LHC Level – 1 Trigger Software and Architecture

Varun Sharma

University of Wisconsin – Madison, USA

HSF – India, Hyderabad India
13 – 17th January 2025

Acknowledgements

Some of these slides are compiled with material from several people: **Sridhara Dasu**, **Wesley Smith**, **Sergio Cittolin**, **Tom Gorski**, **Ales Svetek**, **Sascha Savin**, **Piyush Kumar**, **Isobel Ojalvo**, **Kevin Stenson**, **…**

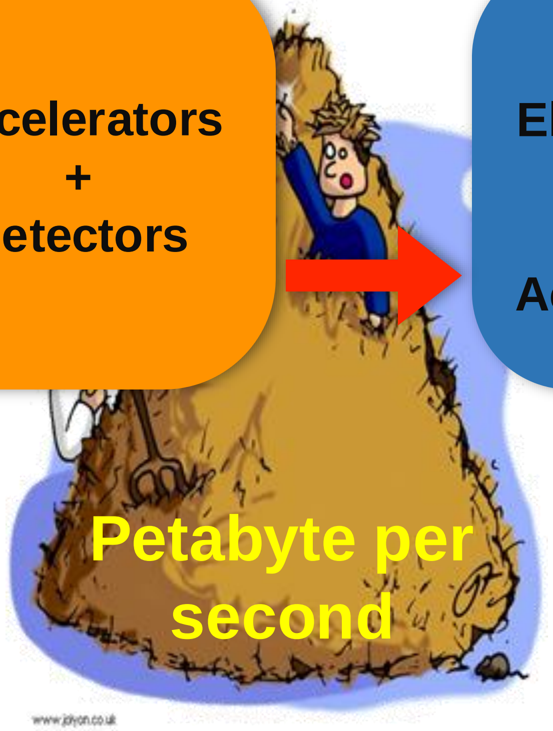
What do we want?

Scientific discoveries

Journal Publications

We have observed a new boson with a mass of 125.3 ± 0.6 GeV at 4.9 σ significance !
Hurray, We found it!

Scientific Process @ Colliders



Accelerators
+
Detectors

→

Trigger
Electronics
&
Data
Acquisition

→

Software
&
Computing

→

SCIENCE
Journal Publications

Petabyte per second

Petabytes per year

Scientific Process @ Colliders

Accelerators
Detectors

Trigger
Electronics
&
Data
Acquisition

Software
&
Computing

SCIENCE
Journal Publications

Petabytes get reduced to a social media post!

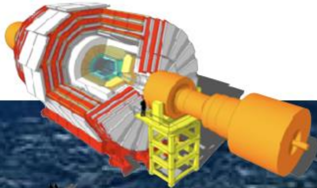
Petabyte per second

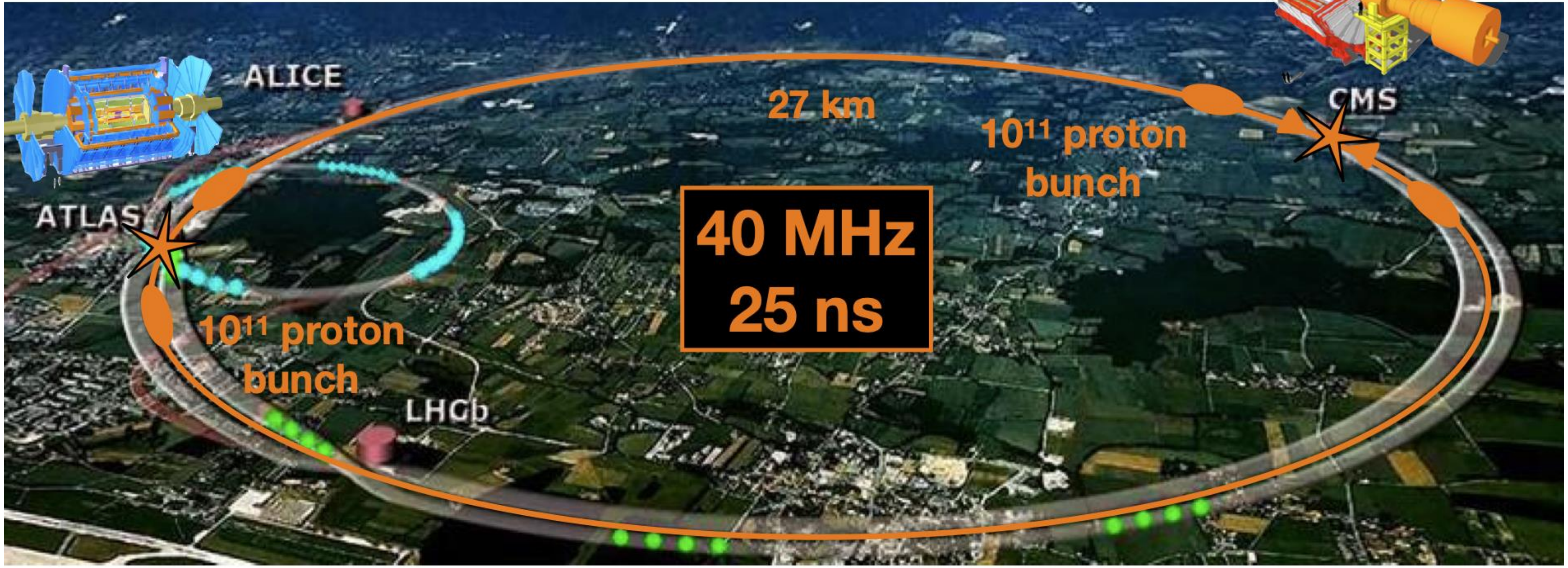
Petabytes per year

Large Hadron Collider

pp collisions @ $\sqrt{s}$ = 13.6 TeV

CMS



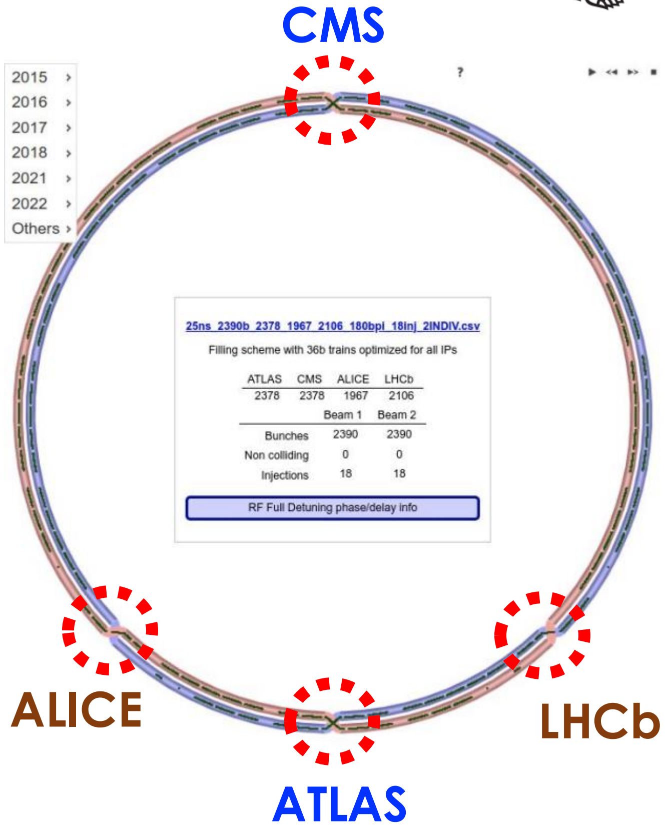


Large Hadron Collider

The LHC accelerates bunches of billions of protons (or ions) from 450 GeV injection energy from SPS to 6.8 TeV and collides them at **13.6 TeV** centre-of-mass energy

LHC circumference is 27km and the minimal distance between bunches is 25ns*c

- Revolution frequency of LHC is 11.24 kHz
- Bunch crossing rate (ZeroBias rate) depends on number of bunches in the machine
- e.g. For 2380 colliding bunches (2023)
  - ZeroBias rate = 26.8 MHz



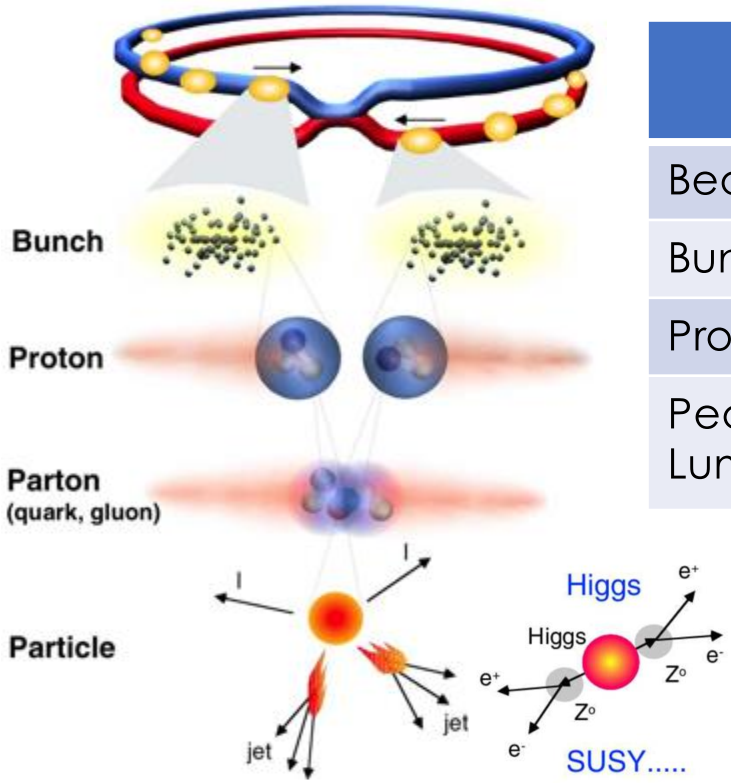
25ns_2390b_2378_1967_2106_180bpi_18inj_2INDIV.csv

Filling scheme with 36b trains optimized for all IPs

ATLAS	CMS	ALICE	LHCb
2378	2378	1967	2106

	Beam 1	Beam 2
Bunches	2390	2390
Non colliding	0	0
Injections	18	18

LHC Parameters



	Run-1 2010-2012	Run-2 2015-2018	Run-3* 2022-2026
Beam Energy	3.5-4.0 TeV	6.5 TeV	6.8 TeV
Bunches/Beam	1380	2556	2556
Protons/bunch	1.15×10^{11}	1.2×10^{11}	1.3×10^{11}
Peak Luminosity	7.7×10^{33} cm^{-2} s^{-1}	2.1×10^{34} cm^{-2} s^{-1}	2.1×10^{34} cm^{-2} s^{-1}

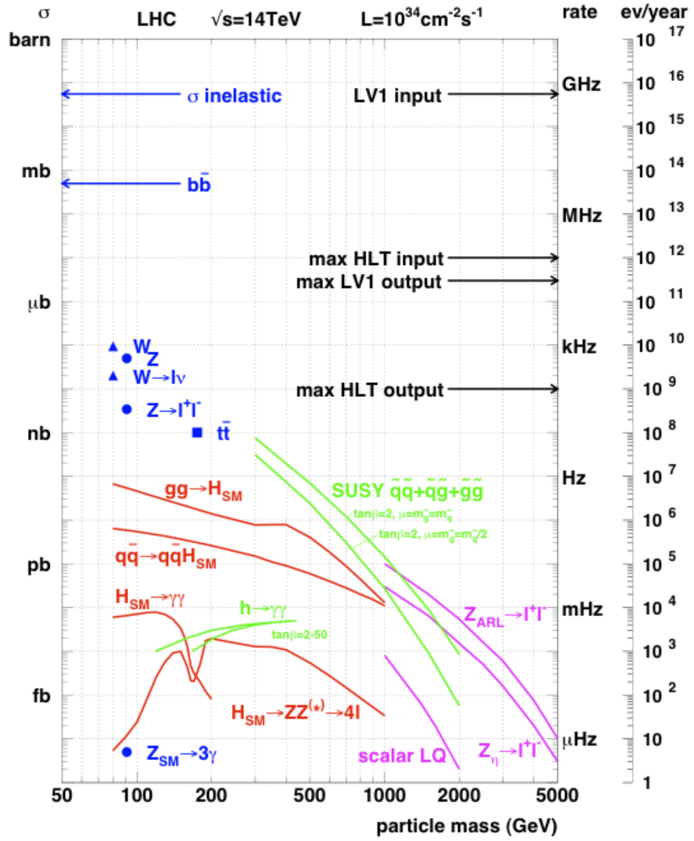
Crossing rate: ~40MHz (every 25ns)

