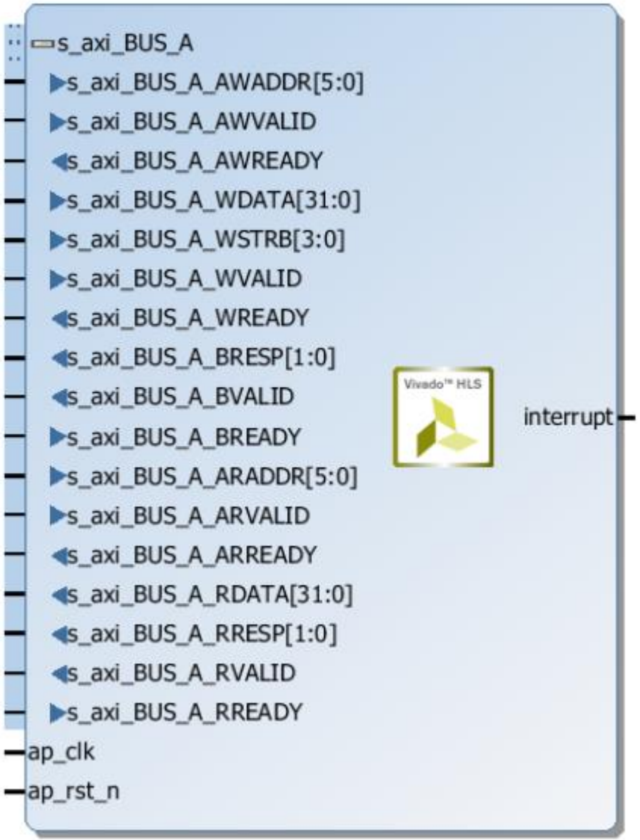
the design to be controlled by a CPU or microcontroller. Using the Vivado HLS AXI4-Lite interface, you can:

- Group multiple ports into the same AXI4-Lite interface.
- Output C driver files for use with the code running on an embedded processor.

The standard API implementation provide functions to perform the following operations.

- Initialize the device
- Control the device and query its status
- Read/write to the registers
- Set up, monitor, and control the interrupt

AXI4-Lite Slave Interfaces with Grouped RTL Ports



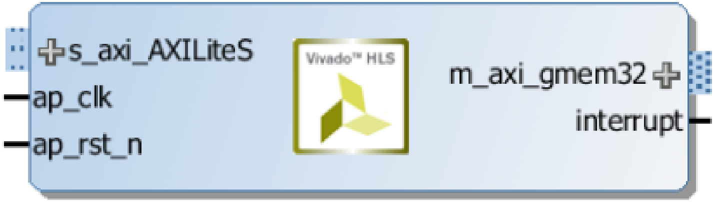
AXI4 Full Interface

You can use an AXI4 master interface on array or pointer/reference arguments, which Vivado HLS implements in one of the following modes:

- Individual data transfers
- Burst mode data transfers

With burst mode transfers, Vivado HLS reads or writes data using a single base address followed by multiple sequential data samples, which makes this mode capable of higher data throughput. Burst mode of operation is possible when you use the C memcpy function or a pipelined for loop.

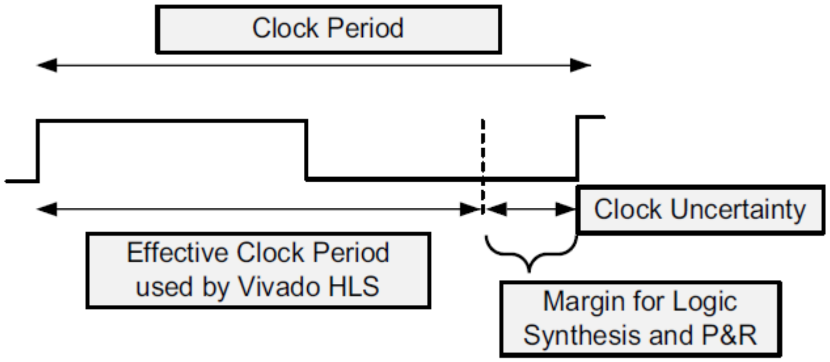
```
void example(volatile int *a){

#pragma HLS INTERFACE m_axi depth=50 port=a
#pragma HLS INTERFACE s_axilite port=return

//Port a is assigned to an AXI4 master interface

  int i;
  int buff[50];

//memcpy creates a burst access to memory
  memcpy(buff,(const int*)a,50*sizeof(int));

  for(i=0; i < 50; i++){
    buff[i] = buff[i] + 100;
  }

  memcpy((int *)a,buff,50*sizeof(int));
}
```



Design Optimization - Clock

Using the clock frequency and device target information Vivado HLS estimates the timing of operations in the design but it cannot know the final component placement and net routing:

- these operations are performed by logic synthesis of the output RTL. As such, Vivado HLS cannot know the exact delays.
- By default, the clock uncertainty is 12.5% of the cycle time. The value can be explicitly specified beside the clock period.
- Vivado HLS aims to satisfy all constraints: timing, throughput, latency.
- If a constraints cannot be satisfied, Vivado HLS always outputs an RTL design

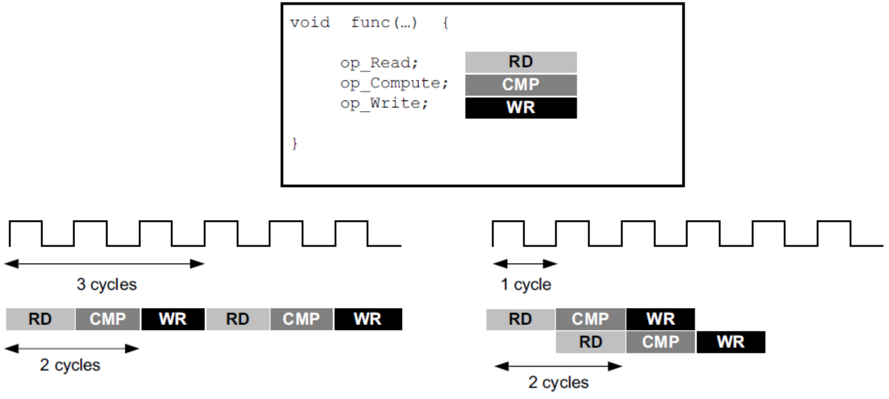


Design Optimization - Throughput

Pipelining allows operations to happen concurrently: each execution step does not have to complete all operations before it begin the next operation. Pipelining is applied to functions and loops.

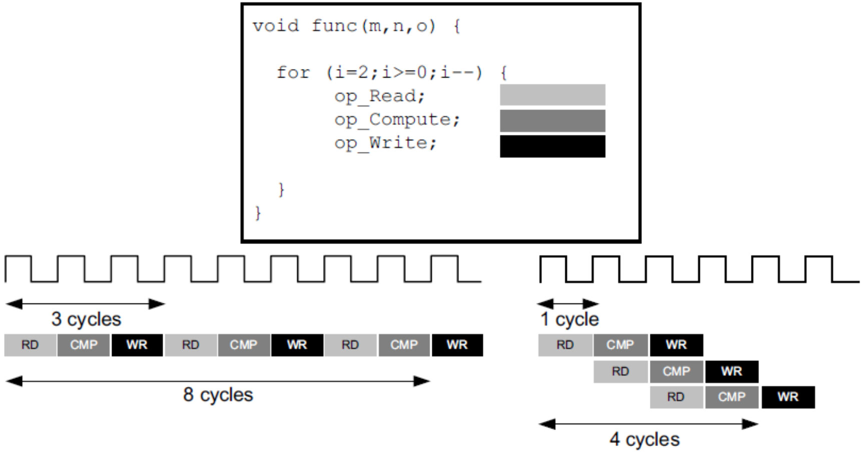
There is a difference in how pipelined functions and loops behave.

- In the case of functions, the pipeline runs forever and never ends.
- In the case of loops, the pipeline executes until all iterations of the loop are completed
- A pipelined function will continuously read new inputs and write new outputs. By contrast, because a loop must first finish all operations in the loop before starting the next loop, a pipelined loop causes a “bubble” in the data stream: a point when no new inputs are read as the loop completes the execution of the final iterations, and a point when no new outputs are written as the loop starts new loop iterations.



(A) Without Function Pipelining

(B) With Function Pipelining



(A) Without Loop Pipelining

(B) With Loop Pipelining

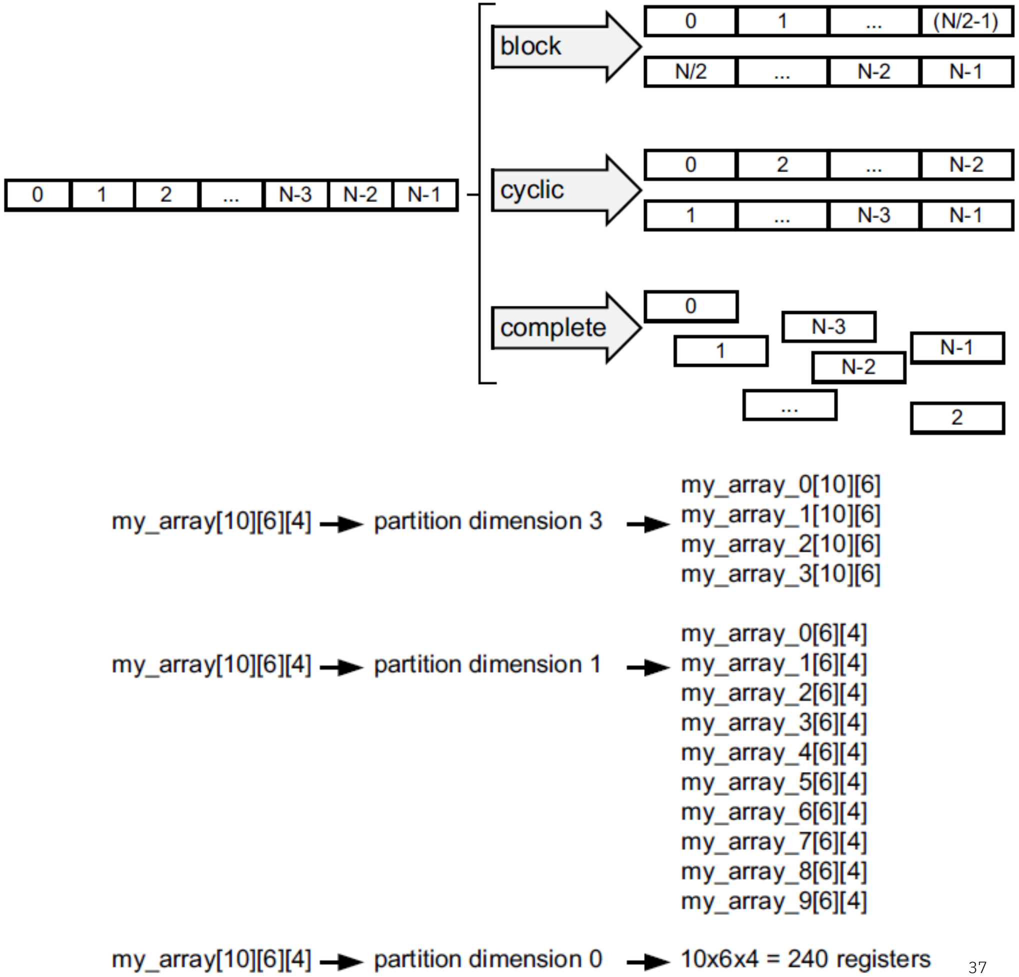
Design Optimization – Array Partitioning

Arrays are implemented as block RAM which only has a maximum of two data ports. This can limit the throughput of a read/write (or load/store) intensive algorithm. The bandwidth can be improved by splitting the array (a single block RAM resource) into multiple smaller arrays (multiple block RAMs), effectively increasing the number of ports. Arrays are partitioned using the ARRAY_PARTITION directive. Vivado HLS provides three types of array partitioning, as shown in the following figure:

• **block**: The original array is split into equally sized blocks of consecutive elements of the original array.

• **cyclic**: The original array is split into equally sized blocks interleaving the elements of the original array.

• **complete**: The default operation is to split the array into its individual elements. This corresponds to resolving a memory into registers.



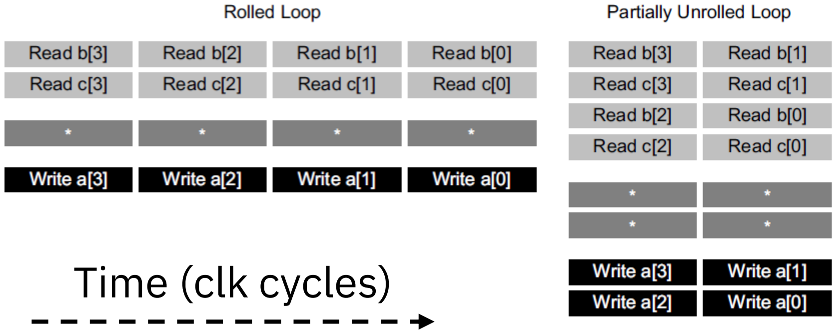
Throughput Optimization – Optimal Loop Unrolling to Improve Pipelining

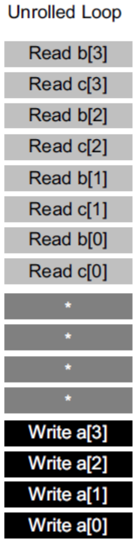
- By default loops are kept rolled in Vivado HLS: all operations in the loop are implemented using the same hardware resources or iteration of the loop.
- VHLS provides the ability to unroll/ partially for-loops using the UNROLL directive.

• **Rolled Loop**: each iteration is performed in a separate clock cycle. This implementation takes four clock cycles, only requires one multiplier and each block RAM can be a single-port block RAM.

• **Partially Unrolled Loop**: two multipliers and dual-port RAMs to support two reads or writes to each RAM in the same clock cycle. Only takes 2 clock cycles to complete: half the initiation interval and half the latency of the rolled loop version.

```
void  top(...) {

 ...
 for_mult:for (i=3;i>0;i--)   {
       a[i] = b[i] * c[i];
  }
 ...
}
```

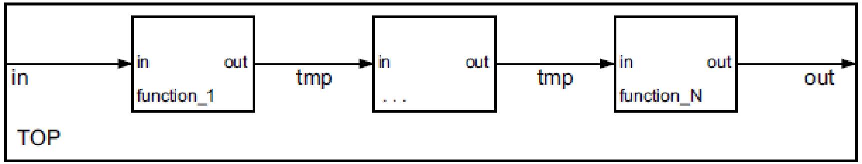




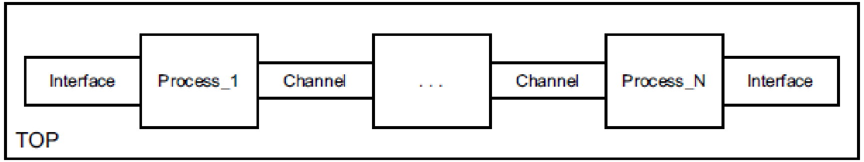
• **Unrolled loop**: all loop operation can be performed in a single clock cycle. This implementation however requires four multipliers. More importantly, this implementation requires the ability to perform 4 reads and 4 write operations in the same clock cycle. Because a block RAM only has a maximum of two ports, this implementation requires the arrays be partitioned.

Throughput Optimization – Task Level Parallelism

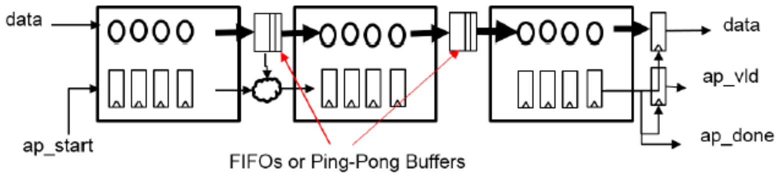
- DATAFLOW optimization creates a parallel process architecture and it is a powerful method for improving design throughput and latency.
- The channels between tasks can be simple FIFOs for scalar variables, or ping-pong buffers for non-scalar variables like arrays. Each of these channels also contain signals to indicate when the FIFO or the ping-pong buffer is full or empty.