Proton-Proton collision @ LHC

Production cross-sections for different physics processes span over many orders of magnitude

- Collision rate is dominated by non interesting physics
- Background discrimination is crucial

Total non-diffractive p-p cross section at LHC ($\sqrt{s}$ = 14 TeV) is **~80mb**

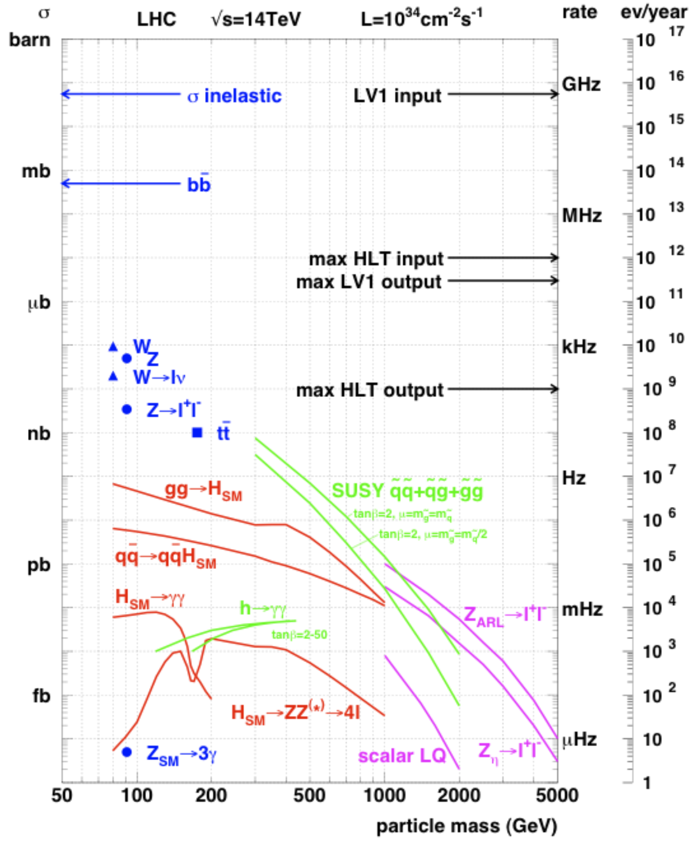


Proton-Proton collision @ LHC

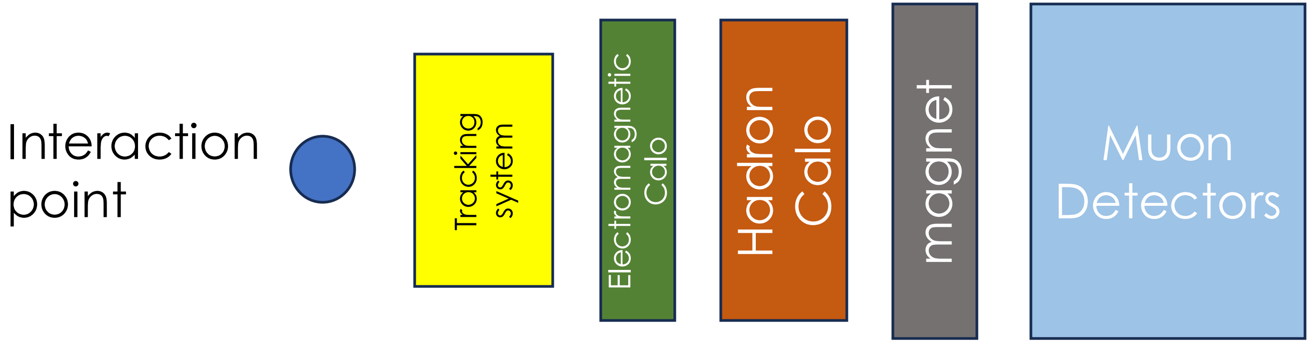
At Instantaneous Luminosity of:

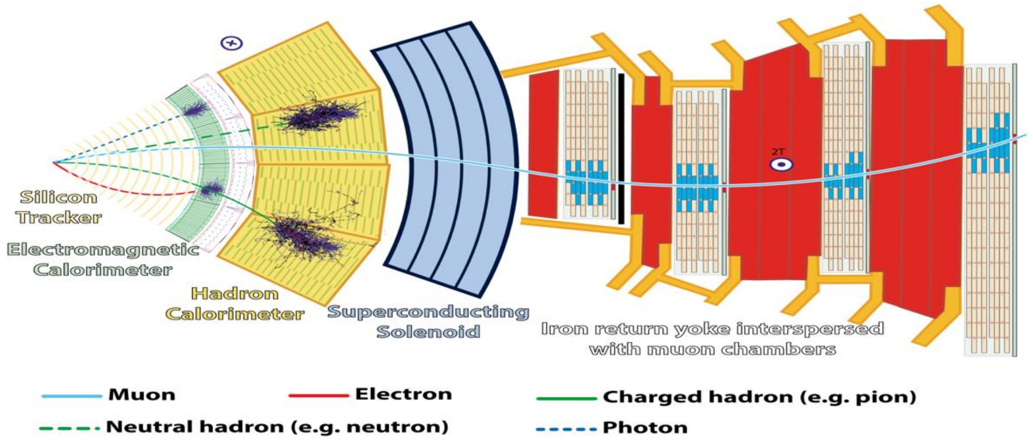
$\sim 2 \times 10^{34} cm^{-2}s^{-1}$

- 50 pp events/25 ns crossing
  - About 1 GHz input collision rate
- **EWK rate:** 1 kHz W/Z events
- **Top rate:** 10 Hz top events
- **Higgs rate:** < 10^4 detectable Higgs/year

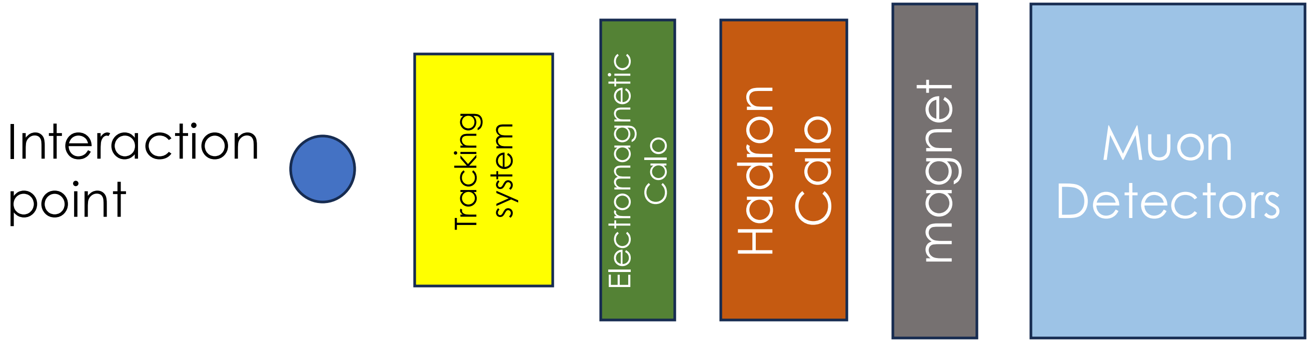


Particle Physics Detectors




Silicon Tracker
Electromagnetic Calorimeter
Hadron Calorimeter
Superconducting Solenoid
Iron return yoke interspersed with muon chambers
2T
Muon
Electron
Charged hadron (e.g. pion)
Neutral hadron (e.g. neutron)
Photon

Example: CMS Detectors



CMS DETECTOR

Total weight	: 14,000 tonnes
Overall diameter	: 15.0 m
Overall length	: 28.7 m
Magnetic field	: 3.8 T

STEEL RETURN YOKE
12,500 tonnes

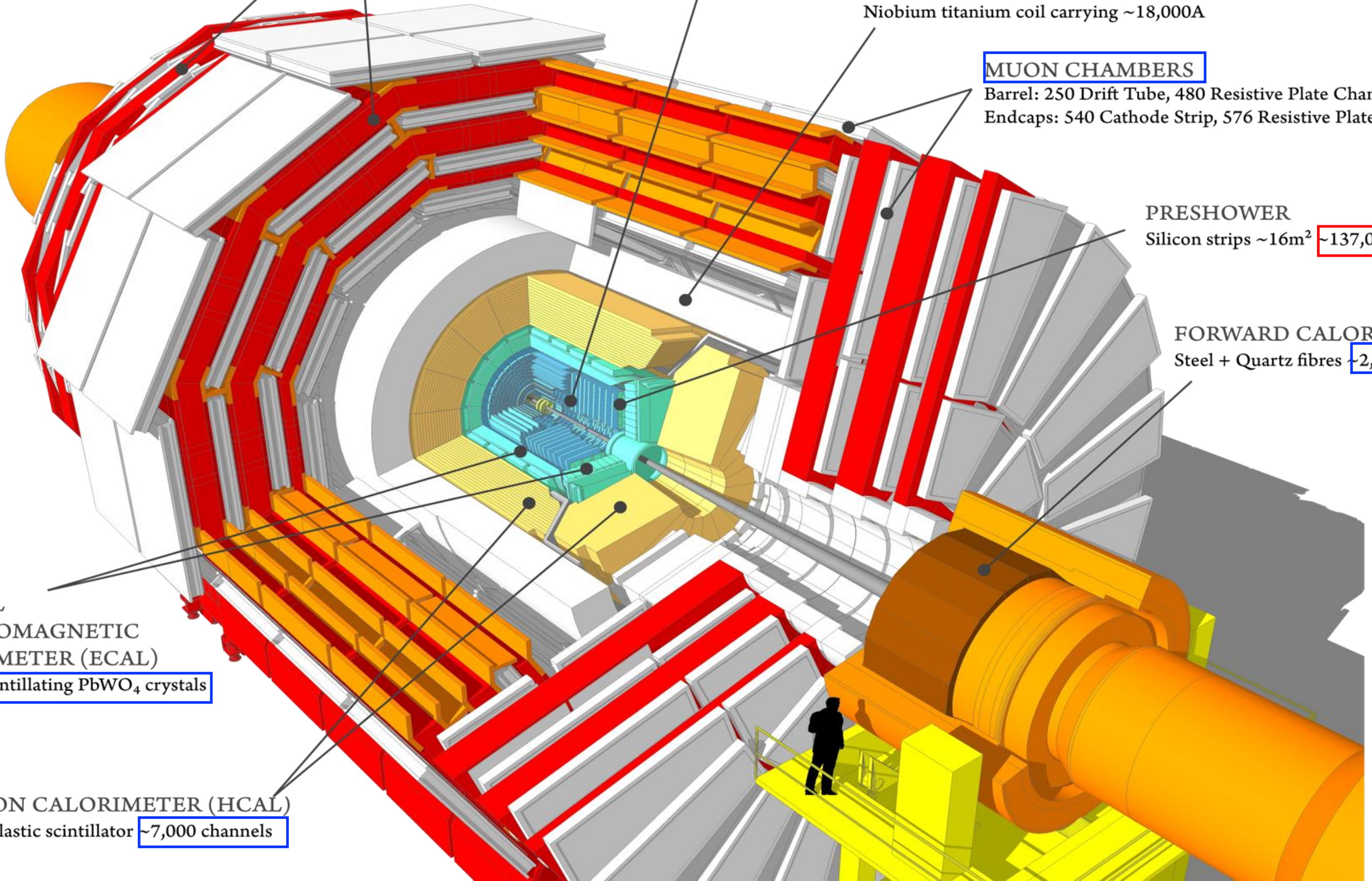
SILICON TRACKERS
Pixel (100x150 μm) ~1m^2 ~66M channels
Microstrips (80x180 μm) ~200m^2 ~9.6M channels