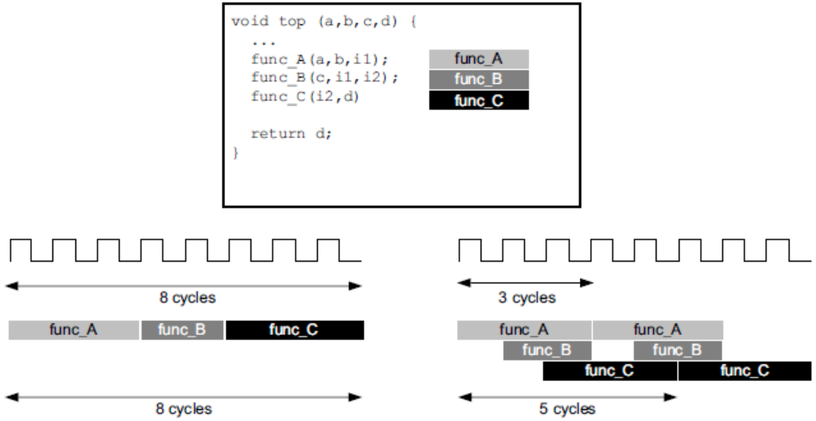


Sequential Functional Description



Parallel Process Architecture





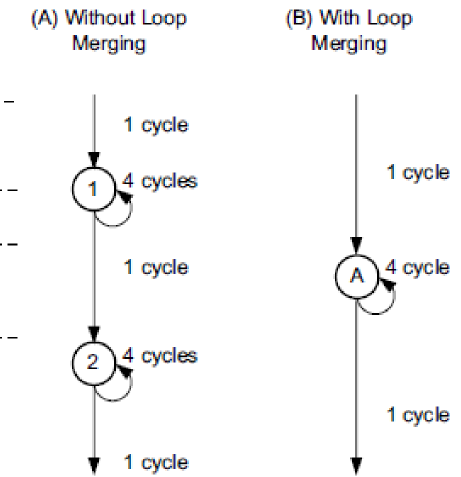
(A) Without Dataflow Pipelining

(B) With Dataflow Pipelining

Latency Optimization – Merging Sequential Loops

- All rolled loops imply and create at least one state in the design FSM. When there are multiple sequential loops it can create additional unnecessary clock cycles and prevent further optimizations.
- The LOOP_MERGE optimization directive is used to automatically merge loops
- Merging loops allows the logic within the loops to be optimized together. In the example, using a dual-port block RAM allows the add and subtraction operations to be performed in parallel.

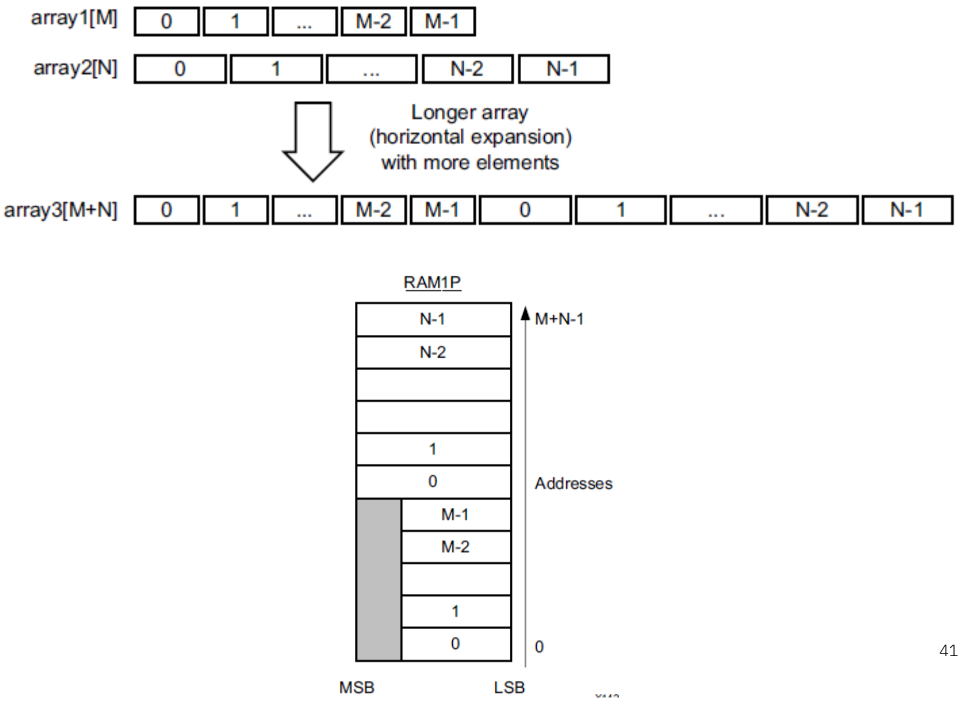
```
void top (a[4],b[4],c[4],d[4]...) {

  ...
  Add: for (i=3;i>=0;i--) {
    if  (d[i])
      a[i] = b[i] + c[i];
  }

  Sub: for (i=3;i>=0;i--) {
    if (!d[i])
     a[i] = b[i] - c[i];
  }
  ...
}
```



Area Optimization – Merging Arrays: Horizontal Mapping

- When there are many small arrays in the C Code, mapping them into a single larger array typically reduces the number of block RAM required.
- Each array is mapped into a block RAM or UltraRAM, when supported by the device. The basic block RAM unit provide in an FPGA is 18K. If many small arrays do not use the full 18K, a better use of the block RAM resources is map many of the small arrays into a larger array.
- **Horizontal mapping**: this corresponds to creating a new array by concatenating the original arrays. Physically, this gets implemented as a single array with more elements.

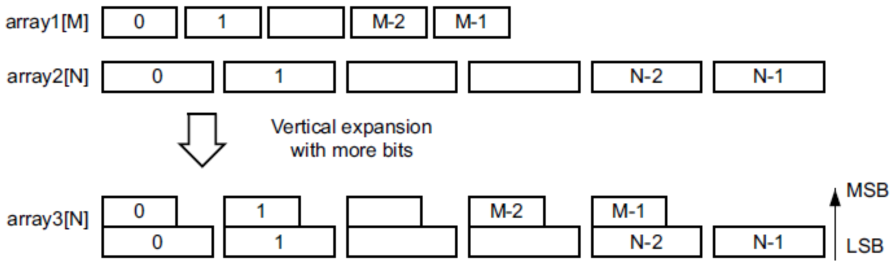
```
void foo (...) {
  int8  array1[M];
  int12 array2[N];
#pragma HLS ARRAY_MAP variable=array1 instance=array3 horizontal
#pragma HLS ARRAY_MAP variable=array2 instance=array3 horizontal
  ...
loop_1: for(i=0;i<M;i++) {
    array1[i] = ...;
    array2[i] = ...;
    ...
}
...
}
```

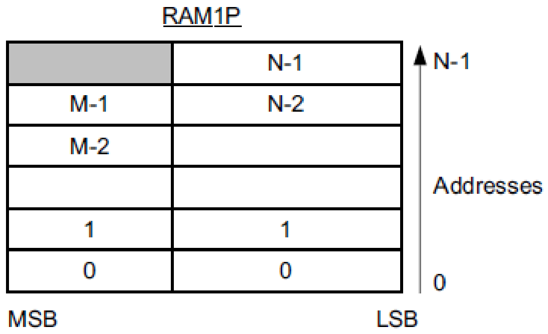


Area Optimization – Merging Arrays: Vertical Mapping

- When there are many small arrays in the C Code, mapping them into a single larger array typically reduces the number of block RAM required.
- Each array is mapped into a block RAM or UltraRAM, when supported by the device. The basic block RAM unit provide in an FPGA is 18K. If many small arrays do not use the full 18K, a better use of the block RAM resources is map many of the small arrays into a larger array.
- **Vertical mapping**: this corresponds to creating a new array by concatenating the original words in the array. Physically, this gets implemented by a single array with a larger bit-width.

```
void foo (...) {
  int8  array1[M];
  int12 array2[N];
#pragma HLS ARRAY_MAP variable=array2 instance=array3 vertical
#pragma HLS ARRAY_MAP variable=array1 instance=array3 vertical
  ...
loop_1: for(i=0;i<M;i++) {
    array1[i] = ...;
    array2[i] = ...;
    ...
}
...
}
```

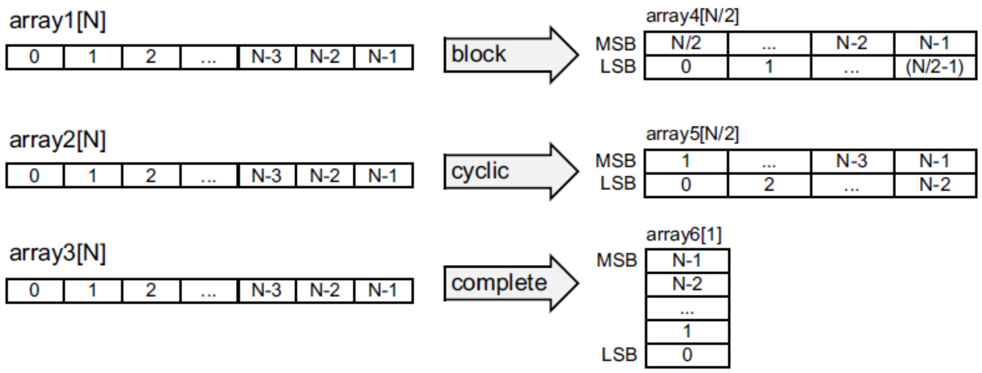




Area Optimization – Merging Arrays: Reshaping

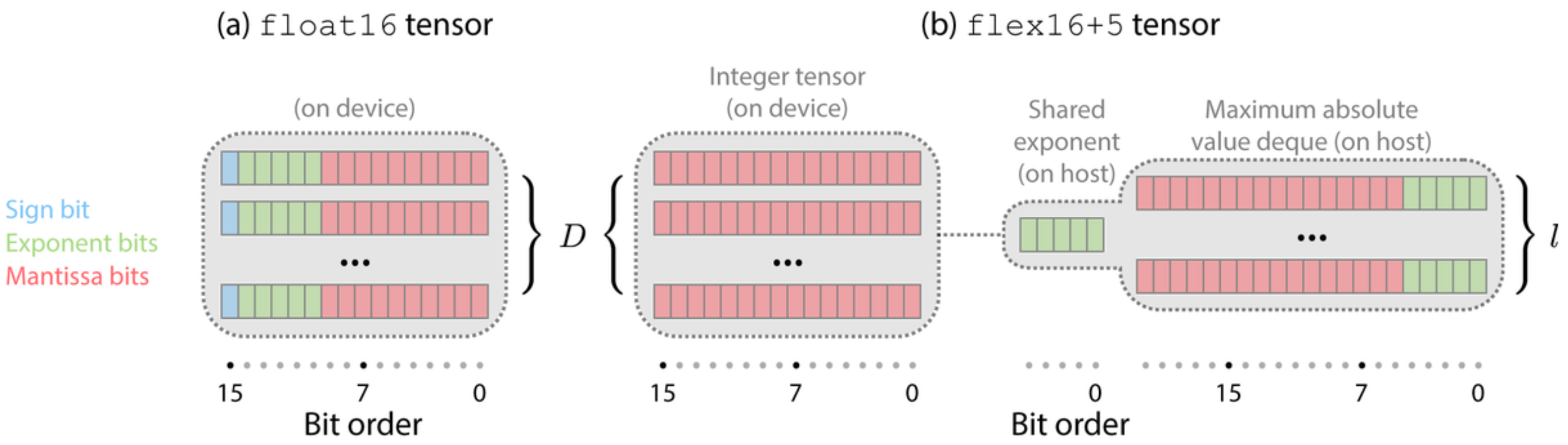
- The ARRAY_RESHAPE directive combines ARRAY_PARTITIONING with the vertical mode of ARRAY_MAP and is used to reduce the number of block RAM while still allowing the beneficial attributes of partitioning: parallel access to the data.
- The ARRAY_RESHAPE directive allows more data to be accessed in a single clock cycle. In cases where more data can be accessed in a single clock cycle, Vivado HLS may automatically unroll any loops consuming this data, if doing so will improve the throughput.
- The loop can be fully or partially unrolled to create enough hardware to consume the additional data in a single clock cycle.

```
void foo (...) {
int  array1[N];
int  array2[N];
int  array3[N];
#pragma HLS ARRAY_RESHAPE variable=array1 block factor=2 dim=1
#pragma HLS ARRAY_RESHAPE variable=array2 cycle factor=2 dim=1
#pragma HLS ARRAY_RESHAPE variable=array3 complete dim=1
...
}
```



Ideal for transprecision support !!!

Flexpoint format



Flexpoint is a tensorial numerical format based on an -bit integer tensor storing mantissas in two's complement form, and an -bit exponent, shared across all elements of the tensor. This format is denoted as ***flexN+M***

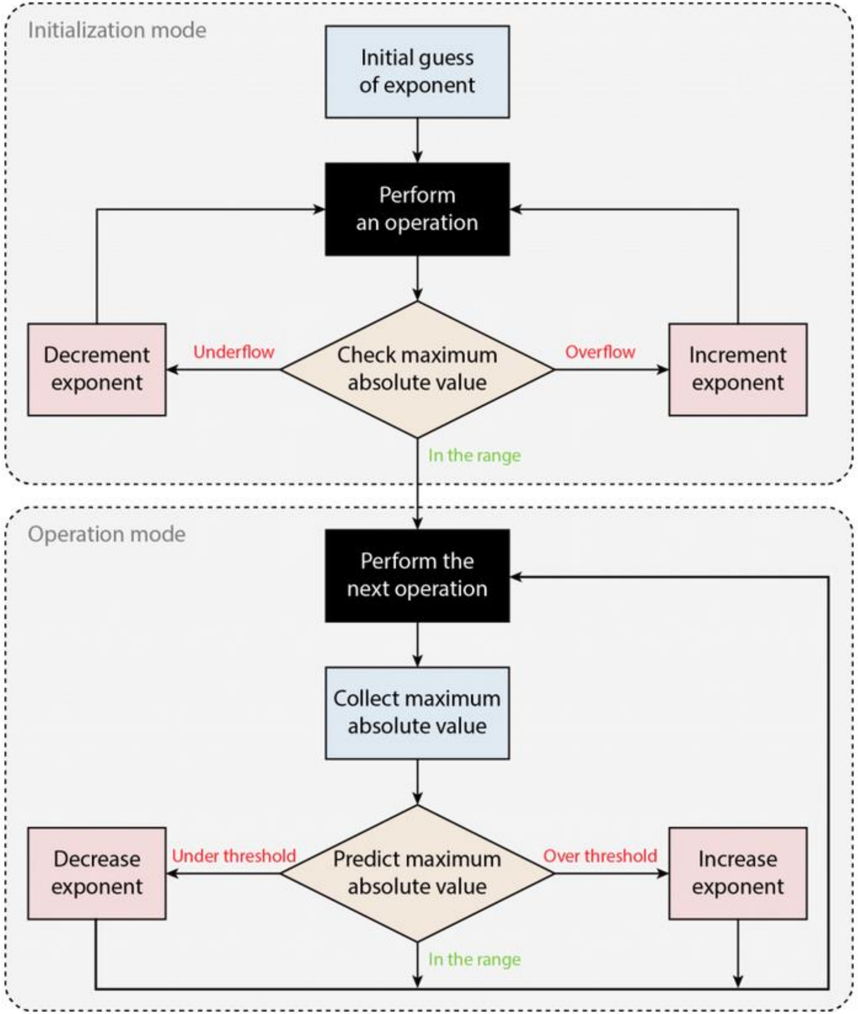
Flexpoint tensor is essentially a fixed point, not floating point, tensor. Even though there is a shared exponent, its storage and communication can be amortized over the entire tensor, a negligible overhead for huge tensors. Most of the memory on device is used to store tensor elements with higher precision that scales with the dimensionality of tensors

https://ai.intel.com/flexpoint-numerical-innovation-underlying-intel-nervana-neural-network-processor/