SUPERCONDUCTING SOLENOID
Niobium titanium coil carrying ~18,000A

MUON CHAMBERS
Barrel: 250 Drift Tube, 480 Resistive Plate Chambers
Endcaps: 540 Cathode Strip, 576 Resistive Plate Chambers

PRESHOWER
Silicon strips ~16m^2 ~137,000 channels

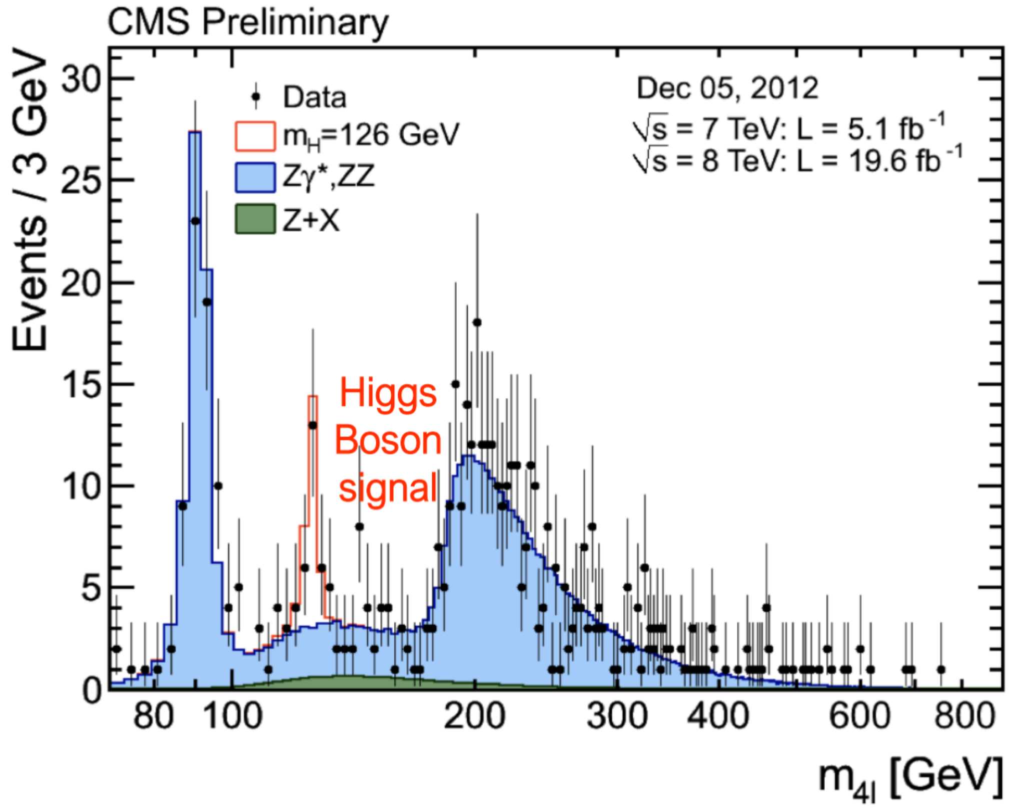
FORWARD CALORIMETER
Steel + Quartz fibres ~2,000 Channels



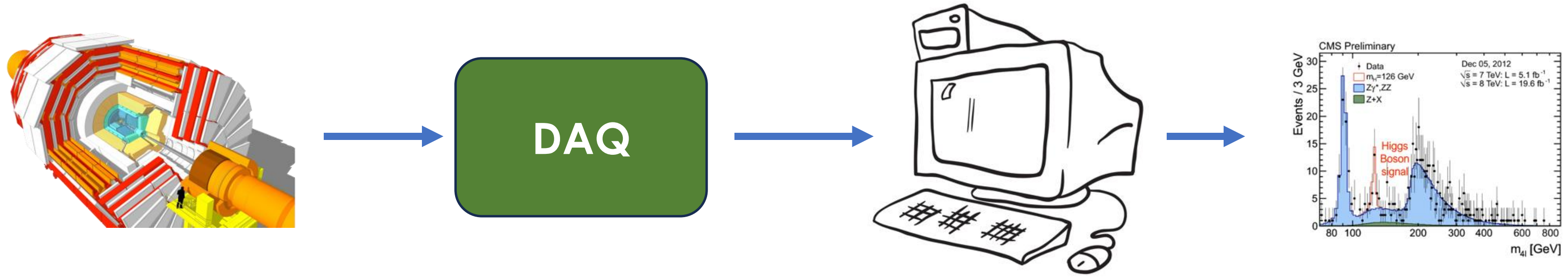
CRYSTAL ELECTROMAGNETIC CALORIMETER (ECAL)
~76,000 scintillating $PbWO_4$ crystals

HADRON CALORIMETER (HCAL)
Brass + Plastic scintillator ~7,000 channels

Goal


CMS Preliminary
Data
m_H=126 GeV
$Z\gamma^*$,ZZ
Z+X
Dec 05, 2012
$\sqrt{s}$ = 7 TeV: L = 5.1 fb^{-1}
$\sqrt{s}$ = 8 TeV: L = 19.6 fb^{-1}
Events / 3 GeV
m_{4l} [GeV]
Higgs
Boson
signal

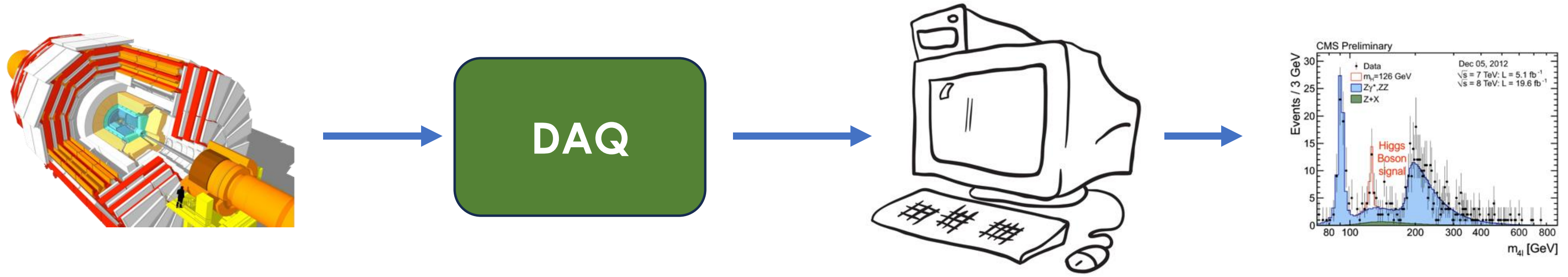
Reminder: DAQ



The Data Acquisition (DAQ) system collects the data from all the sub-detectors, converts the data in a suitable format and saves it to permanent storage

Question: Is that all?

Reminder: DAQ



The Data Acquisition (DAQ) system collects the data from all the sub-detectors, converts the data in a suitable format and saves it to permanent storage

Question: Do we need a trigger? If yes, why & where?

There is a problem…

- At an input rate of 40MHz
- Each raw event being 1-2MB

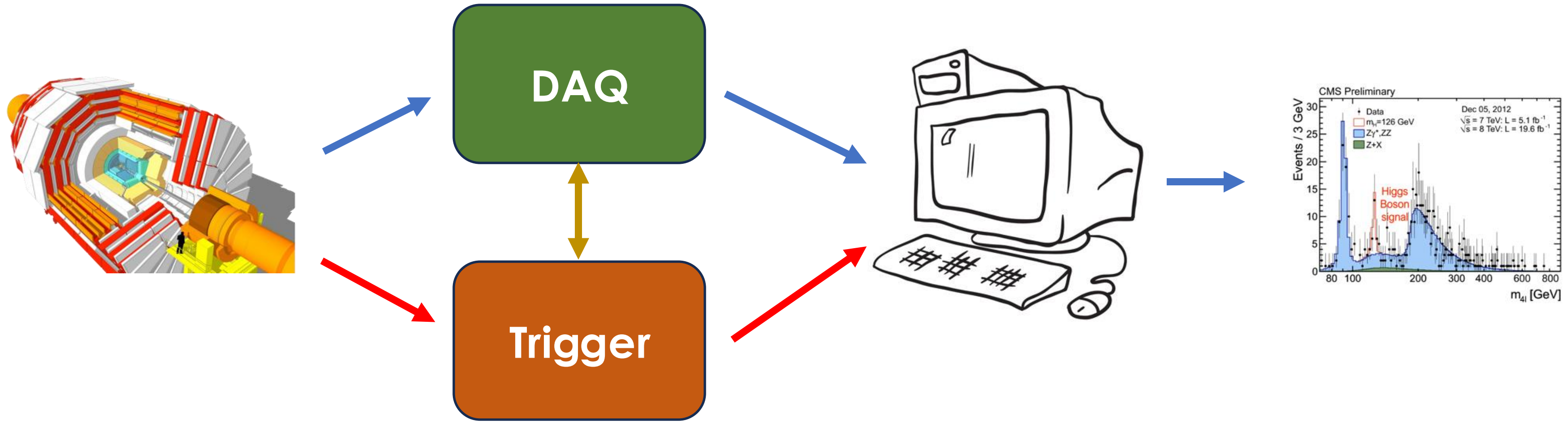
It is impossible to record data at 80 PB/s

There is a problem…

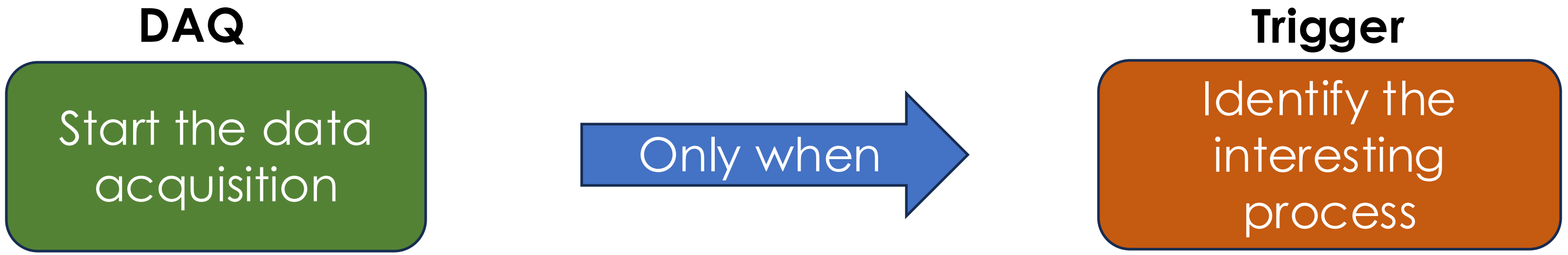
- At an input rate of 40MHz
- Each raw event being 1-2MB

It is impossible to record data at 80 PB/s

Solution ⇒ Be Selective ⇒ Add a trigger



Trigger concept

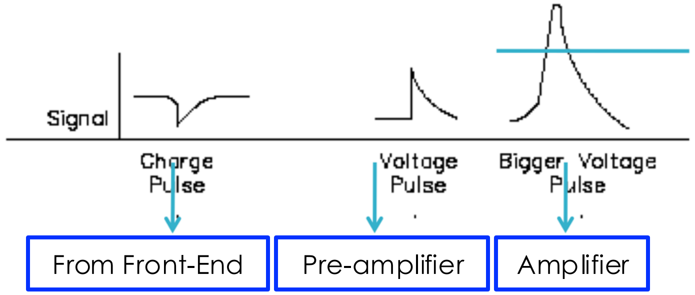


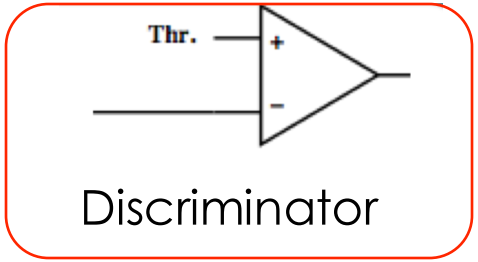
The role of the trigger is to make the **online selection** of particle collisions potentially containing interesting physics