Flexpoint format

- Intel devised an exponent management algorithm called ***Autoflex,*** designed for iterative algorithms such as stochastic gradient descent where tensor operations, e.g. matrix multiplication and convolution, are performed repeatedly and outputs are stored

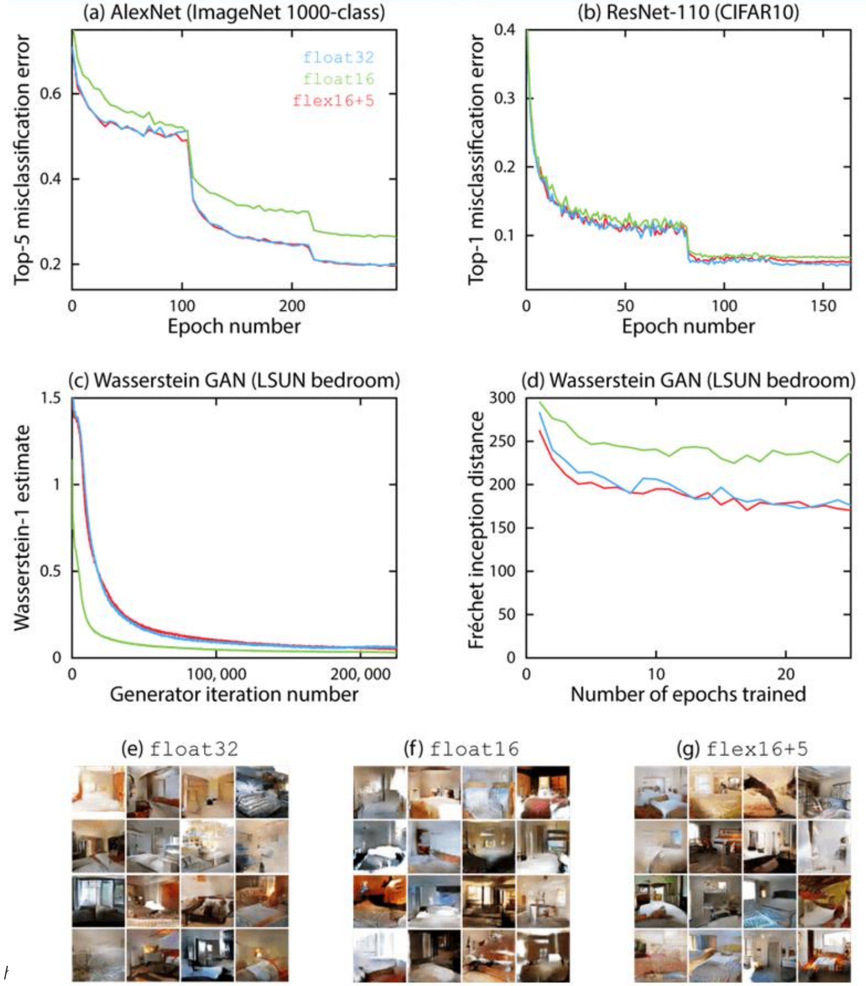
- In initialization mode, exponent of a tensor is iteratively adjusted, starting from an initial guess, until it is proper. During training, each operation on a tensor is wrapped around by an adaptive exponent management, which predicts the trend of the tensor's maximum absolute mantissa value based on statistics gathered from previous iterations in hardware buffers.



https://ai.intel.com/flexpoint-numerical-innovation-underlying-intel-nervana-neural-network-processor/

Flexpoint versus floating point.

- In all three experiments, flex16+5 achieved close numerical parity with float32, whereas significant performance gaps were observed in float16 for certain cases.
- For the two convolutional networks for image classification, misclassification errors were significantly higher in float16 than in float32 and flex16+5.
- In the case of Wasserstein GAN, float16 significantly deviated from float32 and flex16+5, starting from an undertrained discriminator; quality of generated images was also accordingly



https://ai.intel.com/flexpoint-numerical-innovation-underlying-intel-nervana-neural-network-processor/

Challenges in using FPGA

- Programming FPGAs
- **Integrating FPGAs into applications**
- Managing FPGAs in Cloud

What if ...

... you could easily program your FPGA using C/C++
... and get x10 performance* in a few days ?

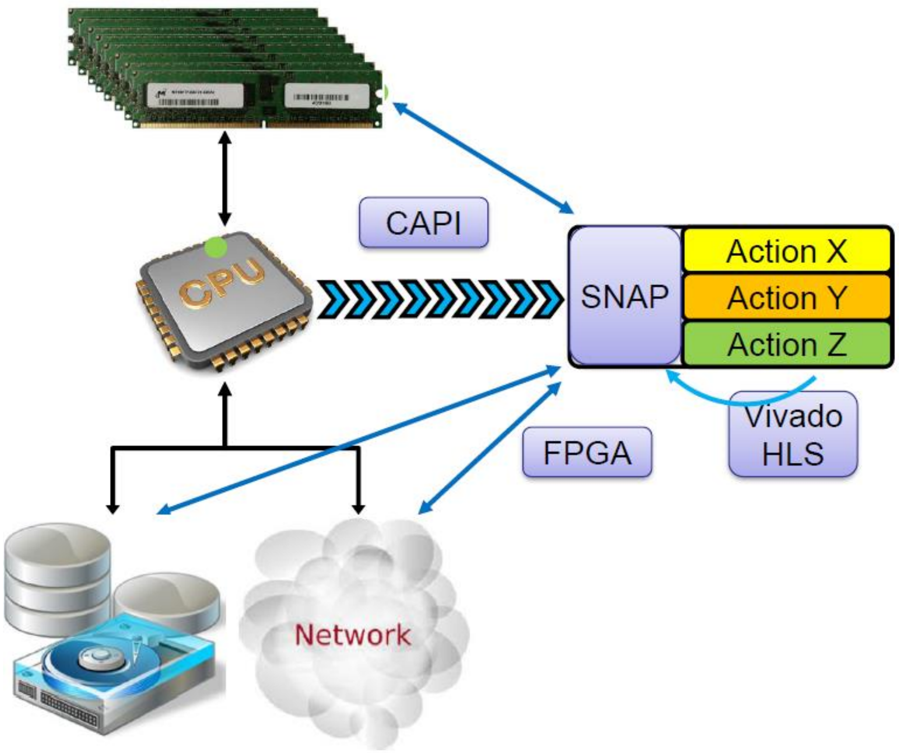
	PCI-E FPGA	CAPI FPGA	CAPI SNAP
Target Customer	Computer Engineers	Computer Engineers	Programmers
Development time	3-6 Months	3-6 Months	Days
Software Integration	PCI-E Device Driver	LibCXL	Simple API
Source Code	VHDL, Verilog, OpenCL	VHDL, Verilog, OpenCL	C/C++, Go
Coherency, Security	None	POWER + PSL	POWER + PSL

CAPI SNAP Overview, CAPI education, 2017

A framework for application developers to quickly and easily create accelerated applications on POWER.

*compared to running the same C/C++ in software

The CAPI – SNAP concept



CAPI — FPGA becomes a peer of the CPU
➔ Action **directly** accesses host memory

\+

SNAP — Manage server threads and actions
Manage access to IOs (memory, network)
➔ Action **easily** accesses resources

\+

FPGA — Gives on-demand compute capabilities
Gives direct IOs access (storage, network)
➔ Action **directly** accesses external resources

\+

Vivado HLS — Compile Action written in C/C++ code
Optimize code to get performance
➔ Action code **can be ported efficiently**

=

Best way to **offload/accelerate** a C/ C++ code with :
- Minimum change in code
- Quick porting
- Better performance than CPU

CAPI SNAP Overview, CAPI education, 2017

Why SNAP?

FPGA acceleration can provide huge **deployment** performance benefits compared to software.

However, traditional **development** of an FPGA accelerator takes specialized skills (RTL) and is quite time consuming (many month of development).

CAPI-SNAP makes it easy !

CAPI-SNAP provides the infrastructure that :

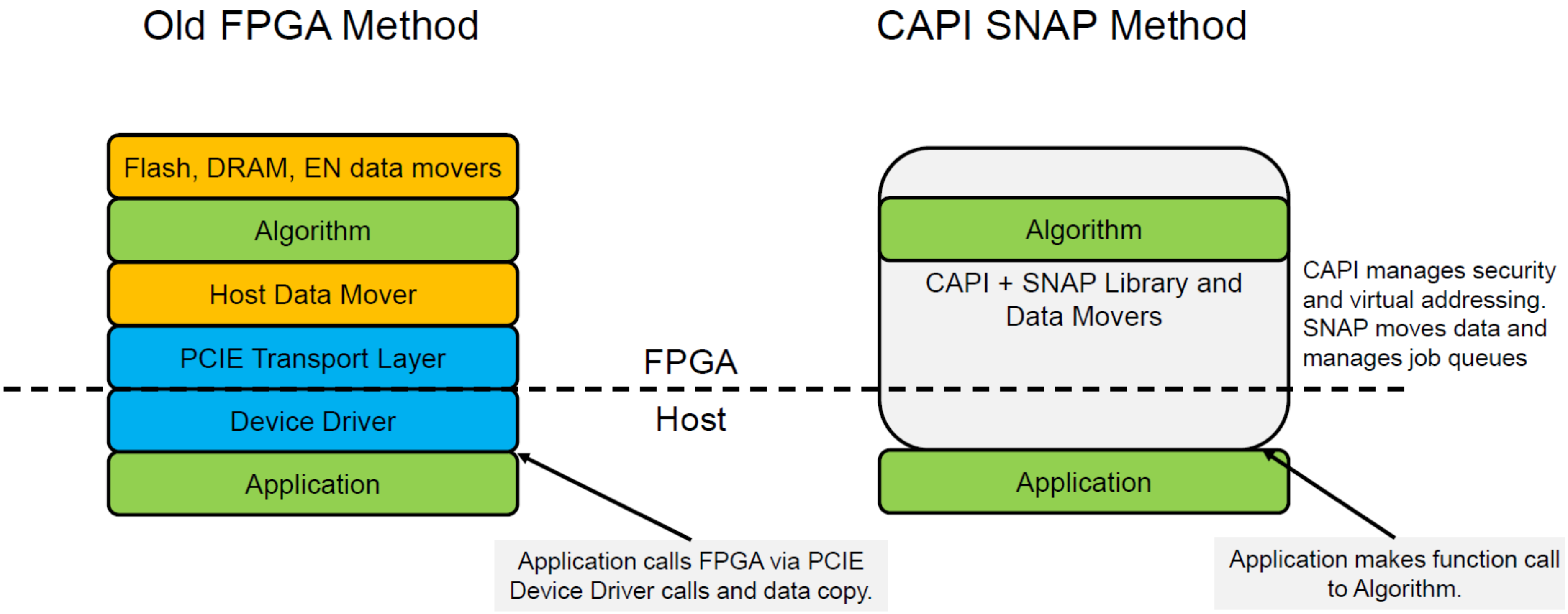
1. Allows programmers to directly port algorithms to the FPGA (e.g. C/C++->FPGA)
2. Has a simple API for the host code to call the accelerated action

Development (items 1–2)

3. Manage jobs during runtime (including virtualization)
4. Manages data access and put-away to/from the accelerated action during runtime

Deployment (items 3–4)