- **What is 'Interesting'?:**
  - Define what is signal and what is background
- **What is the final affordable rate of the DAQ system?**
  - Define the maximum allowed rate
- **How fast the selection must be?**
  - Define the maximum allowed processing time

A Simple Trigger System

- **Data Input:** signals from front-end electronics



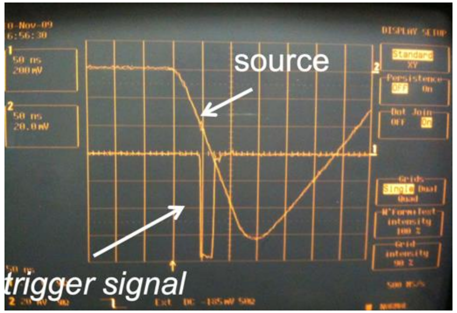


Accept

Reject

The simplest trigger: apply a threshold

- Look at the signal
- Put a threshold as low as possible

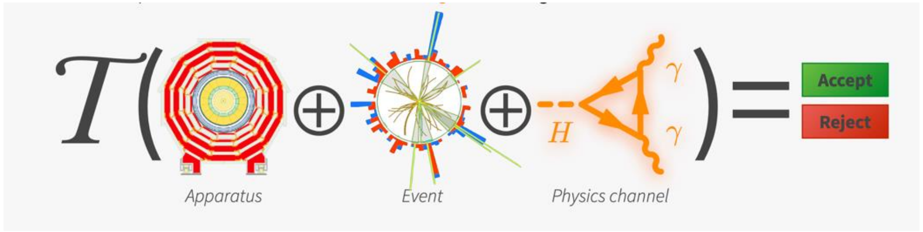


CMS Trigger System

Two level triggering

- *Level – 1 Trigger (L1T)*
  - Custom hardware using FPGAs
  - 40 MHz ➛ 100 kHz
- *High Level Trigger (HLT)*
  - Computing farm
  - 100 kHz ➛ 1kHz

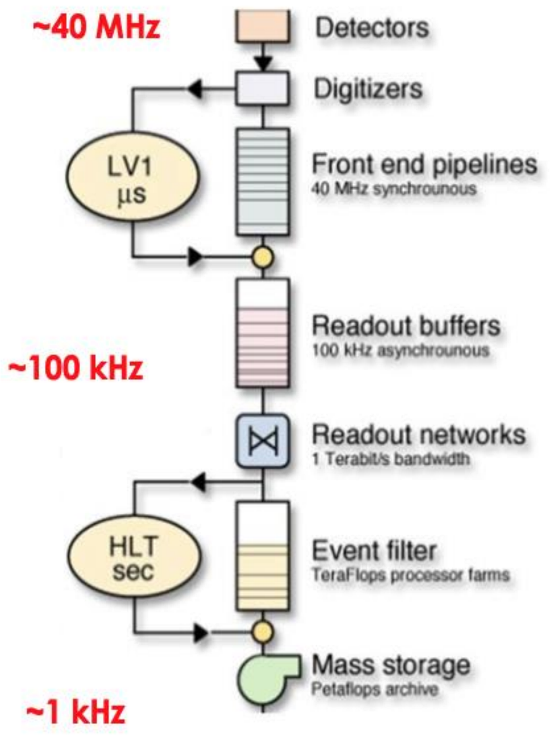
Experiment	# of levels
ALICE	4
ATLAS	3
LHCb	3
CMS	2



Level-1
Level-2
Level-3

Question: Why different levels?

CMS Trigger Architecture

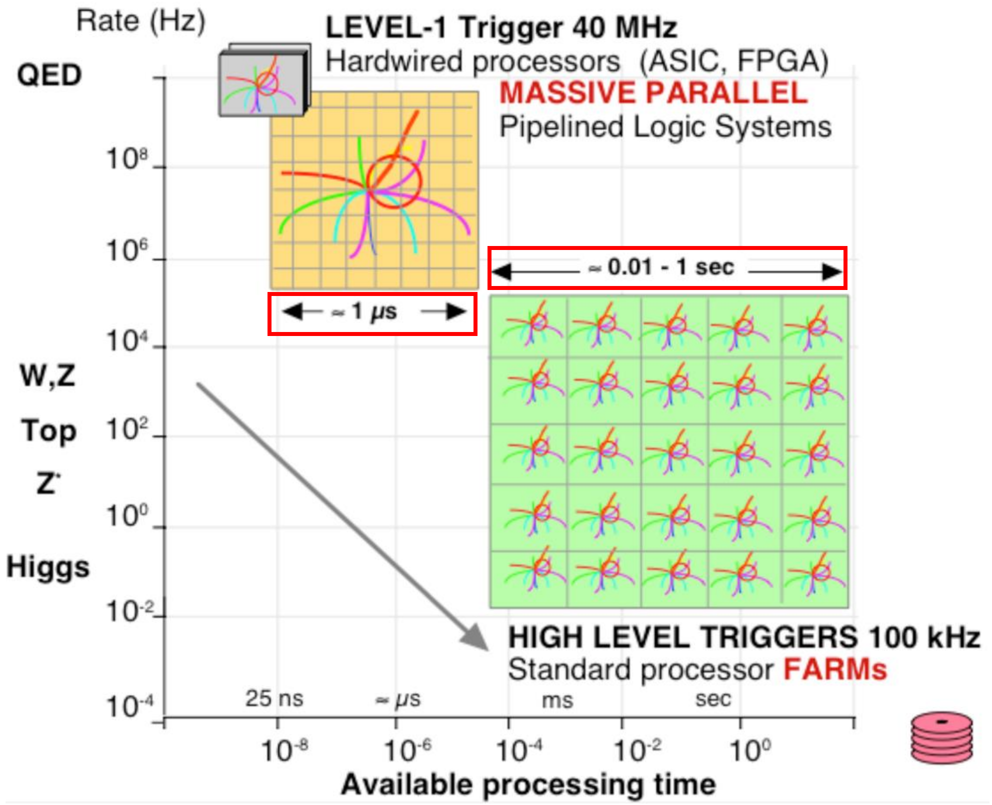


Data path split here:
Coarse (L1), raw (DAQ)

Data sitting in buffers, waiting for decision from L1

L1 latency sets the depth of buffers (and $$)

Data Processing to Trigger



“Interesting” Physics Signatures

Electroweak Symmetry Breaking Scale

- Higgs (125 GeV) studies and higgs sector characterization
- Quark, lepton Yukawa couplings to higgs

New physics at TeV scale to stabilize higgs sector

- Spectroscopy of new EWK produced resonances (SUSY or otherwise)
- Find dark matter candidate

Multi-TeV scale physics (loop effects)

- Indirect effects on flavor physics (mixing, FCNC, etc.)
- Lepton flavor violation

Planck scale physics

- Large extra dimensions to bring it closer to experiment
- New heavy bosons
- Blackhole production

How to Identify these “interesting” events?

Electroweak Symmetry Breaking Scale

- Higgs (125 GeV) studies and higgs sector characterization ← Low P_T γ, e, μ
- Quark, lepton Yukawa couplings to higgs ← Low P_T B, τ jets

New physics at TeV scale to stabilize higgs sector

- Spectroscopy of new EWK produced resonances (SUSY or otherwise) ← Multiple low P_T objects
- Find dark matter candidate ← Missing E_T

Multi-TeV scale physics (loop effects)

- Indirect effects on flavor physics (mixing, FCNC, etc.) ← ~ Dedicated triggers (CMS) or experiment (LHCB)
- Lepton flavor violation ← Low P_T leptons

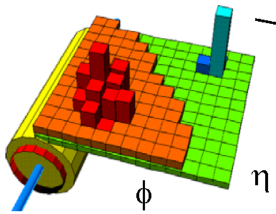
Planck scale physics

- Large extra dimensions to bring it closer to experiment ← High P_T leptons and photons

  Multi particle and jet events
- New heavy bosons
- Blackhole production

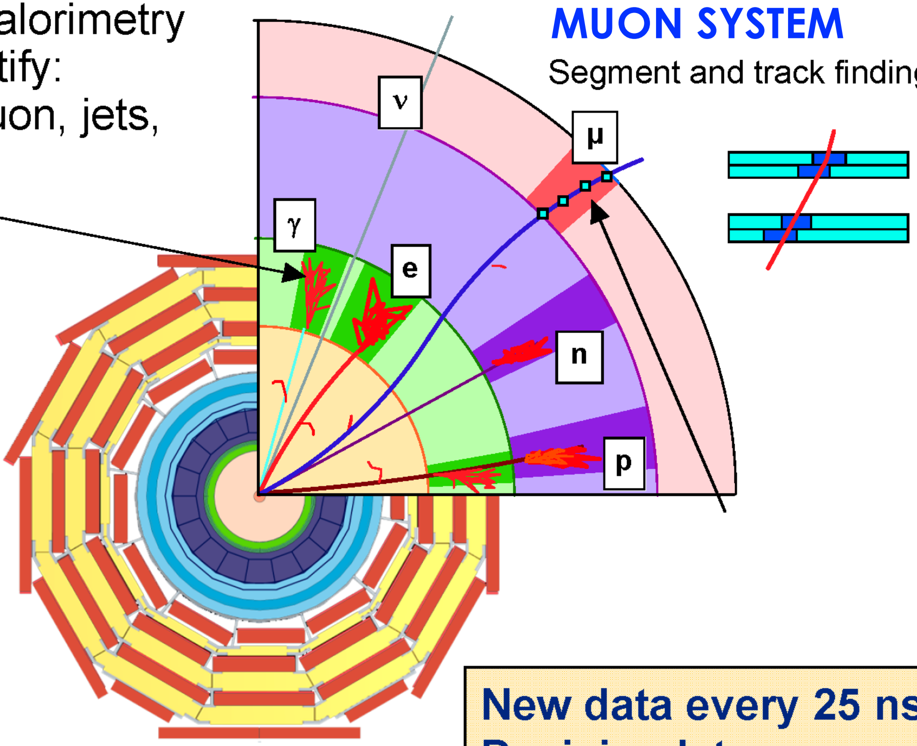
Input to CMS level-1 Trigger

Use prompt data (calorimetry and muons) to identify:
High p_t electron, muon, jets, missing E_T



CALORIMETERS

Cluster finding and energy deposition evaluation



MUON SYSTEM

Segment and track finding

New data every 25 ns
Decision latency ~ μs

NO TRACKER

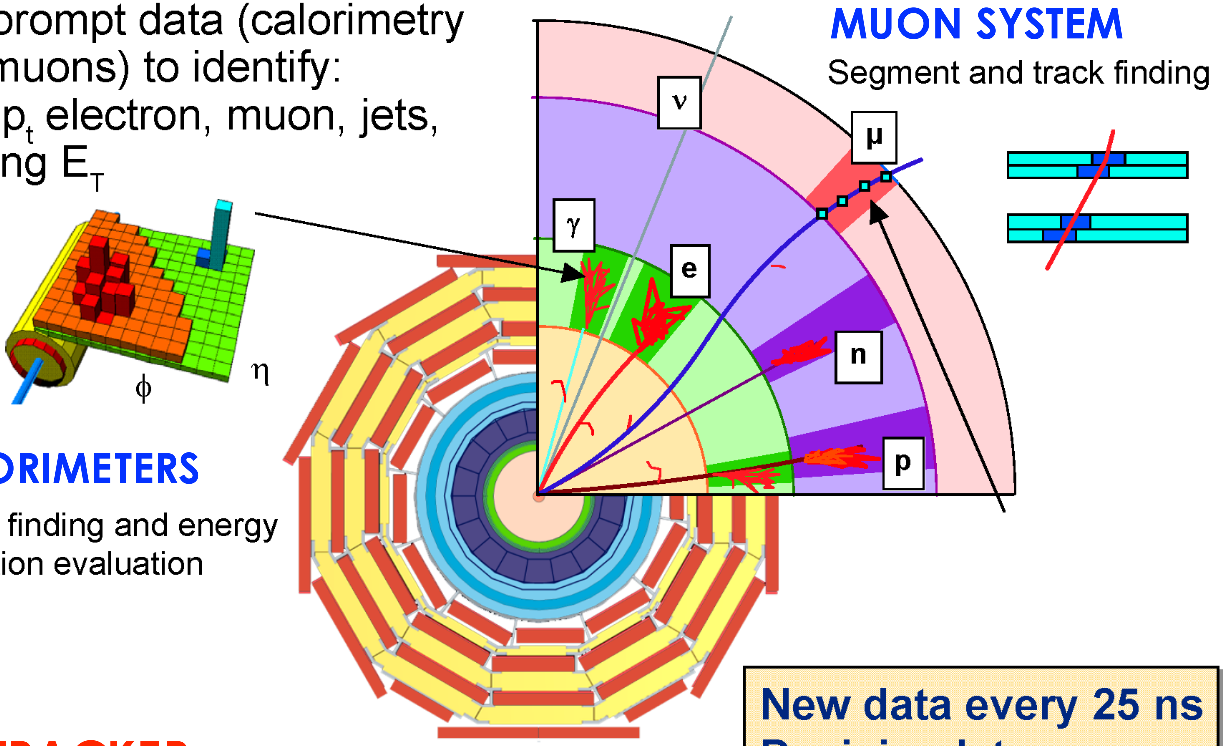
Question:
Why not Tracker?