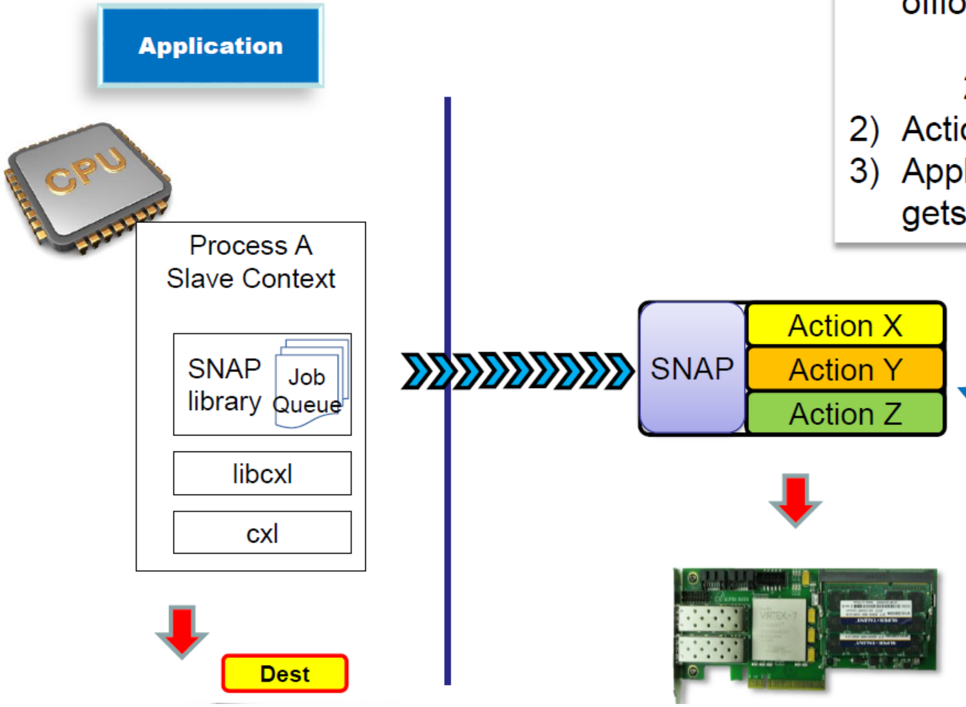
FPGA stack : then and now

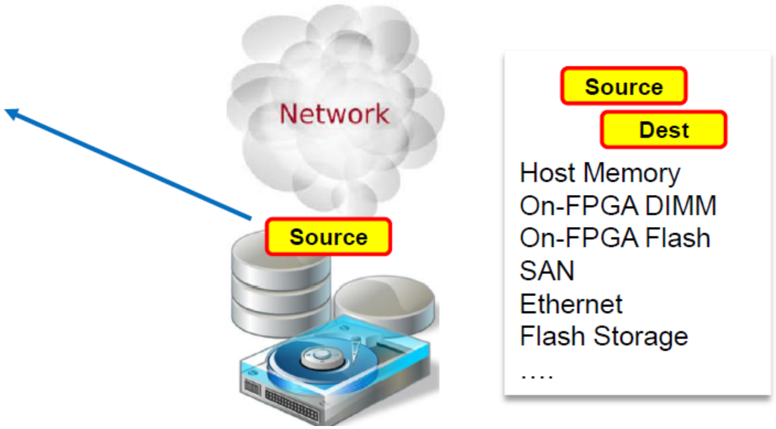


CAPI SNAP Overview, CAPI education, 2017

CAPI-SNAP: the framework

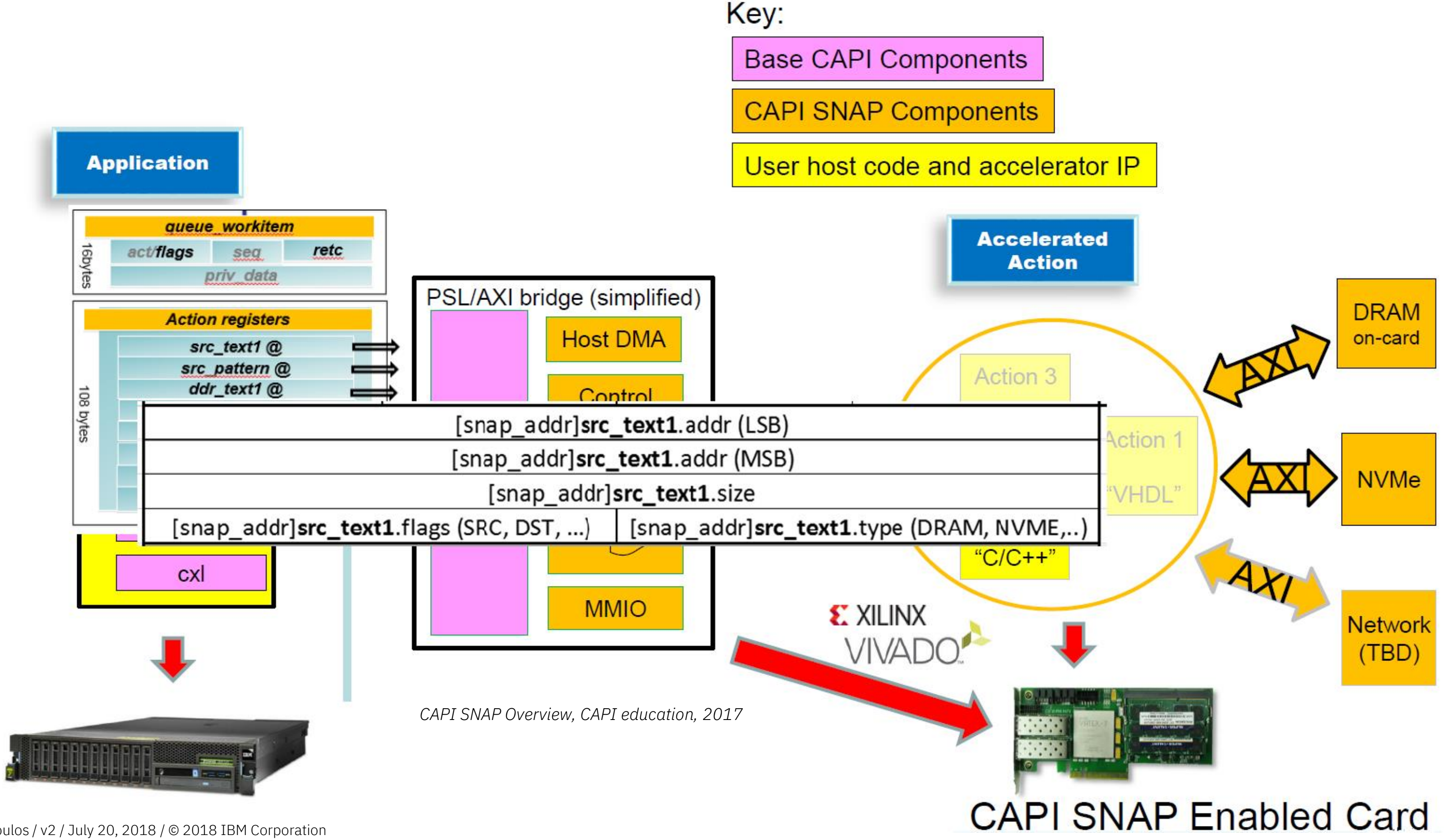
1) Application writer decides what to offload/accelerate
    1) Pointer to Source of data **Source**
    2) Pointer to where results should go **Dest**
2) Action is performed on FPGA
3) Application is informed of action completion or gets result data directly in memory





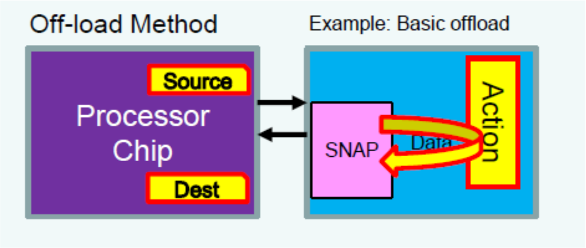
CAPI SNAP Overview, CAPI education, 2017

CAPI-SNAP: the framework

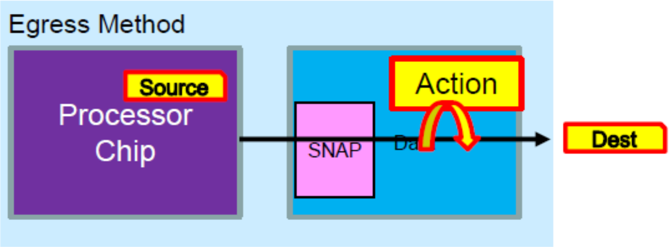


CAPI SNAP Overview, CAPI education, 2017

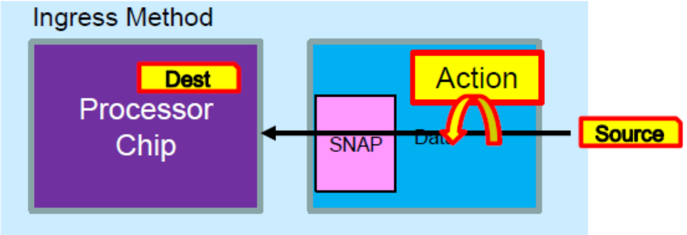
CAPI-SNAP paradigms



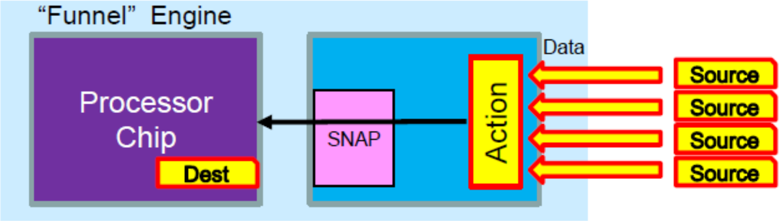
Examples: Machine Learning, Genomic algorithms, Erasure Code offload, Deep Computation



Examples: Encryption, Compression, Erasure Code prior to network or storage



Examples: Video Analytics, Deep Packet Inspection (DPI), Video Encoding (H.265) etc

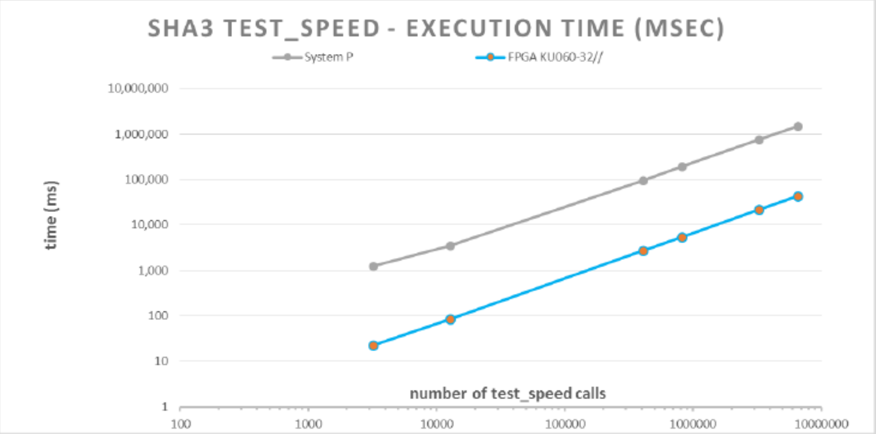


Examples: Database searches, joins, intersections, merges

CAPI SNAP Overview, CAPI education, 2017

Offload method: SHA3 kernel: *FPGA is >35x faster than CPU*

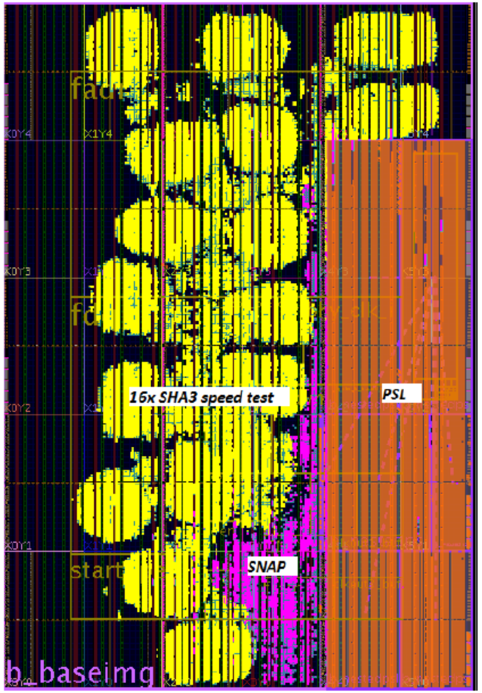
					slices/32	CPU (antipode) 16 cores - 160 threads	
					FPGA KU060-32//	**System P**	FPGA Speedup
NB_ROUNDS	**NB_TEST_RUNS**	**nb_elmts**	**freq**	**test_speed calls**	**(msec)**	(msec)	
100,000	65,536	32	65,536	3,200,000	22	1,260	57
100,000	65,536	128	65,536	12,800,000	85	3,460	41
100,000	65,536	4,096	65,536	409,600,000	2,715	95,975	35
100,000	65,536	8,192	65,536	819,200,000	5,429	190,347	35
100,000	65,536	32,767	65,536	3,276,700,000	21,709	754,198	35
100,000	65,536	65,536	65,536	6,553,600,000	43,418	1,505,790	35



CAPI SNAP Overview, CAPI education, 2017

Funnel engine: Array intersection benchmark

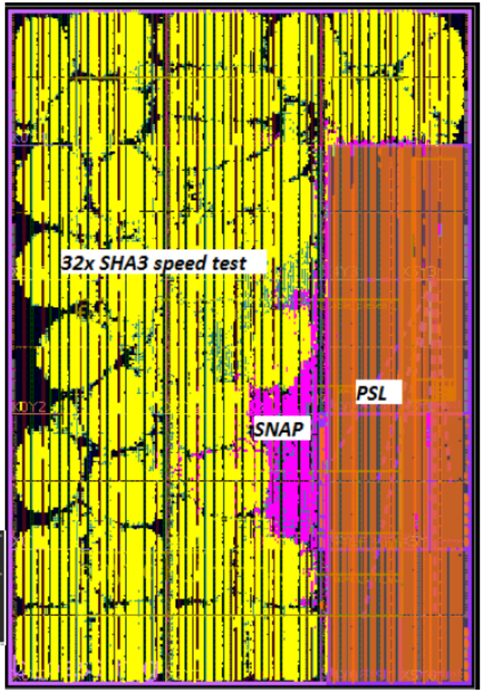
_16 test_speed functions in parallel:_



Site Type	Used	Fixed	Available	Util%
CLB LUTs	151842	69756	331680	45.78
LUT as Logic	137137	55073	331680	41.35
LUT as Memory	14705	14683	146880	10.01

_32 test_speed functions in parallel:_

Site Type	Used	Fixed	Available	Util%
CLB LUTs	225387	69756	331680	67.95
LUT as Logic	210666	55073	331680	63.51
LUT as Memory	14721	14683	146880	10.02



CAPI SNAP Overview, CAPI education, 2017

Funnel engine: Array intersection benchmark

Given two unsorted arrays, write a function that returns the third array which is the intersection of the two arrays. This means that the resulting array should only contain elements that appear in both input arrays. Order of elements in the resulting array is irrelevant.

SW CPU procedure:
- Copy Source arrays from DDR to Host
- SW intersection

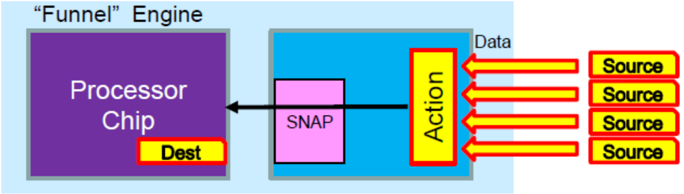
FPGA procedure:
- HW intersection
- Copy Result array from DDR to Host

KBytes	Host total	Host executior	Host Access	FPGA Total	FPGA executi	FPGA Result store	FPGA speedu
16	30686	**30,621**	65	265	**228**	37	134
32	33668	**33,562**	106	467	**422**	45	80
64	34165	**33,978**	187	925	**819**	106	42
128	28706	**28,354**	352	1760	**1,627**	133	17
256	32467	**31,775**	692	3500	**3,245**	255	10

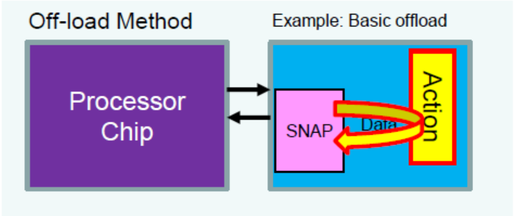
CAPI SNAP Overview, CAPI education, 2017

Key questions to identify candidate algorithms

- What is the first operation you do on data as you pull it into the server?
  - Are you culling the data?
    - Search? Merge? Join? Intersections?

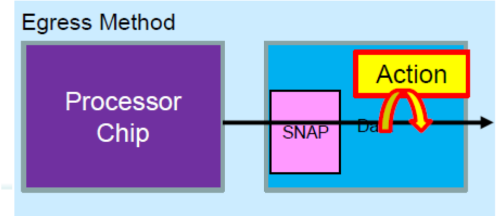


- Do you have long running algorithms?
  - What code does your profiling identify as taking a high percentage of your CPU time?
  - Do you have a lot of recursion or looping?
  - Numerical intensive operations?



- Are you doing data clean-up or formatting before storing to IO?

CAPI SNAP Overview, CAPI education, 2017



SNAP enabled card details: Alpha-Data ADM-PCIE-KU3

3.5MB Block Ram
on FPGA

Two 40Gb QSFP+ Ports
Future Use: Currently no Bridge to SNAP

8GB DDR3
Latency to FPGA: 230ns

FPGA to Host Memory Access
Latency to/from FPGA: 0.8us
Bandwidth to FPGA: ~3.8GB/s reads and writes

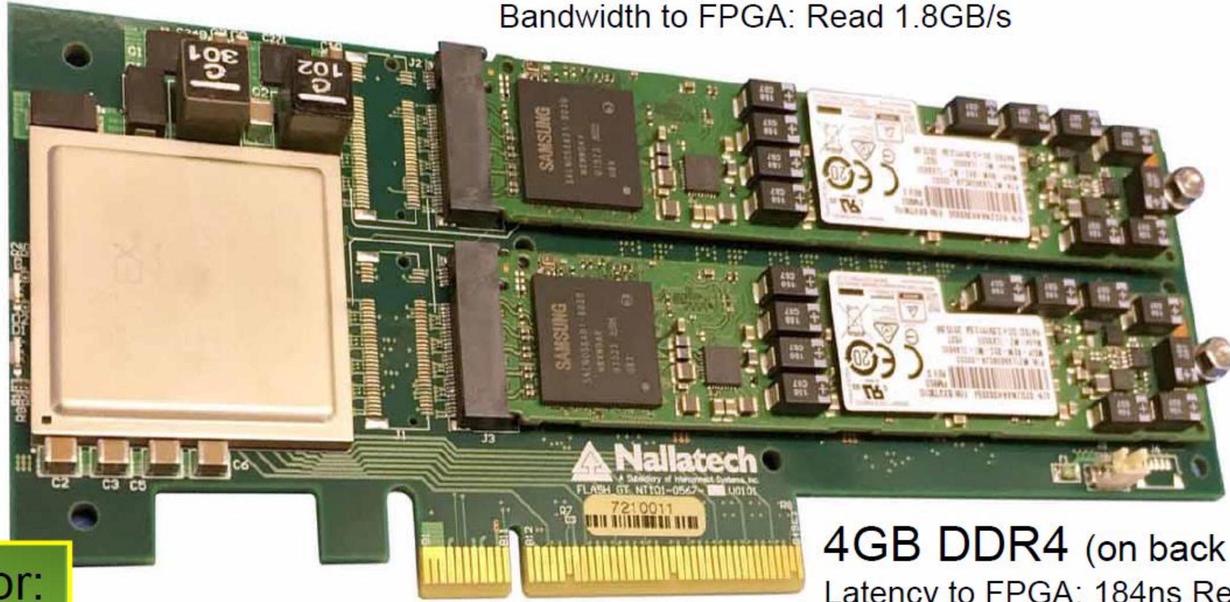
Choose this card for:
External IO
Offload and DRAM

CAPI SNAP Overview, CAPI education, 2017

SNAP enabled card details: Nallatech 250S

Two 1TB NVMe sticks (1.92TB effective)
Latency to FPGA: ~0.8ms
Bandwidth to FPGA: Read 1.8GB/s

3.5MB Block Ram on FPGA



4GB DDR4 (on back of card)
Latency to FPGA: 184ns Read / 105ns write

Choose this card for:
2TB of on-card Flash

FPGA to Host Memory Access
Latency to/from FPGA: 0.8us
Bandwidth to FPGA: ~3.8GB/s reads and writes (CAPI limit)

CAPI SNAP Overview, CAPI education, 2017

Job-Queue mode

When to use this mode?

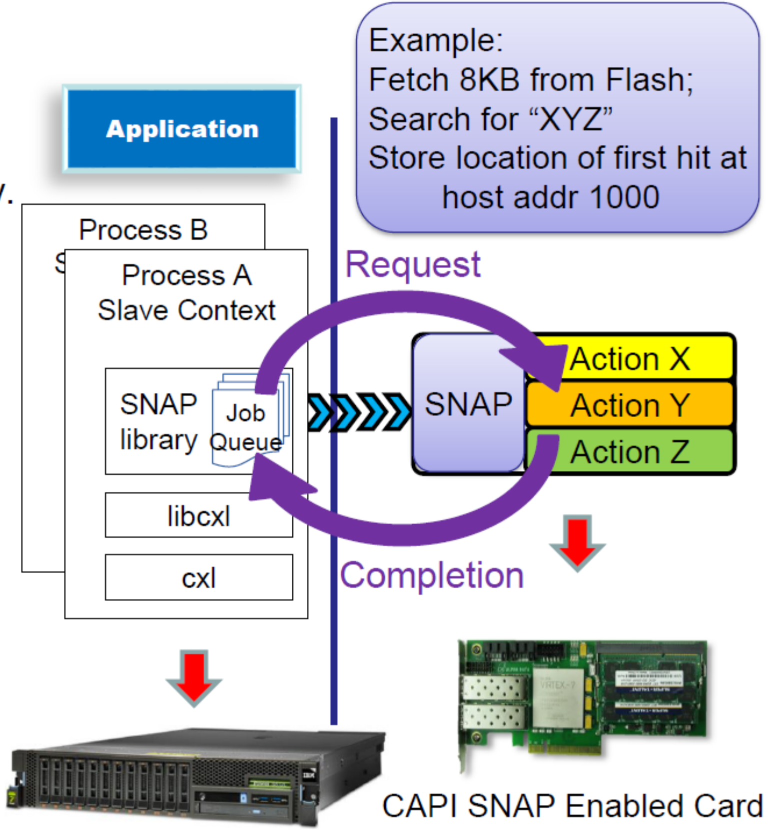
- FPGA-action executes a job and returns after completion
- Action can be called by multiple processes simultaneously.

What do you get?

- Support for multiple processes N scheduled on a single action.
  - Future: multiple FPGA-actions M virtually in parallel controlled by built in job-managerGeneric job-execution model with request and completion queue per AFU context
- Prefetching of memory areas, if possible

Possible use-cases

- FPGA acceleration within a Cloud, e.g. using Docker virtualization



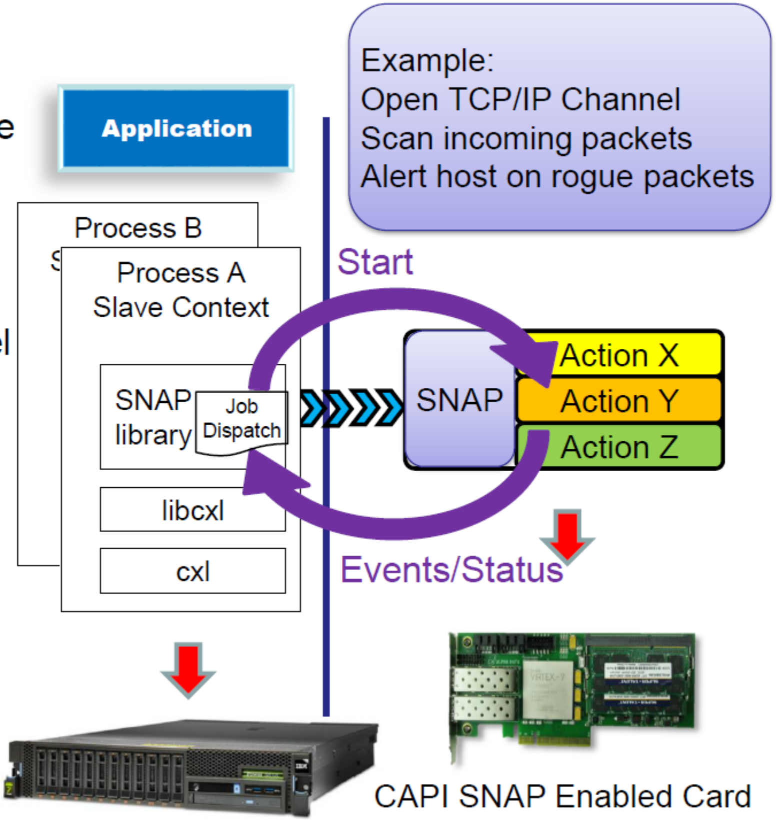
CAPI SNAP Overview, CAPI education, 2017

Direct-access mode

- When to use this mode?
  - FPGA-action is designed to permanently run
  - Data-streaming approach with data-in and data-out queue
  - Event driven operation

- What do you get?
  - Support for N processes using N FPGA-actions in parallel
  - FPGA-action attachment to one process exclusively
  - Selected FPGA-action MMIOs are mapped into the process address space
  - Dedicated interrupt(s) per action
  - Process A and Process B occur sequentially (no concurrence)

- Possible use-cases
  - Use-cases where FPGA-action must permanently run
  - Networking



CAPI SNAP Overview, CAPI education, 2017

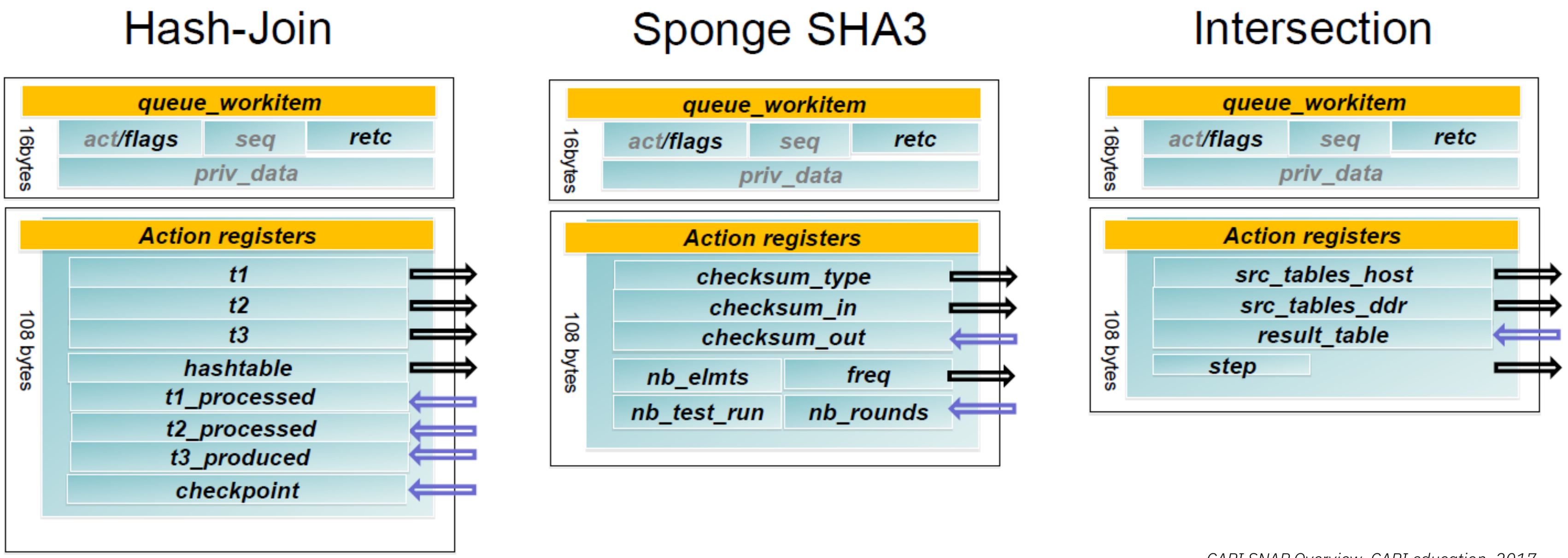
Define your API parameters

Key questions:

- Does your action require parameters?
- Where does your data reside (source data)?
- For Job Queue mode: What happens when your action completes?
    - How does your application know that an action completed?
    - Are there results (destination data)?
- For Direct Access Mode:
    - How does the accelerator start? Do you need to open a channel?
    - What happens when the action observes an event?
- Does your application need to monitor the “action”? (e.g. MMIO register)

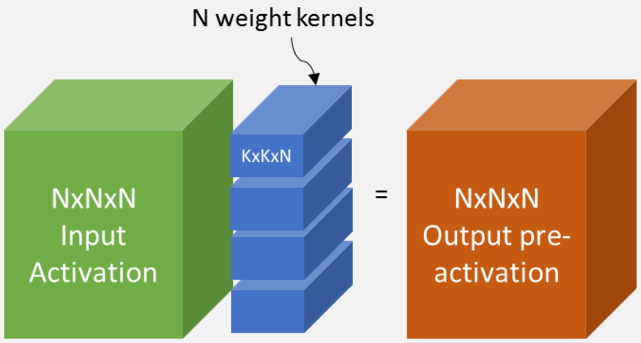
Define your API parameters

API will be a structure passed to the SNAP Library.
Some examples:

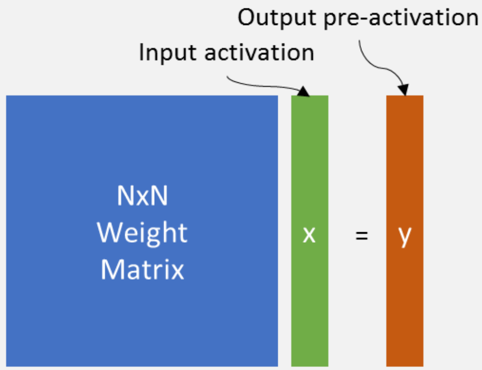


CAPI SNAP Overview, CAPI education, 2017

Common Scenarios for accelerators (NN example)



$O(N^3)$ data
$O(N^4K^2)$ compute



$O(N^2)$ data
$O(N^2)$ compute

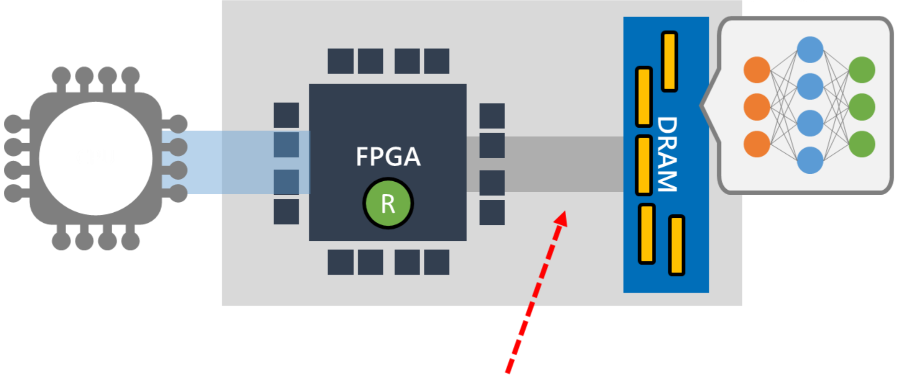
Convolutional Neural Network (CNN)
High Compute-to-Data Ratio

MLPs, LSTMs, GRUs
Low compute-to-data ratio

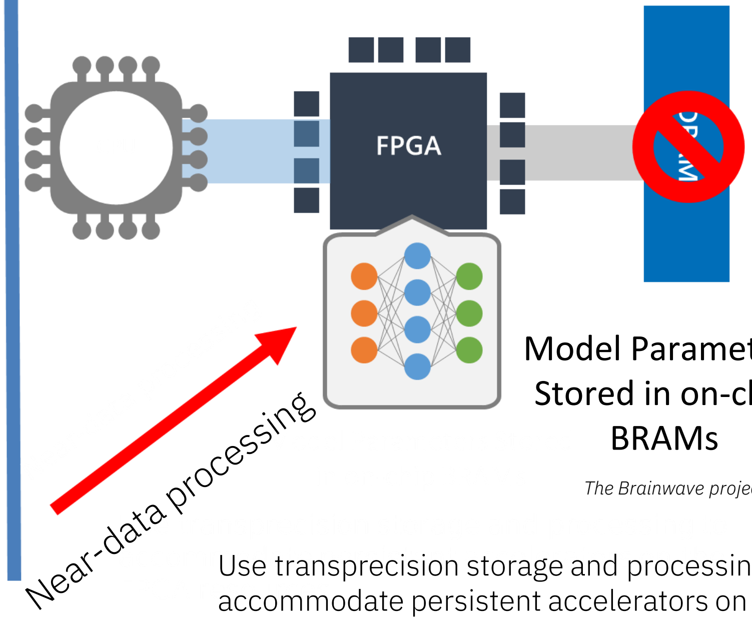
The Brainwave project, 2017

Common Scenarios for accelerators (NN example)

Model Parameters Initialized in DRAM



DRAM read energy much higher than FPGA processing

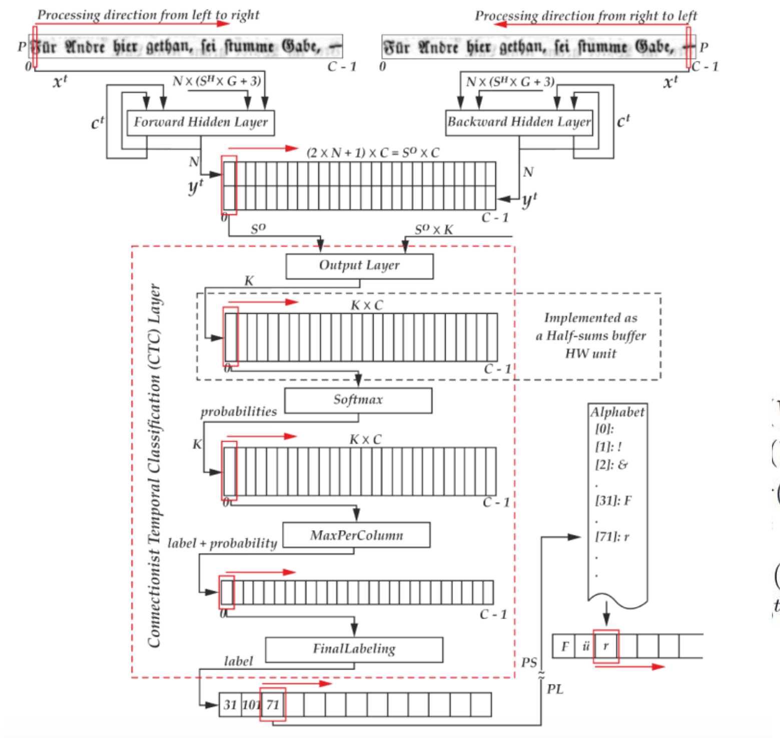


Model Parameters Stored in on-chip BRAMs

The Brainwave project, 2017

Use transprecision storage and processing to accommodate persistent accelerators on the FPGA resources