What is a latency?

Why μs?

Input to CMS level-1 Trigger

Use prompt data (calorimetry and muons) to identify:
High p_t electron, muon, jets, missing E_T



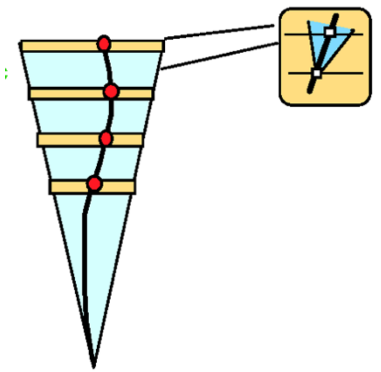
CALORIMETERS

Cluster finding and energy deposition evaluation

MUON SYSTEM

Segment and track finding

New data every 25 ns
Decision latency ~ μs



Pattern recognition much faster/easier

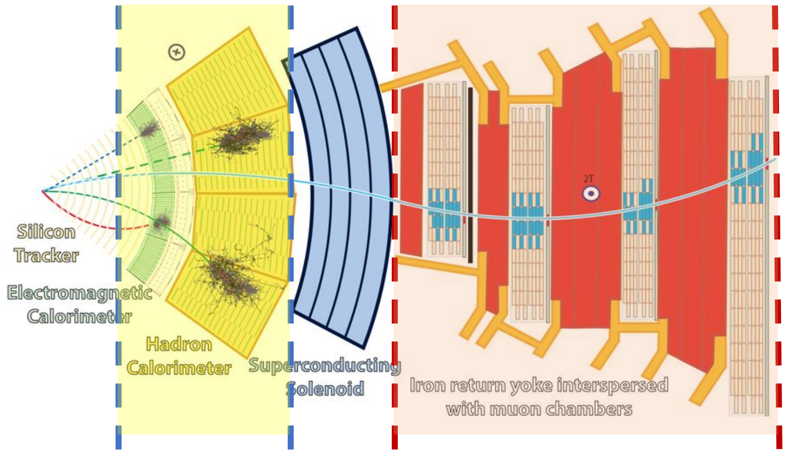
NO TRACKER

Input to CMS level-1 Trigger

Level-1 trigger receives data with coarse granularity from

- Calorimeters (ECAL, HCAL, HF)
- Muon systems (CSC, DT, RPC, GEM)

Collision data are buffered locally for < 4μs

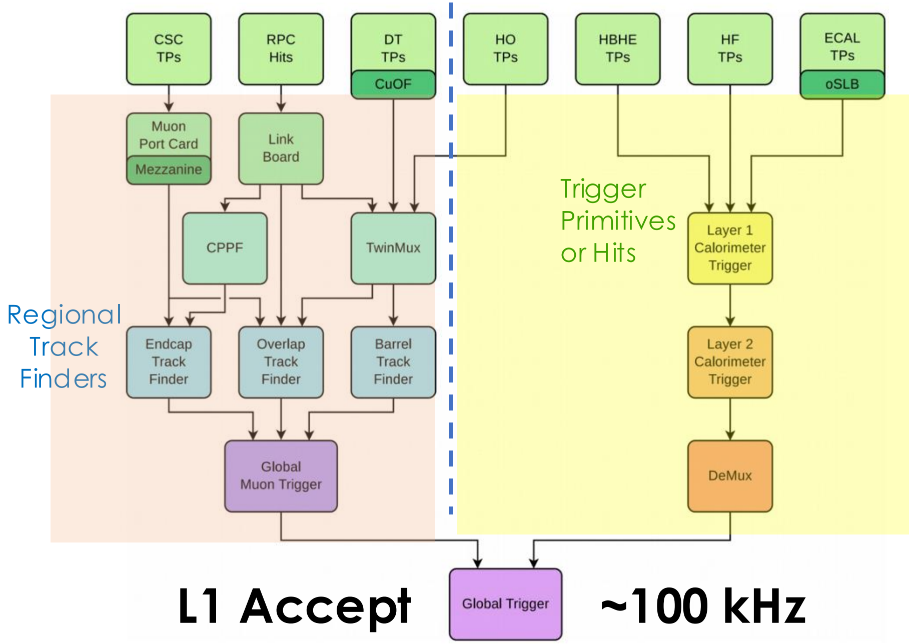

Silicon Tracker
Electromagnetic Calorimeter
Hadron Calorimeter
Superconducting Solenoid
Iron return yoke interspersed with muon chambers
2T

Calorimeters   Muon Systems

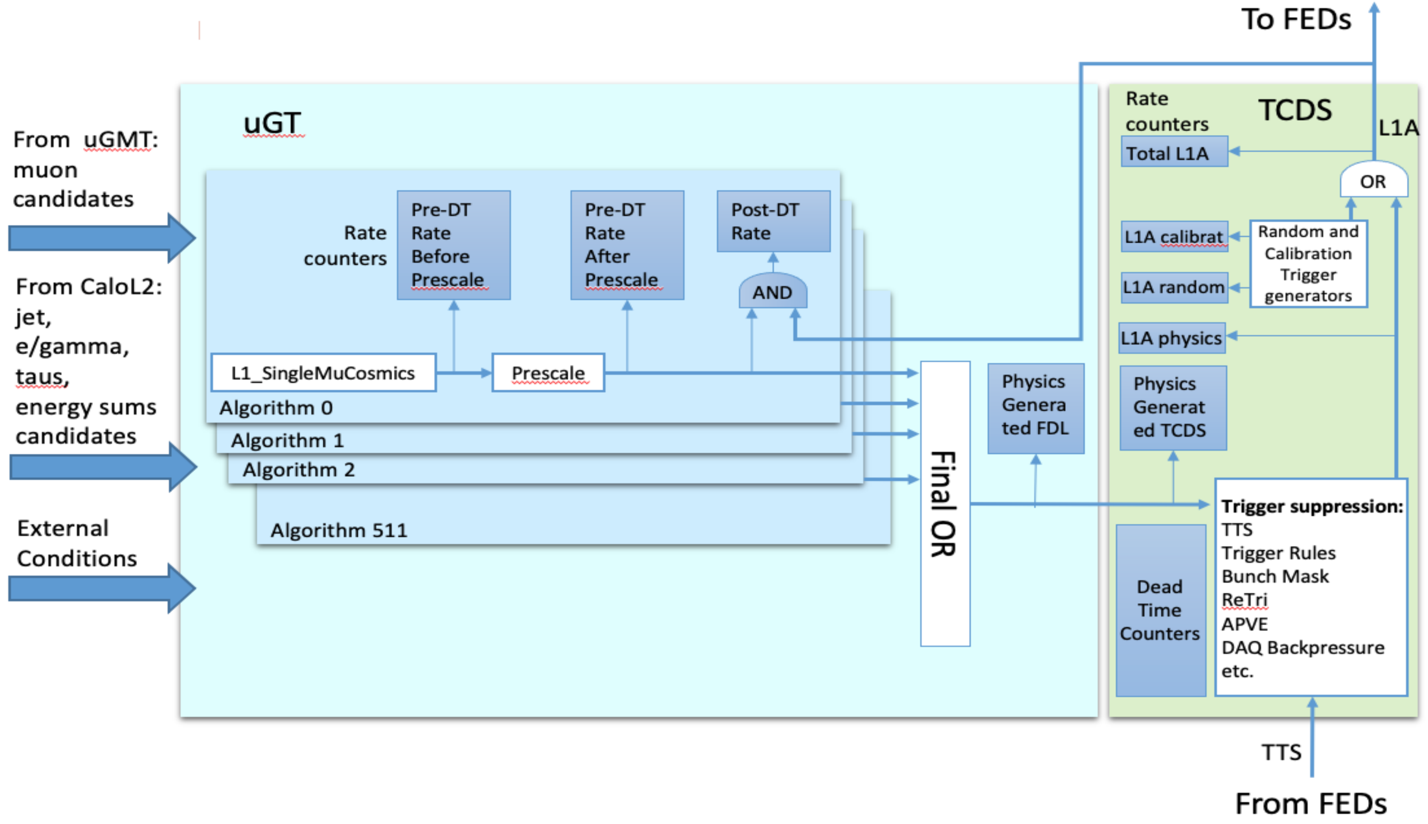
L1 Trigger is implemented in hardware

Uses field programmable gate arrays (FPGAs)

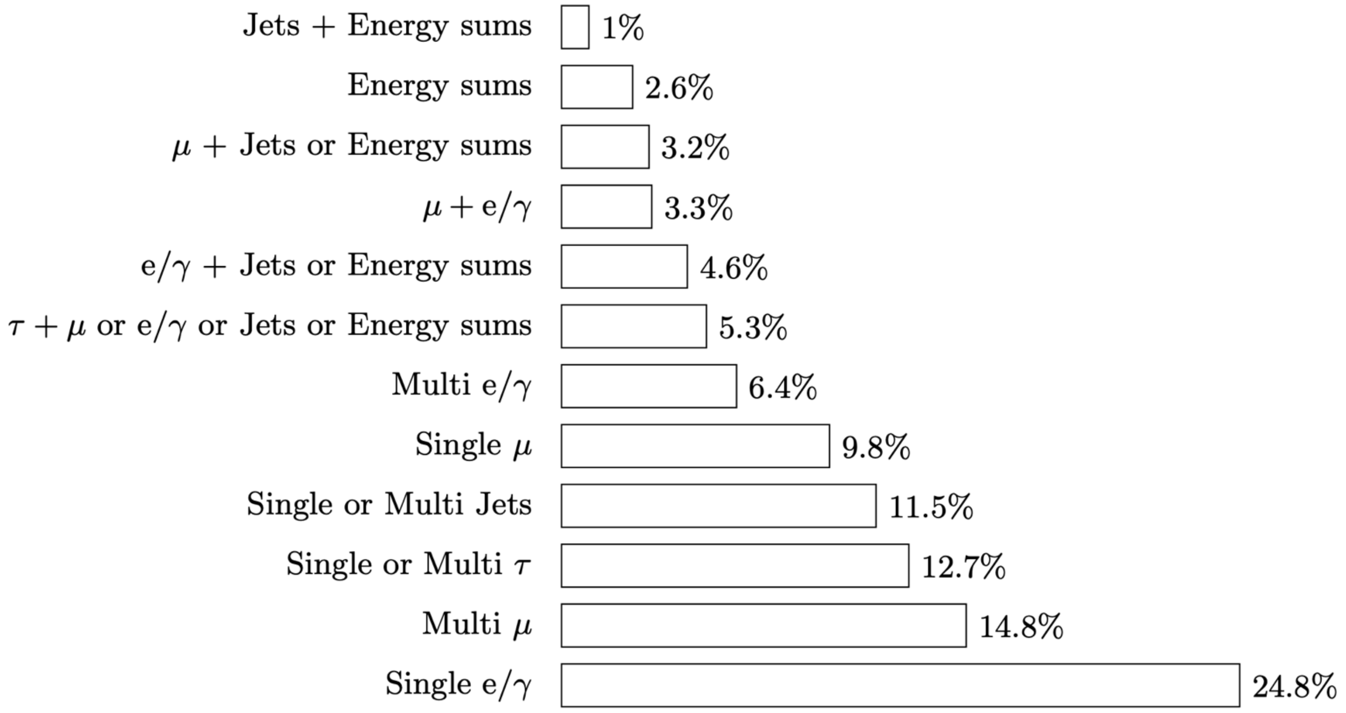
Operates synchronously to the LHC clock (40 MHz)



Trigger Final Decisions



What all we keep?