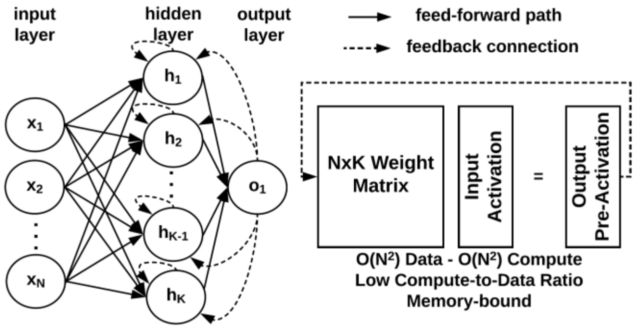
Bidirectional Long-Short Term Memory based OCR



Rybalkin, V., et al., *Hardware Architecture of Bidirectional Long Short-Term Memory Neural Network for Optical Character Recognition, 2017.*

- Input: Image of one Text-Line
- Algorithm:
  - FW and BW pass to updated LSTM cells
  - Merge and produce final prediction
- Output: Detected text sequence

$$W_I x^t + R_I y^{t-1} + b_I)$$
$$(W_i x^t + R_i y^{t-1} + p_i \odot c^{t-1} + b_i)$$
$$(W_f x^t + R_f y^{t-1} + p_f \odot c^{t-1} + b_f)$$
$$\odot I^t + f^t \odot c^{t-1}$$
$$(W_o x^t + R_o y^{t-1} + p_o \odot c^t + b_o)$$
$$^t \odot h(c^t)$$

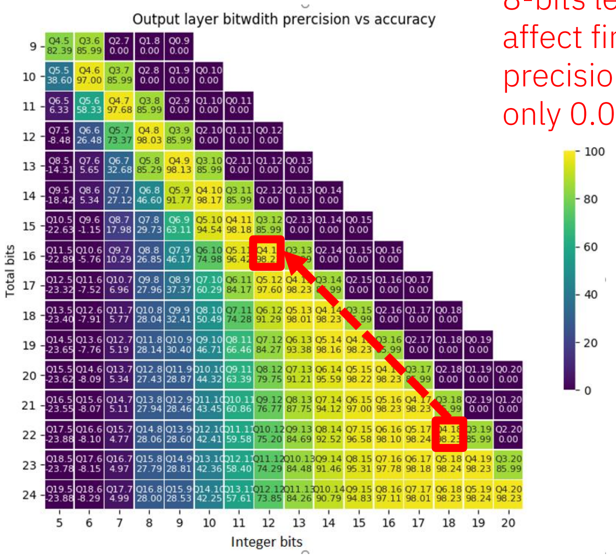


- Baseline performance: 98.23% accuracy (float)

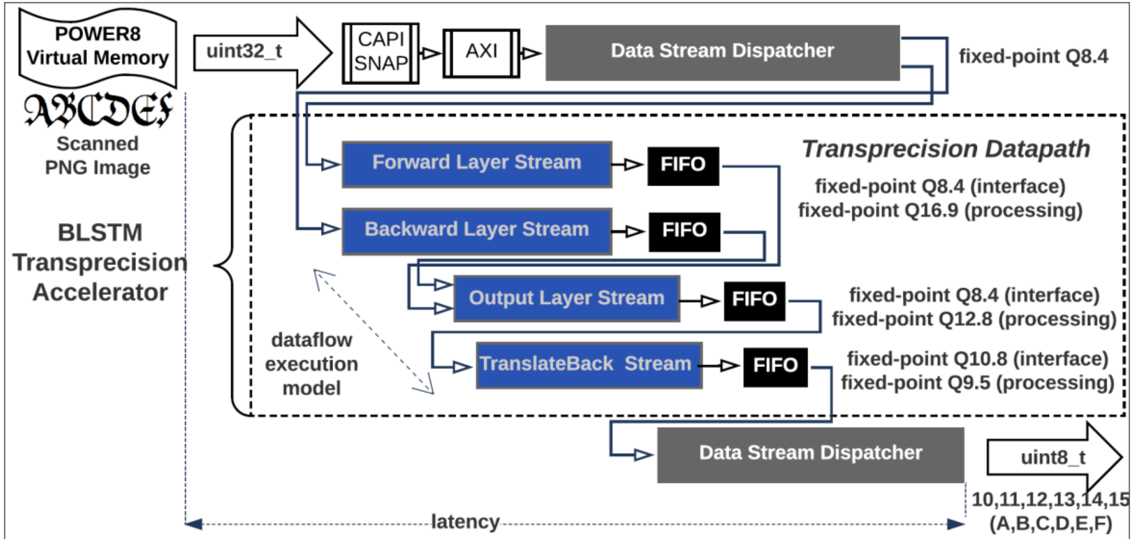
Transprecision accelerator for BLSTM algorithm

The BLSTM AXI4 transprecision accelerator

- transprecision pipeline:
  - four streaming engines in dataflow architecture, each with different fixed-point format.
- Weight & activation precision calibration
  - quantization of trained FP32 models to custom fixed-point, while minimizing accuracy loss.
  - high-throughput & resource-efficient accelerator for FPGA.

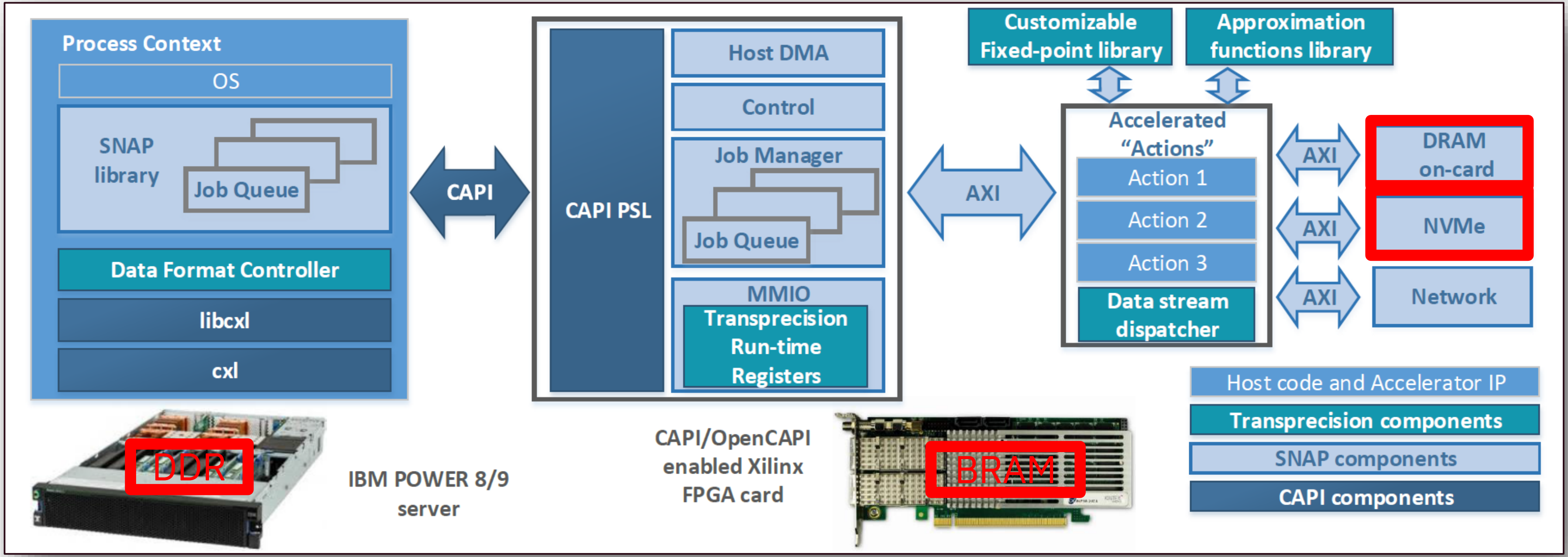


8-bits less affect final precision with only 0.01%



Architecture for near-memory transprecision accelerators

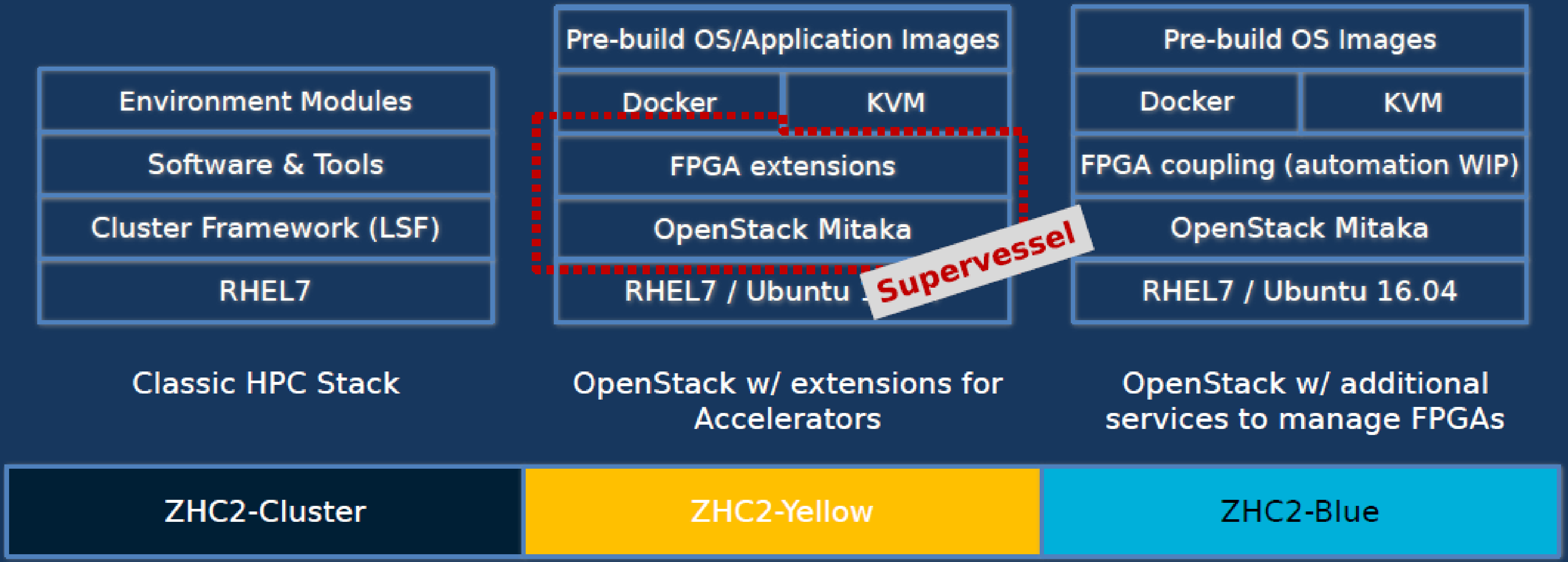
- System stack innovation to drive Cost/Performance.
- Libraries and tools to intergrade energy-efficient FPGA accelerators to commodity HW and SW.



Challenges in using FPGA

- Programming FPGAs
- Integrating FPGAs into applications
- **Managing FPGAs in Cloud**

Zurich Heterogenous Compute / Cloud



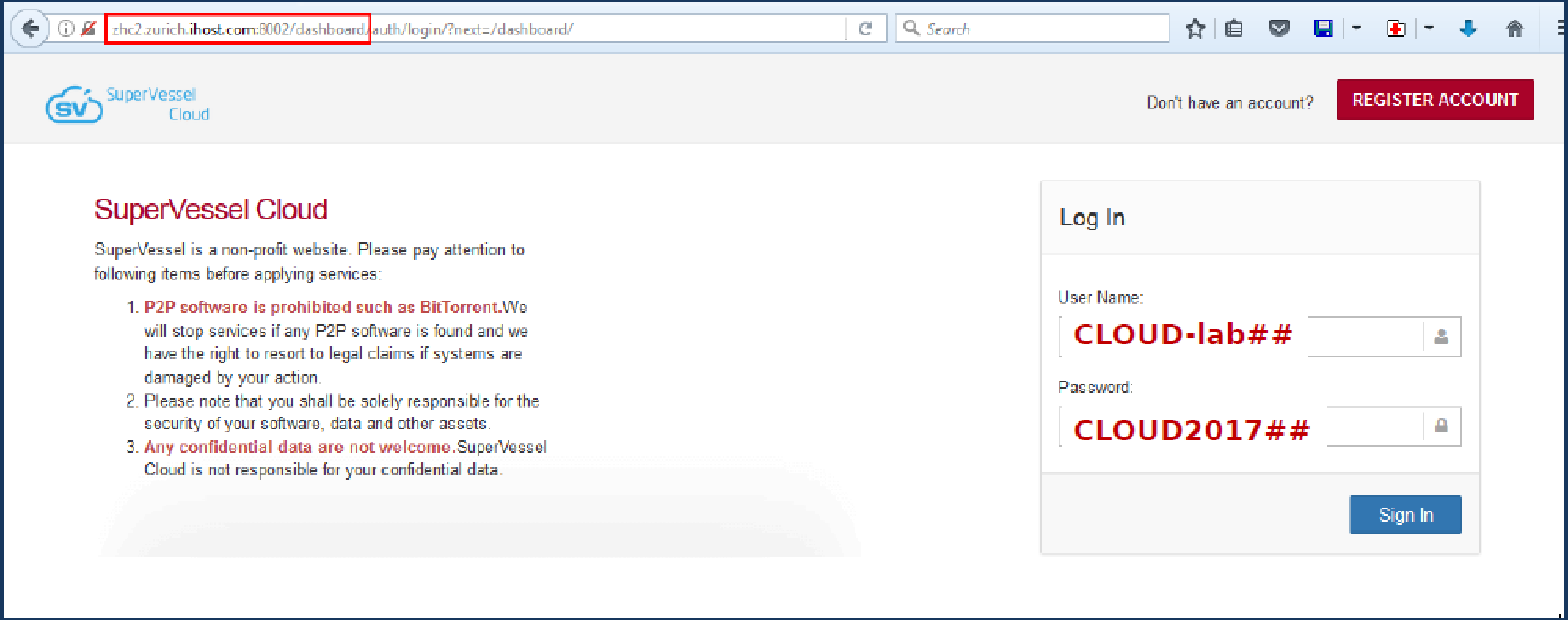
Homogeneous cluster (Minsky)

Mix of different nodes and accelerators (x86/POWER, GPU/FPGA)

Prototype for Hyperscale FPGA (Homogeneous FPGA resources)

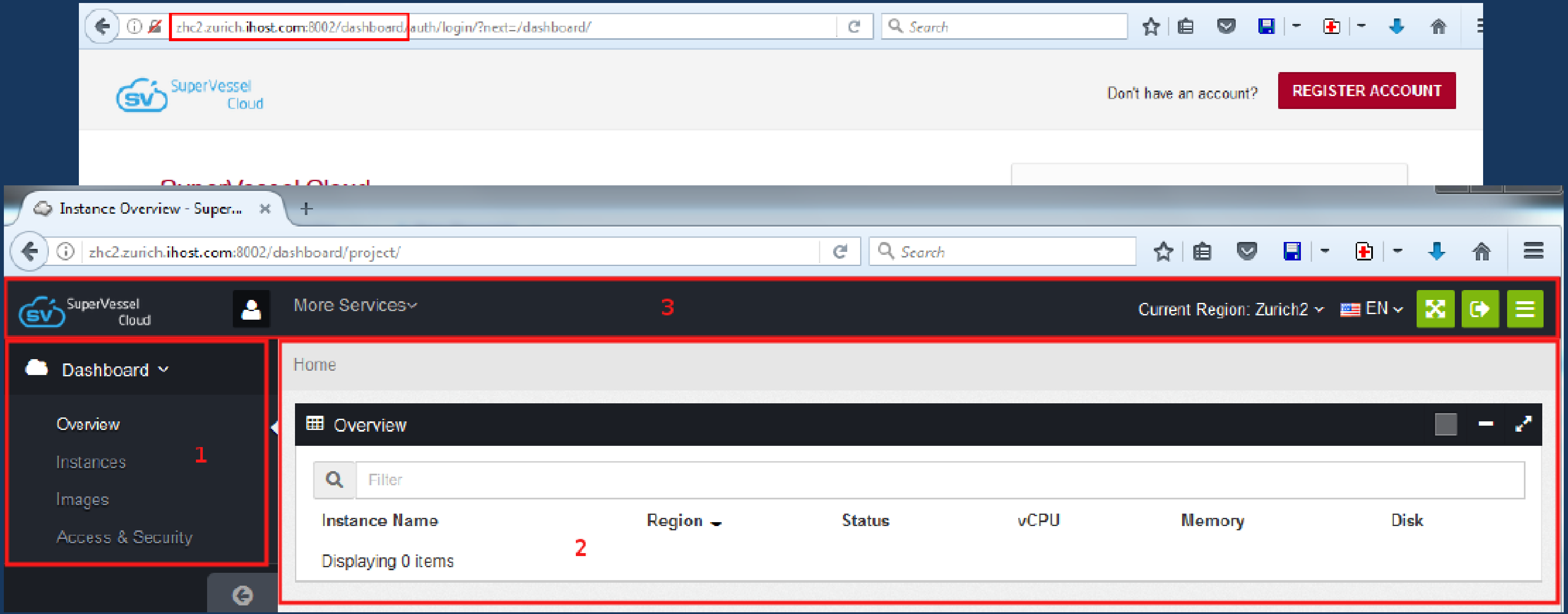
Zurich Heterogenous Compute / Cloud

zhc2.zurich.ihost.com:8002/dashboard or http://ibm.biz/supervessel_ch

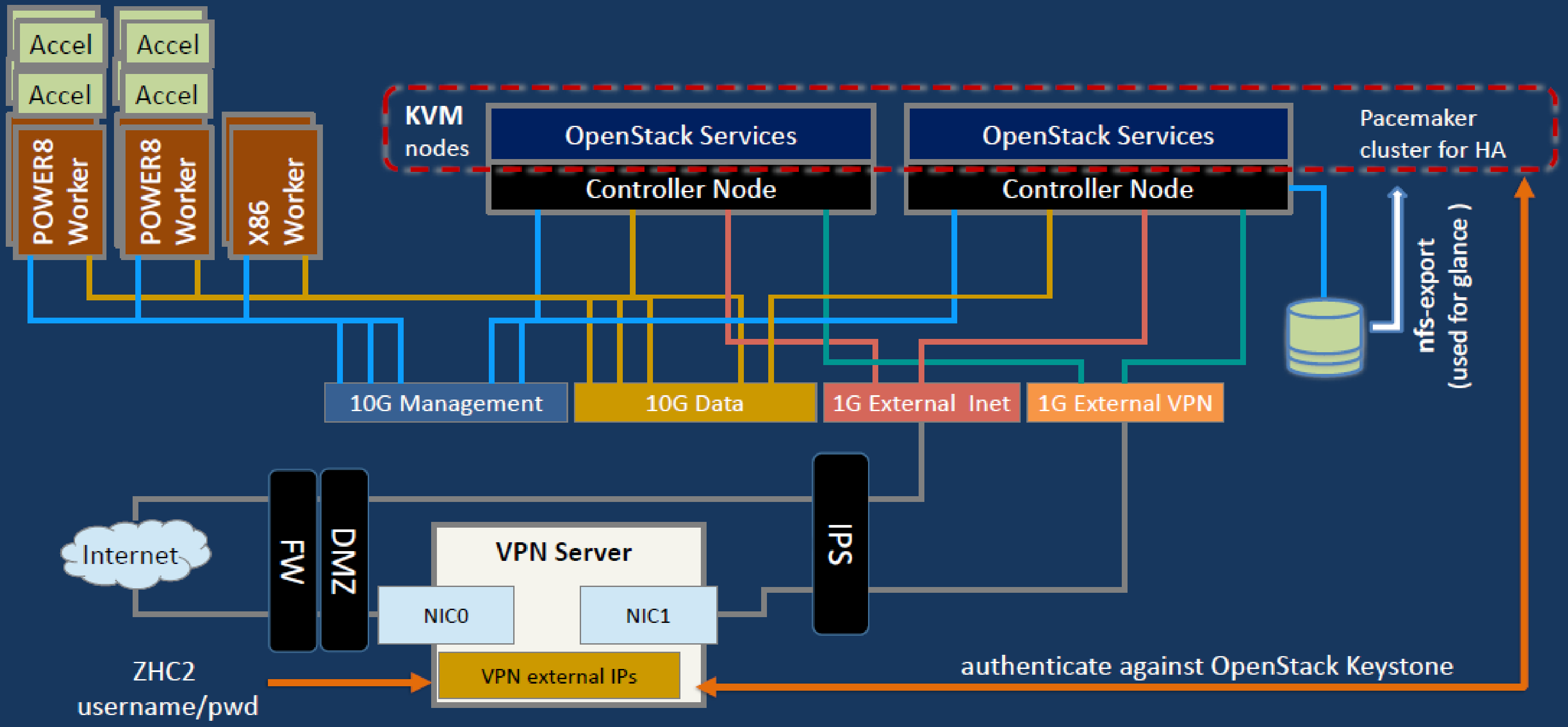


Zurich Heterogenous Compute / Cloud

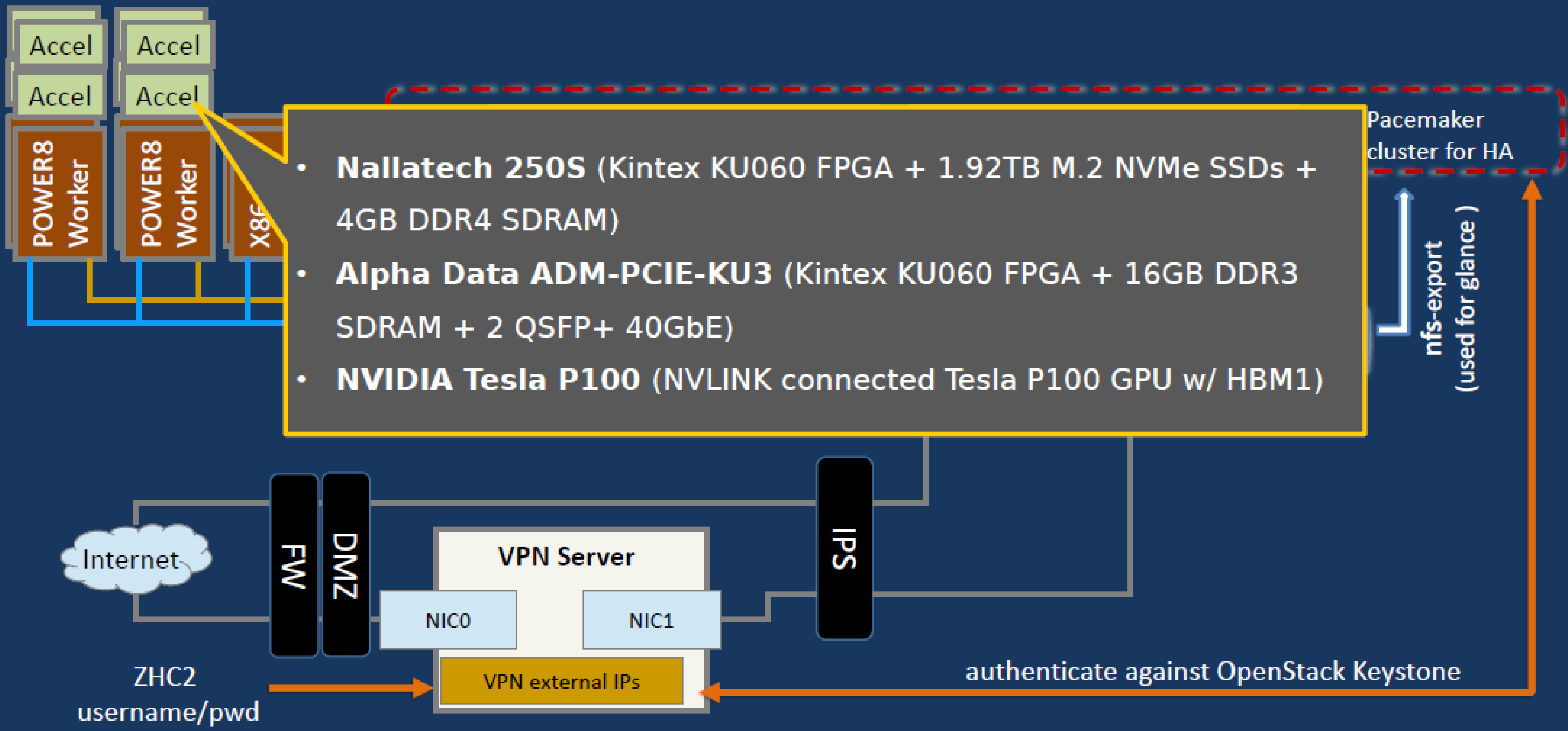
zhc2.zurich.ihost.com:8002/dashboard or http://ibm.biz/supervessel_ch



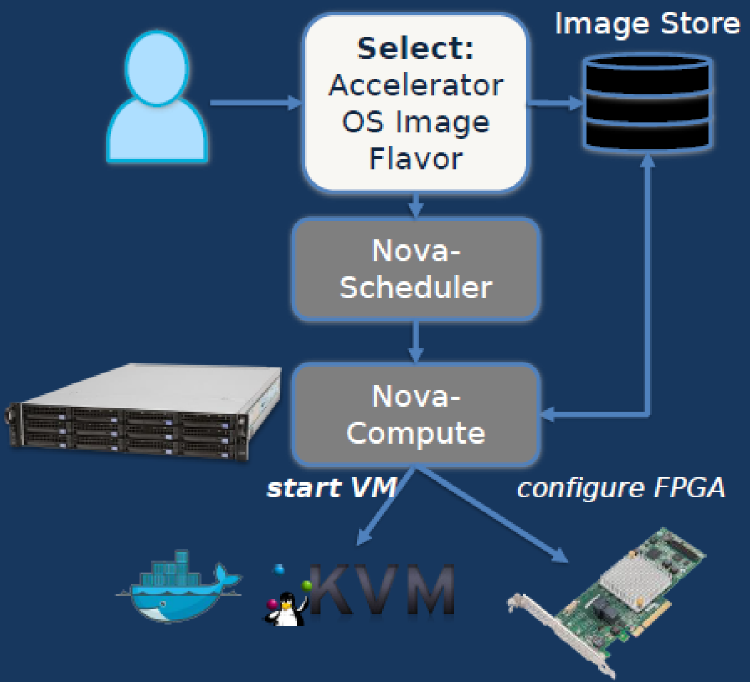
ZHC2 Yellow - system architecture

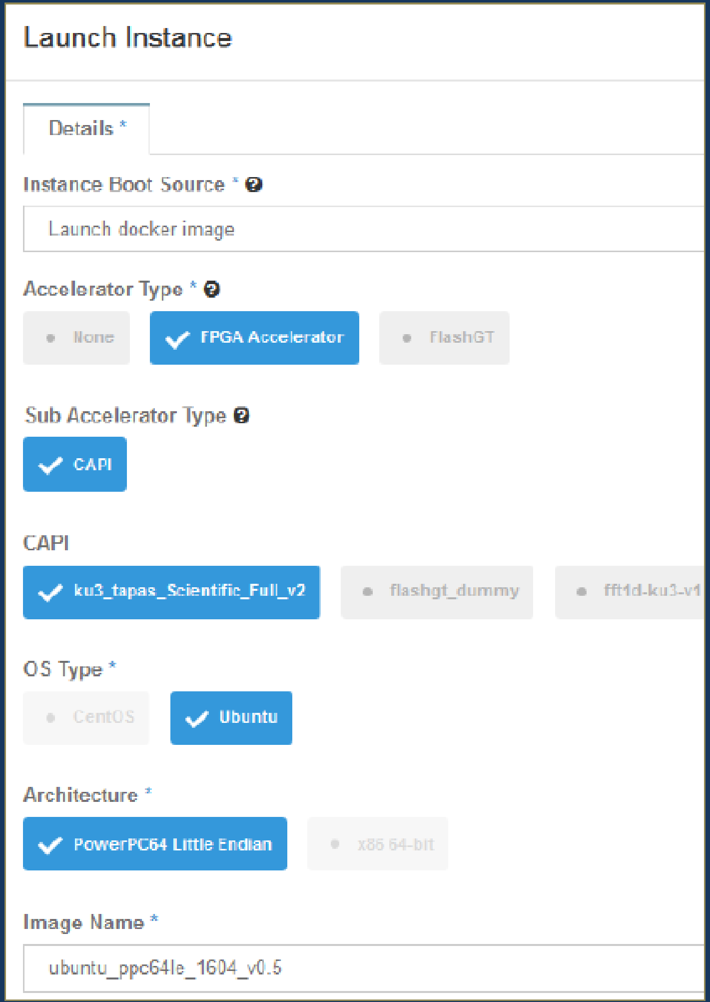


ZHC2 Yellow - system architecture



VM Configuration

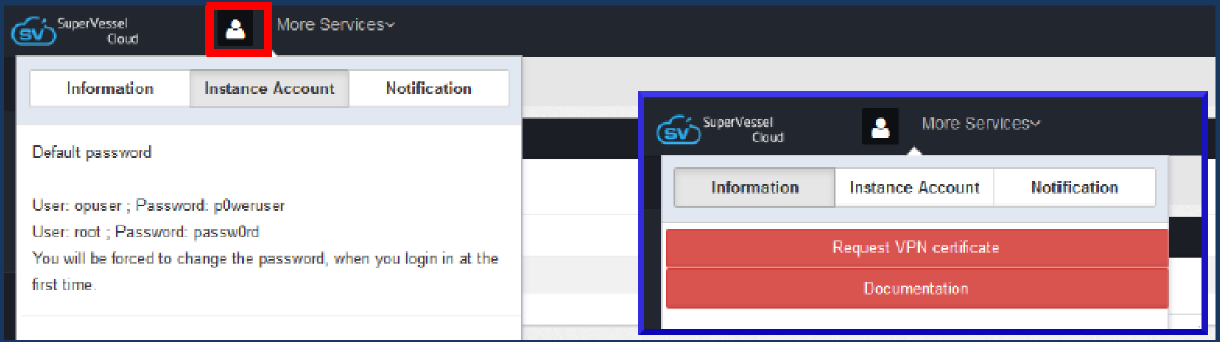


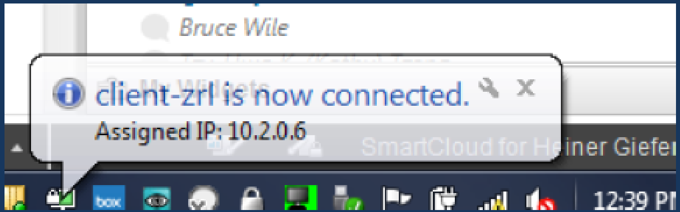


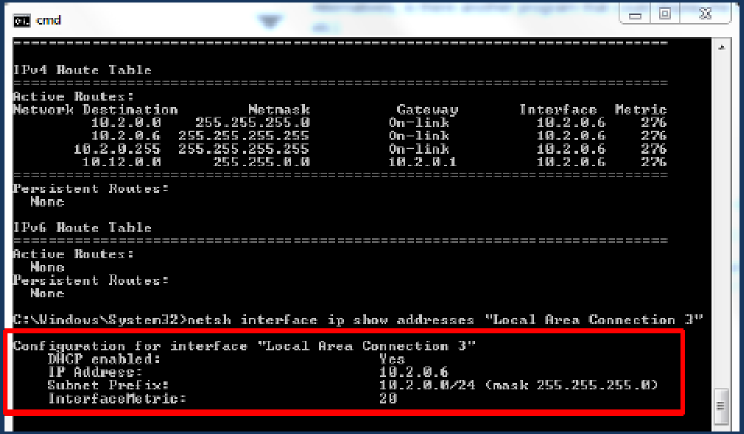
VPN access

You need to install an OpenVPN compliant VPN client, i.e.