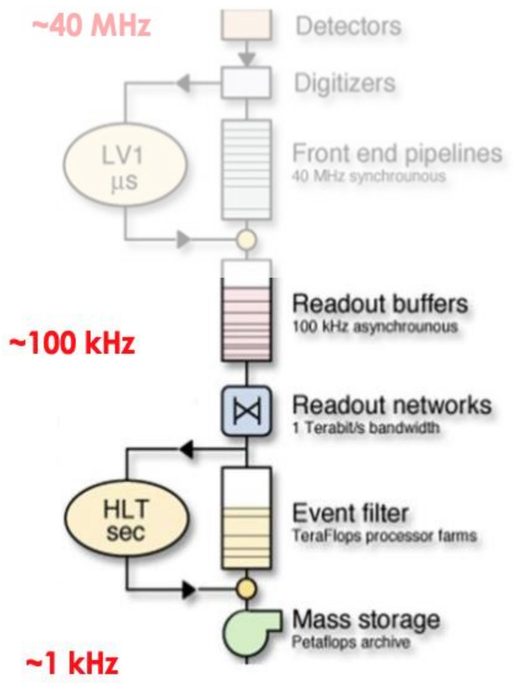
High Level Trigger

- Implemented using generic processors **(CPUs/GPUs)**
- Muon Systems, Calorimeters **and Tracker**
- Increase in number of Trigger, algorithms, selections and complexity
- Event Filtering, **Selections are made sequentially: When an event fails** a given selection criteria then the **processing stop**s in order to allow the node to be used by a **new event**
- Data accepted by the HLT are recorded for offline physics analysis
- HLT contains hundred of paths, each of which is seeded by one or more trigger at L1. **Example:**

HLT_Mu17_TkMu8

Non isolated muon with pT > 17 GeV

Muon with pT > 8 GeV from tracker

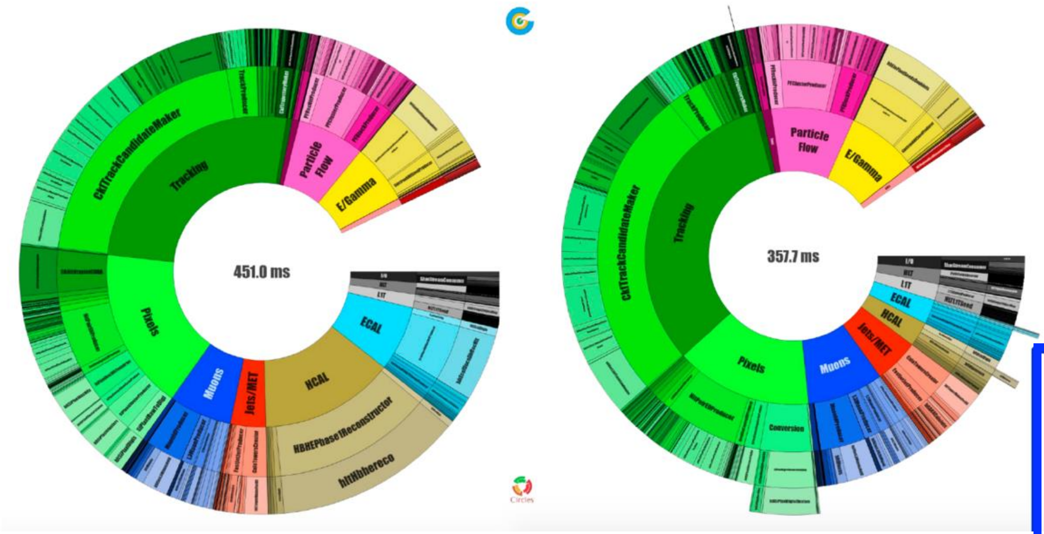


GPU Acceleration @ CMS HLT

21% processing time reduction

CPU

CPU + GPU



The pie-chart shows the distribution of CPU time in different instances of CMSSW modules (outermost ring), their corresponding C++ class (one level inner), grouped by physics object or detector (innermost ring). The empty slice indicates the time spent outside of the individual algorithms.

The time spent in the conversion of GPU-friendly *Structure of Arrays* data formats to legacy data formats is indicated by "Conversion" in the extra internal ring.

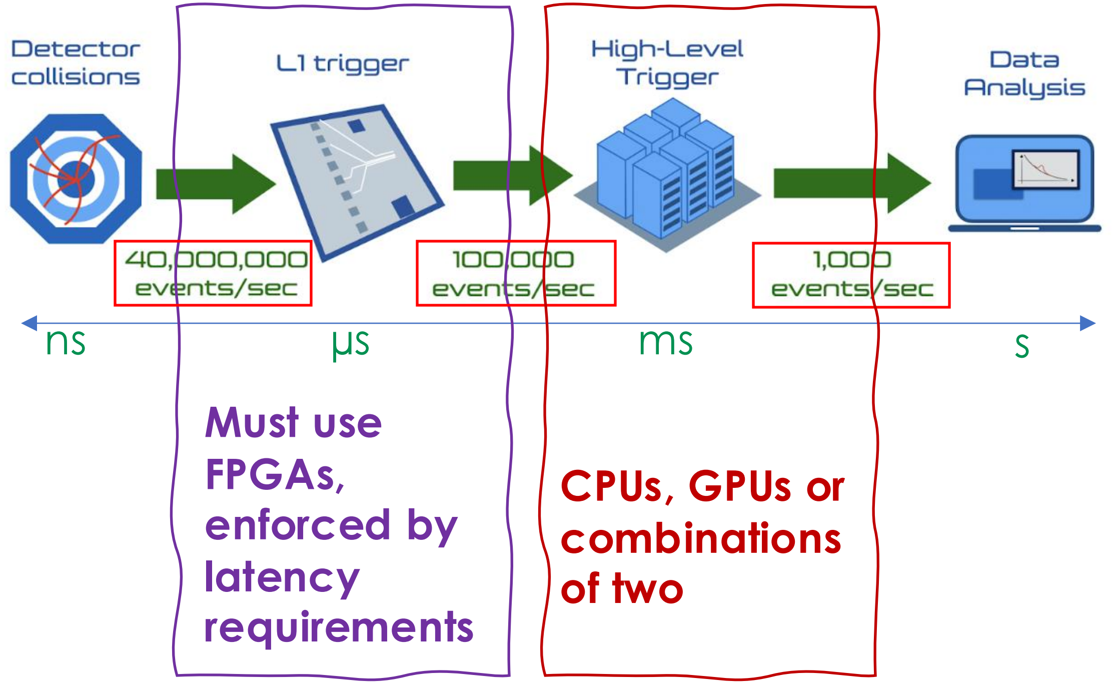
The timing is measured on pileup 50 events from Run2018D on a full HLT node (2x Intel Skylake Gold 6130) with HT enabled, running 16 jobs in parallel, with 4 threads each - equipped with an NVIDIA T4 GPU.

Using the GPU to accelerate:

- pixel local reconstruction, track and vertex reconstruction
- HCAL local reco (MAHI)
- ECAL unpacking and local reconstruction (multifit)

reduces the CPU usage by 21%, increasing the throughput by 26%.

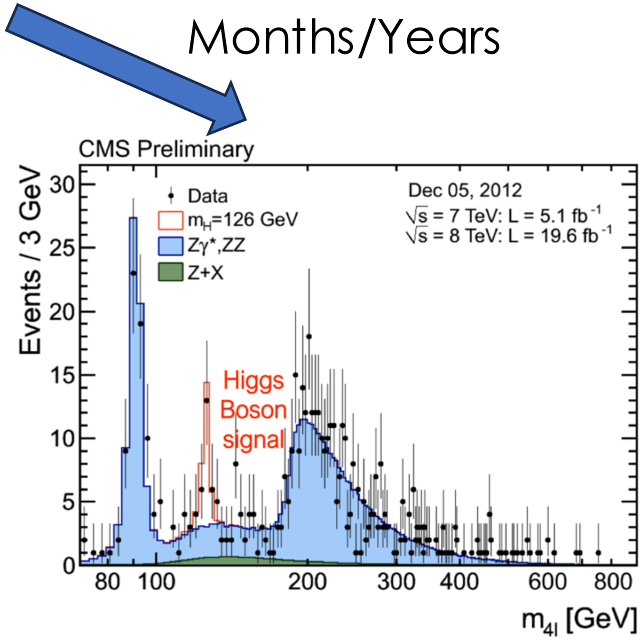
All set to do physics analyses

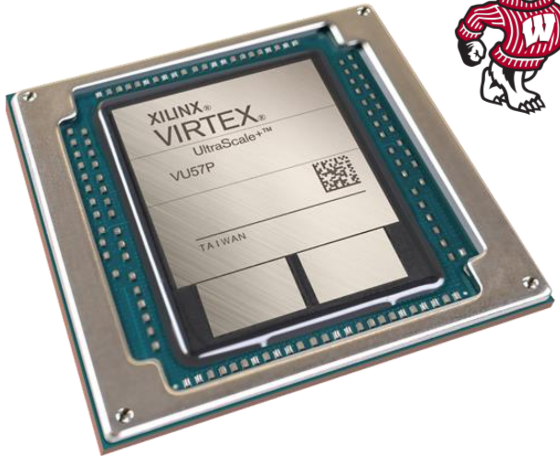


Must use FPGAs, enforced by latency requirements

CPUs, GPUs or combinations of two

Months/Years





Lets discuss about FPGAs

FPGA: **F**ield **P**rogrammable **G**ate **A**rray

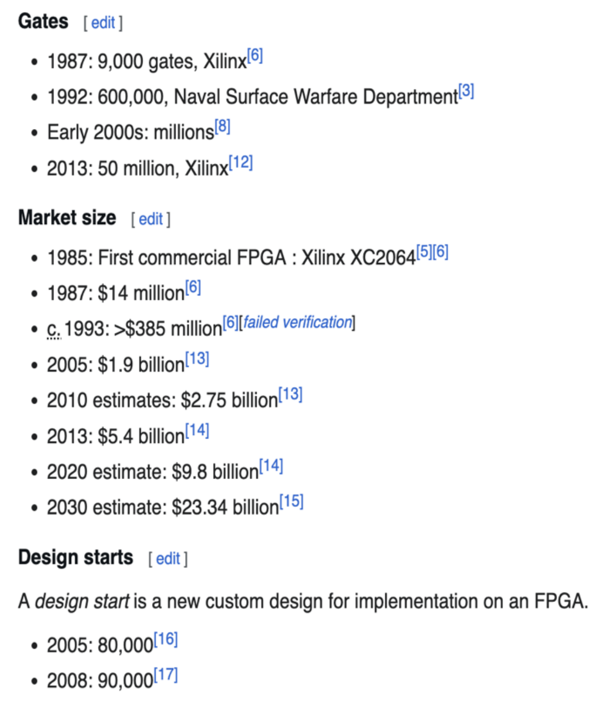
Xilinx Field Programmable Gate Array

Xilinx: All Programmable

Software Defined, Hardware Optimized

You may know Xilinx because we invented the FPGA. Or maybe you know us because we turned the semiconductor world upside down and created the fabless model. With over 3500 patents and more than 60 industry firsts, we continue to pioneer new programmable technology putting our customers first. Today Xilinx's portfolio combines All Programmable devices in the categories of FPGAs, SoCs, and 3DICs, as well as All Programming models, including software-defined development environments. Our products are enabling smart, connected, and differentiated applications driven by 5G Wireless, Embedded Vision, Industrial IoT, and Cloud Computing.

First FPGA invented by Xilinx Inc. in 1985



Gates [edit]

- 1987: 9,000 gates, Xilinx[6]
- 1992: 600,000, Naval Surface Warfare Department[3]
- Early 2000s: millions[8]
- 2013: 50 million, Xilinx[12]

Market size [edit]

- 1985: First commercial FPGA : Xilinx XC2064[5][6]
- 1987: $14 million[6]
- c. 1993: >$385 million[6][*failed verification*]
- 2005: $1.9 billion[13]
- 2010 estimates: $2.75 billion[13]
- 2013: $5.4 billion[14]
- 2020 estimate: $9.8 billion[14]
- 2030 estimate: $23.34 billion[15]

Design starts [edit]

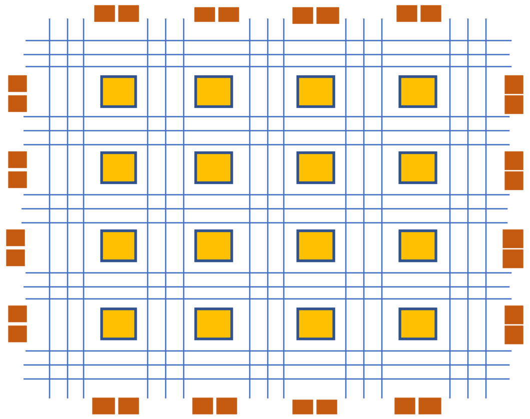
A *design start* is a new custom design for implementation on an FPGA.

- 2005: 80,000[16]
- 2008: 90,000[17]

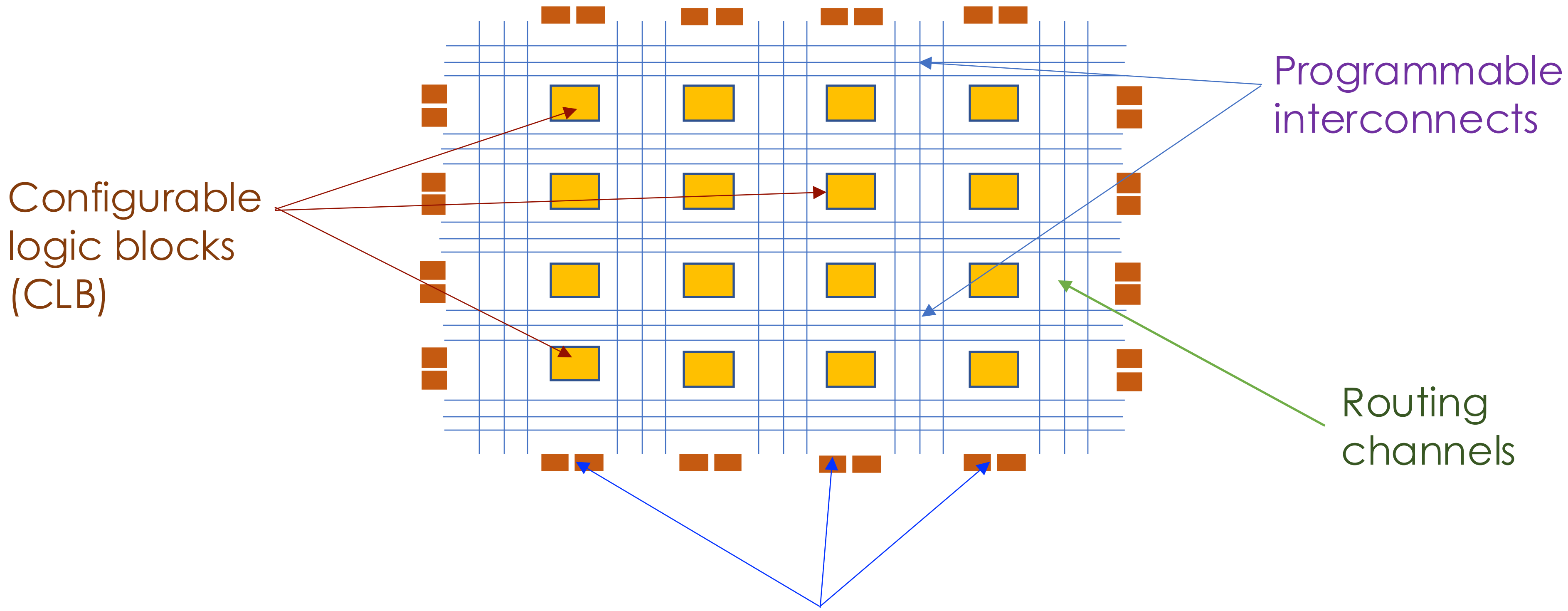
Source: *https://en.wikipedia.org/wiki/Field-programmable_gate_array*

FPGAs:

- Programmable hardware whose sub-component configuration can be changed even after fabrication: ***"field-programmable"***
- Has 2D array of logic gates in its architecture: ***"Gate Array"***
- A silicon '**breadboard**' of configurable logic gates, memories, transceivers, Digital Signal Processors (DSPs), registers (flip flops)
- FPGA industry sprouted from **programmable readonly memory (PROM)** and **programmable logic devices (PLDs)**



FPGA Architecture



Input/output blocks

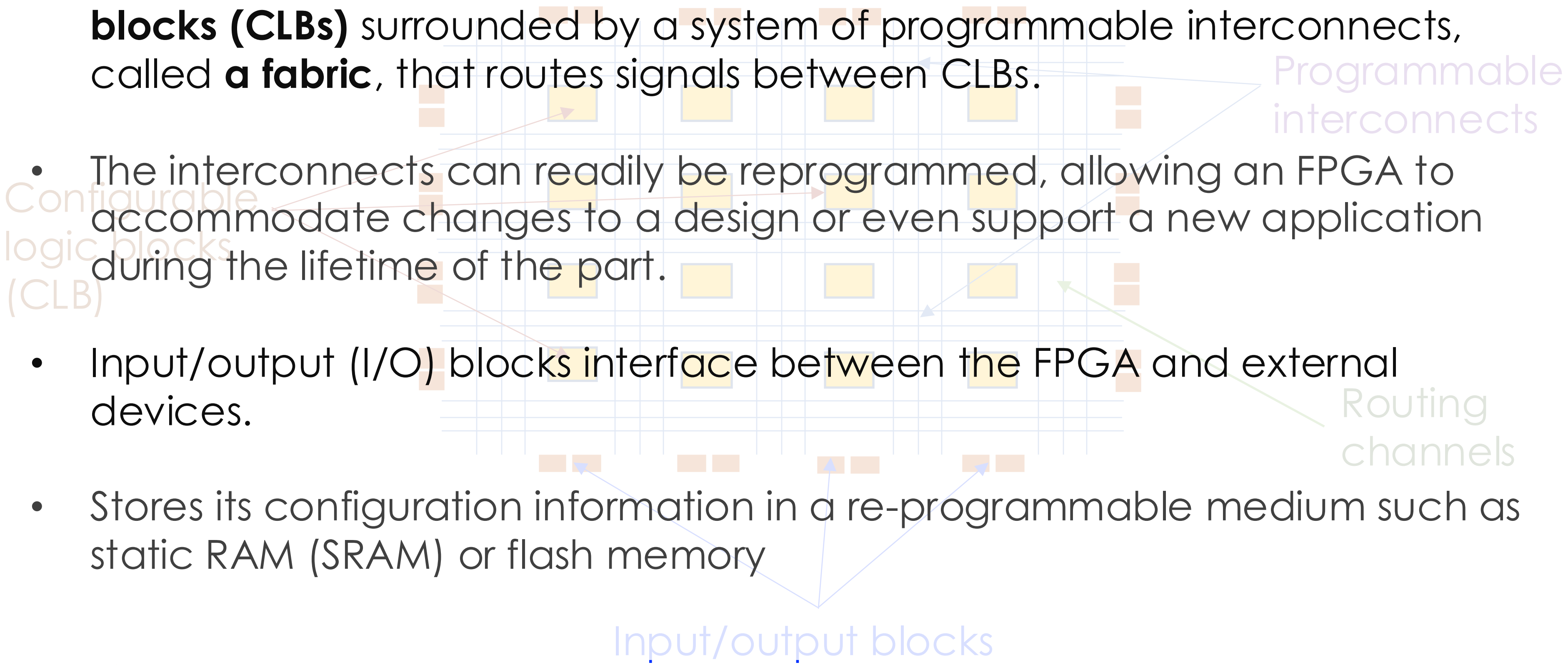
FPGA Architecture

- Contains thousands of fundamental elements called **configurable logic blocks (CLBs)** surrounded by a system of programmable interconnects, called **a fabric**, that routes signals between CLBs.

- The interconnects can readily be reprogrammed, allowing an FPGA to accommodate changes to a design or even support a new application during the lifetime of the part.

- Input/output (I/O) blocks interface between the FPGA and external devices.

- Stores its configuration information in a re-programmable medium such as static RAM (SRAM) or flash memory



FPGA Components

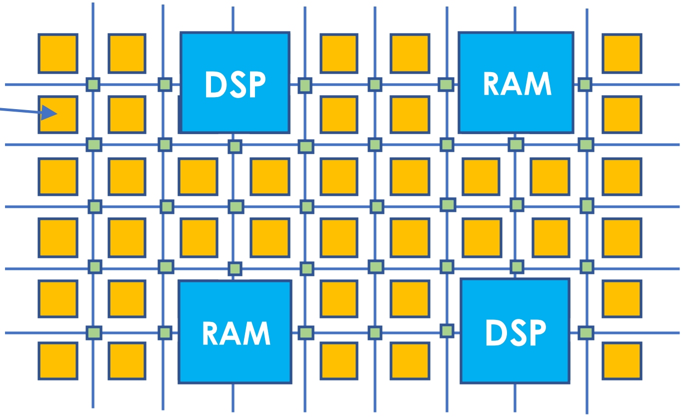
The basic structure of an FPGA is composed of:

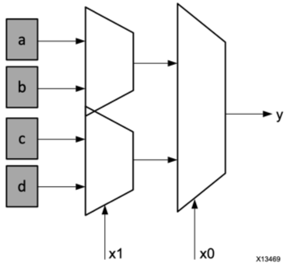
- Look-up table (LUT)
- Flip-Flop (FF)
- Slices and CLBs
- Block Memory (BRAM)
- DSP Blocks
- Interconnect and routing resources: Wires & Input/Output (I/O) pads

FPGA Components: LUT

LUTs or logic cells:

- Basic building block of FPGA used for implementing combinational logic
- Capable of performing any arbitrary functions on small bitwidth inputs (N), generally N ≤ 6
- Memory location accessed by LUTs: 2^N
- Example: a **4-input LUT** can implement any Boolean function with 4 variables by storing 16 (2^4) output values
- It can be used as both a function compute engine and a data storage element





Functional Representation of a LUT as Collection of Memory Cells

FPGA Components: Flip Flops

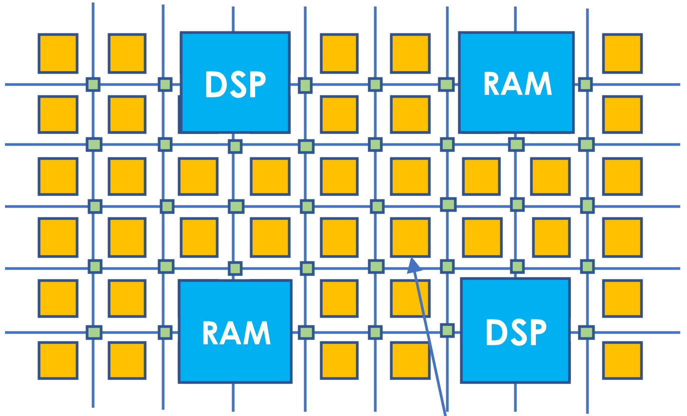
Flip-Flops:

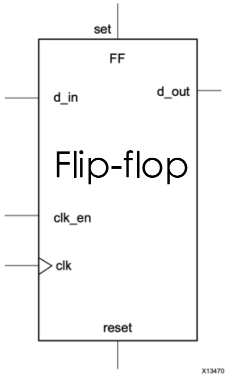
- Basic storage unit within the FPGA fabric
- Circuit that can store and recall a single bit of information. **Used for sequential logic**.