- Tunnelblick (Mac OS)
- OpenVPN GUI (Win)
- OpenVPN (Linux)

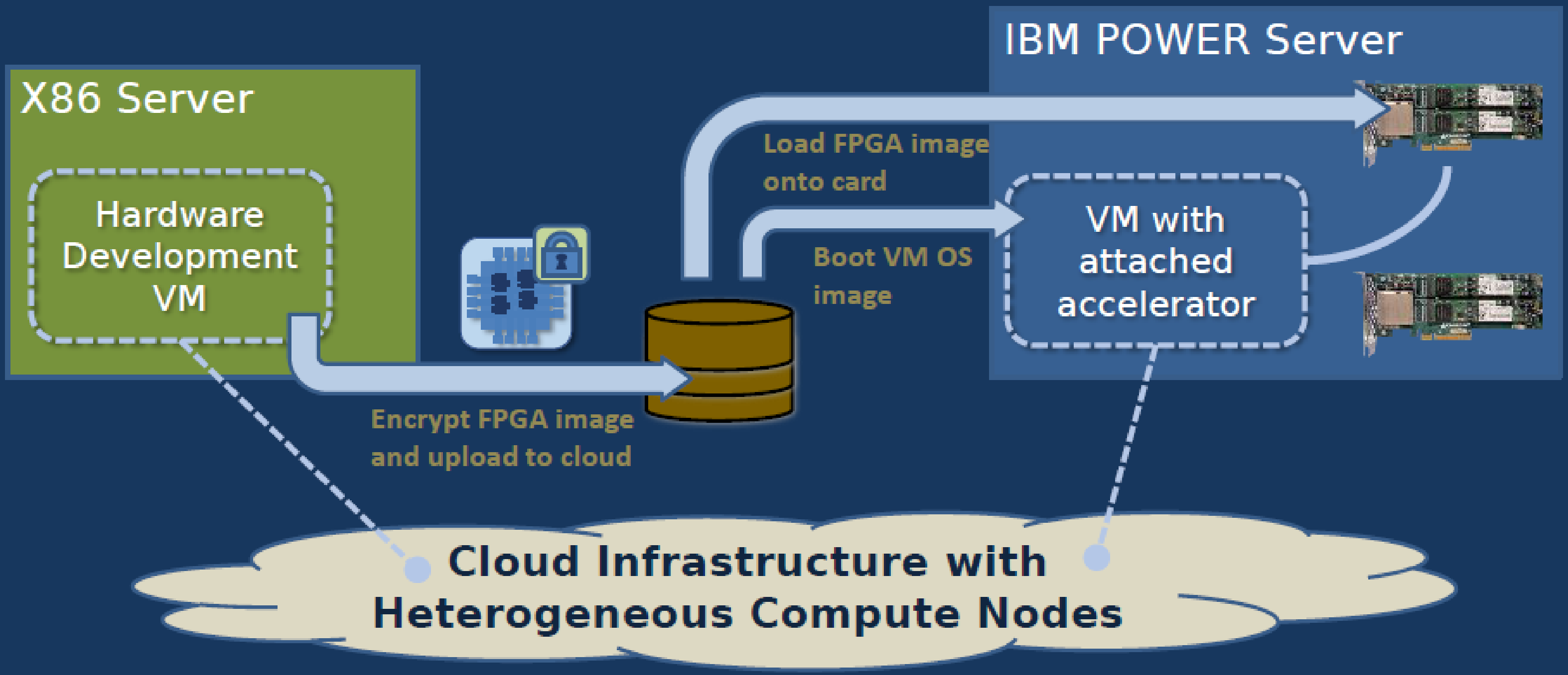






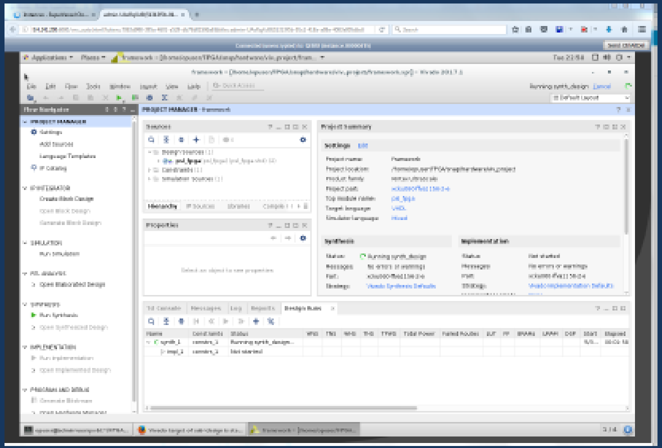
Your local VPN IP: 10.2.0.x
Your instances External IP: 10.12.0.y

Using FPGAs in Heterogenous Cloud

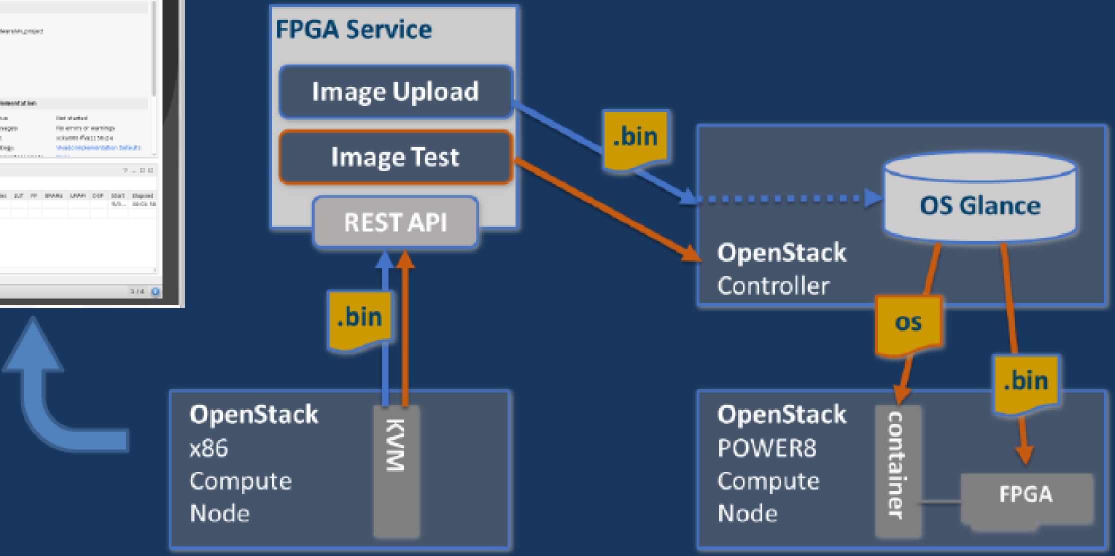


FPGAs Development in ZHC2

1. Upload FPGA configuration to cloud image store
2. Create new instance with configured FPGA attached



FPGA Development



Research insight

- Showcasing a catalyst for HPC through OpenPOWER ecosystem

OpenPOWER™

OPRECOMP
Open Transprecision Computing

Vibrant ecosystem through open collaborative & development

+

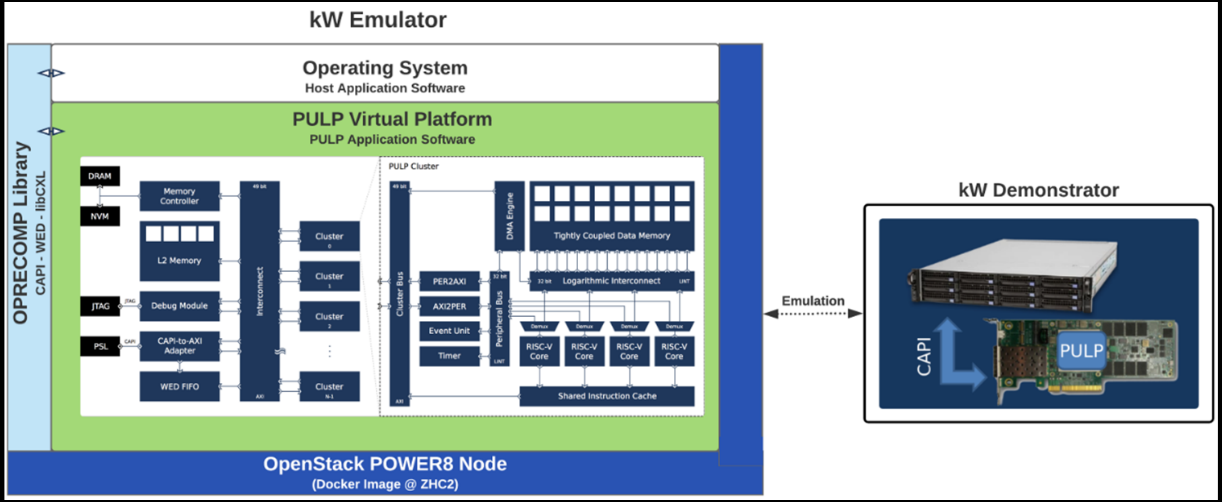
Energy-efficient computing through precision relaxation

High Performance Computing
P8/P9 nodes

Cloud Computing
Hyperscale & Large scale Datacenters

Transprecision Computing Architecture
Functional Units, Processing Elements, Communication Infrastructure, Memory Hierarchies, Runtime Management, Programming Model

Emulation-As-A-Service



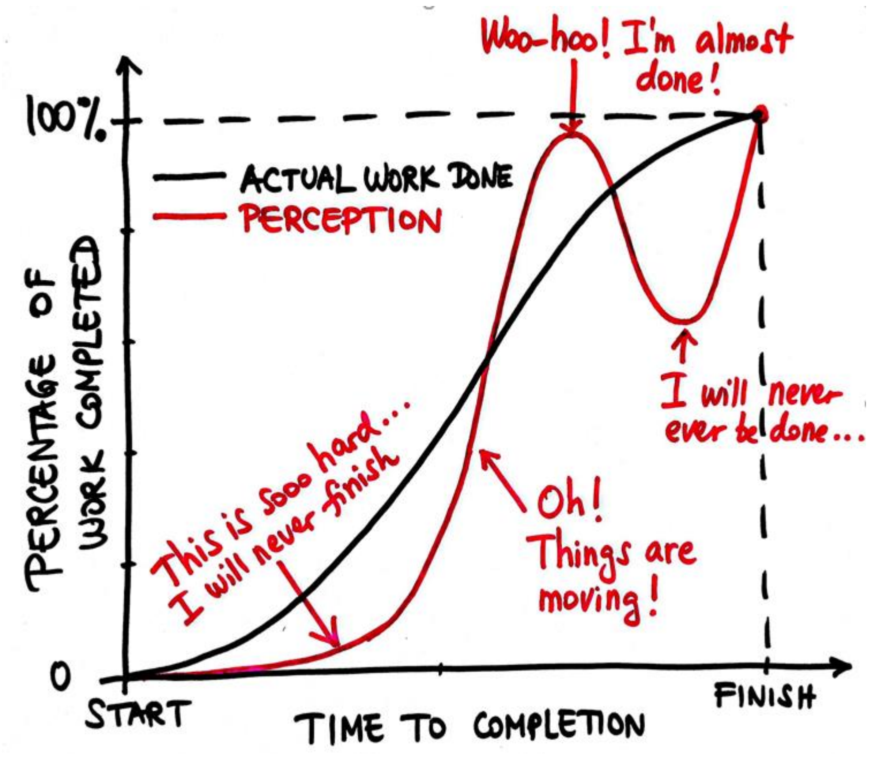
Accelerator-As-A-Service

Thank you

Dionysios Diamantopoulos
PostDoc Researcher - Accelerator Technologies
—
did@zurich.ibm.com
+41-44-724-85-25

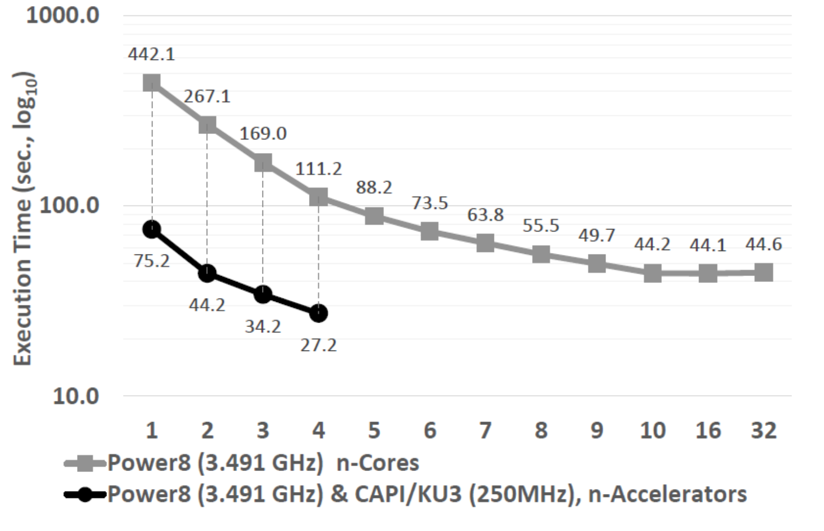
The Golden Age of FPGAs is about to start!

It will be fun to be part of it!



backup

Implementation results



Instantiated up to 4-Accs on the FPGA

- comparing with up to 8-POWER8 cores (affinity OpenMP threads / cores = 1:1).
- saturating the available FPGA on-chip memory with 96%.
- constant speedup of 4.8-5.6x using the same acceleration scalability step, i.e. OpenMP threads for software and FPGA accelerators for HW.
- Minimal CPU usage of "HW solution" (interrupt-based CAPI API).

- 22x energy efficiency in kPixels/Je.
- Negligible accuracy loss compared to software (<0.6% for 3401 images).
- BLSTM in synthesizable C++
    - Algorithmic, Transp. & HLS optimizations: from 8sec/image to 44ms/image

	FPGA			CPU					
Cores/Accs	1	2	4	1	2	4	10	16	32
Power (W)	**11.8**	12.1	12.9	**112**	119	125	179	198	216
Time (Sec.)	75.2	44.6	**28.3**	442	267	111	44.2	**44.1**	44.6
kPixels/s	827	1395	**2199**	140	233	560	1408	**1411**	1395
kPixels/J	70.0	115	**170**	1.25	1.9	4.5	**7.8**	7.12	6.45
kJ/solution	0.88	0.54	**0.36**	49.5	31.7	13.8	**7.91**	8.73	9.63

Floorplans of 1-4 BLSTM accelerators on KU3