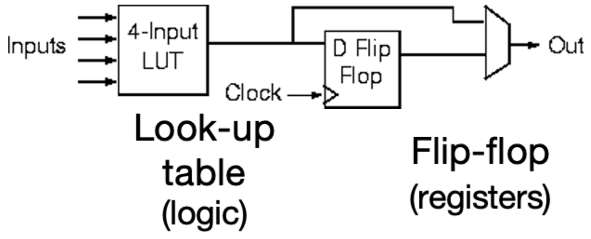
- Always paired with a LUT to assist in logic pipelining and data storage

- **Operation:** value at the data input port is latched and passed to the output on every pulse of the clock

- Data is passed only when clock and clock enable = 1



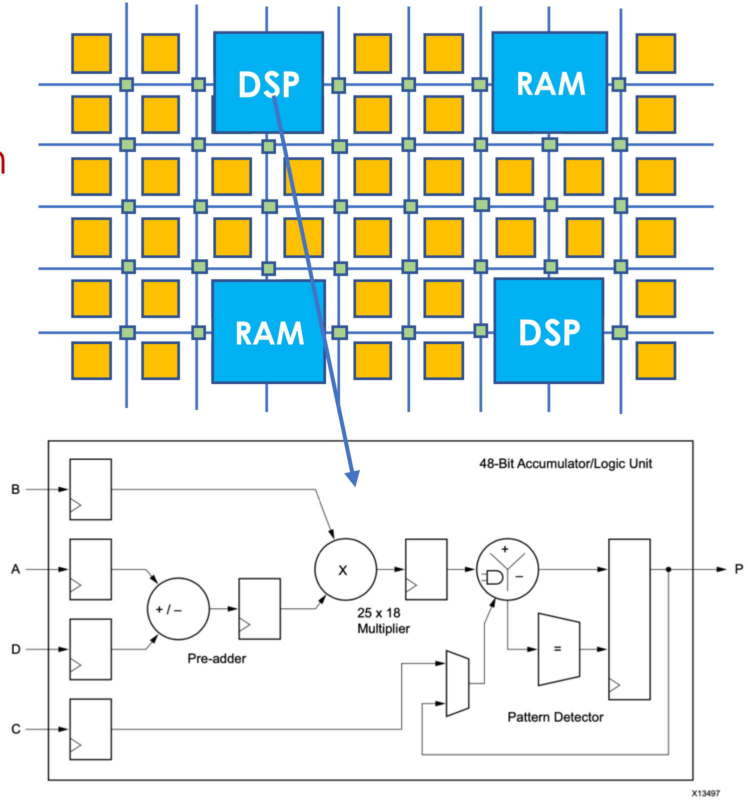




FPGA Components: DSP

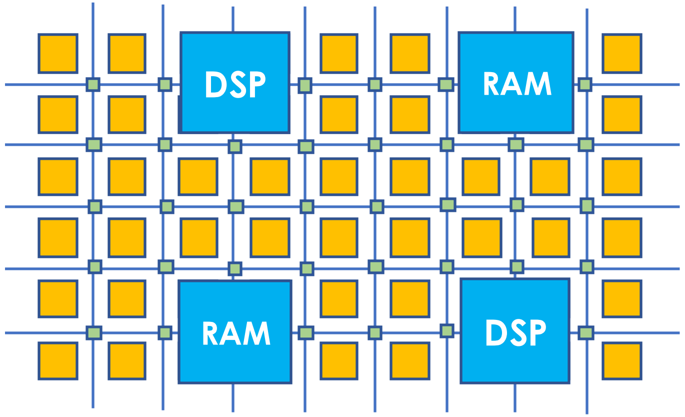
Digital Signal Processor Block:

- Most complex computational block available in a FPGA
- Arithmetic Logic unit: specialized unit for multiplication and arithmetic
  - Eg: $p = a \times (b + d) + c$
- Faster and more efficient than using LUTs for these types of operations
- Often most scarce in available resources



FPGA Components: Storage elements

BRAMs (Block RAM)

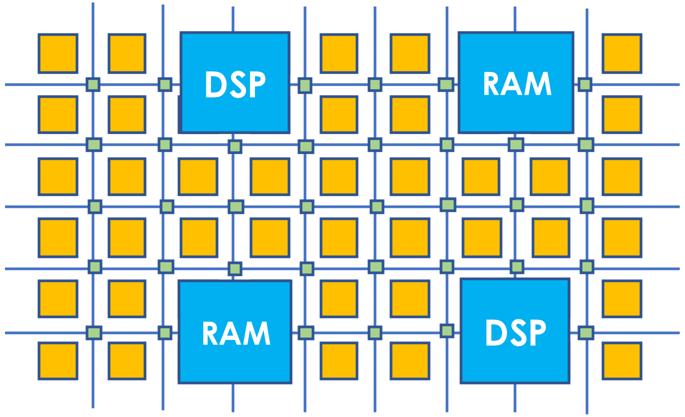


- Embedded memory elements that can be used as Random-access-memory
- BRAM is a dual-port RAM module instantiated to provide on-chip storage for a relatively large set of data
  - can hold either 18 k or 36 k bits
- Useful for **low latency** & **high bandwidth access** (data buffering, complex algos)
- BRAMs can implement either a RAM or a ROM. The only difference is when the data is written to the storage element.

FPGA Components: Storage elements

LUTs as storage element:

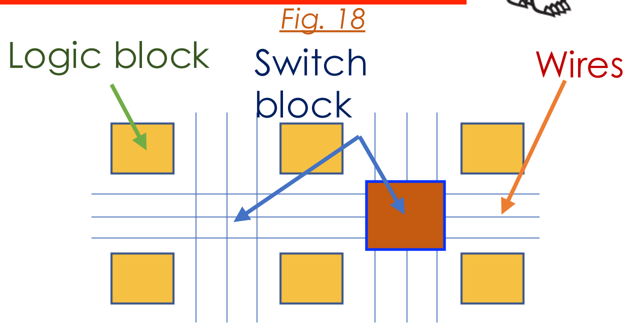
- They can be used as 64-b memories due to its structural flexibility
- Commonly referred to as distributed memories

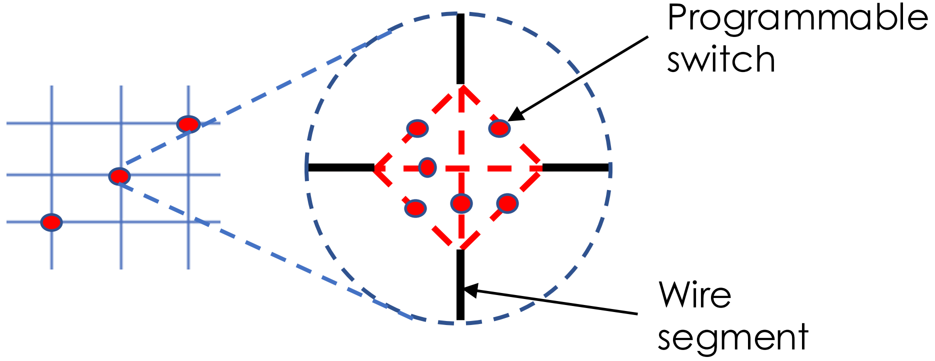


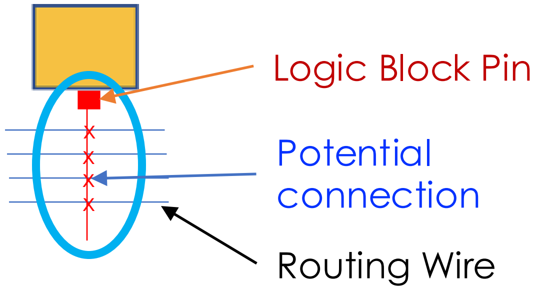
- Fastest kind of memory available on the FPGA device, because it can be instantiated in any part of the fabric that improves the performance of the implemented circuit
- Memories using BRAMs more efficient than using LUTs

FPGA Components: Routing

- Between rows and columns of logic blocks are wiring channels
- These are programmable – a logic block pin can be connected to one of many wiring tracks through programmable switch
- Xilinx FPGA have dedicated switch block circuits for routing (flexible)
- Each wiring segment can be connected in one of many ways







The main advantage and attraction of FPGA comes from the programmable interconnect – more so than the programmable logic.

FPGA Components: I/O

There are specialised blocks for I/O

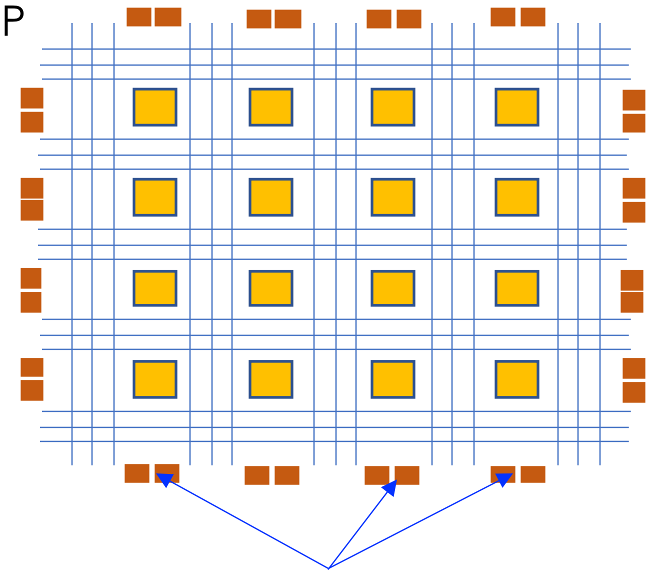
- Making FPGAs popular in embedded systems and HEP triggers

High speed transceivers

- with Tb/s total bandwidth PCIe
- (Multi) Gigabit Ethernet
- Infiniband

Support highly parallel algorithm implementations

Low power per Operation (relative to CPU/GPU)



Programming FPGA

- Programming an FPGA requires **Firmware** to be written and synthesized into a **"bit file"** to load into the chip

- Languages used to write the logic implementation:
  - Hardware Description Languages (HDLs)
    - Verilog
    - VHDL (VHSIC Hardware Description Language)
    - System Verilog
  - High-Level Synthesis (HLS) Languages
    - Code written in C/C++ is converted to RTL (Verilog/VHDL)
  - OpenCL

FPGA Parallelism

Program execution on a Processor

A processor executes a program as a **sequence of instructions**

- Translated into useful computation for a software application
- Compiler transforms the C/C++ into assemble language

$z = a + b;$ $\Rightarrow$ `ADD $R1,$R2,$R3`

- The assemble code defines the addition operation to compute the value of z in terms of the internal registers of a processor
- The complete assembly program to compute the value of z is as follows:

```
LD      a, $R1
LD      b, $R2
ADD     $R1,$R2,$R3
ST      $R3, c
```

- **Even a simple operation, such as the addition of two values, results in multiple assembly instructions**

Program execution on a Processor

- **Depending on the location of a and b, the LD operations take a different number of clock cycles to complete:**
  - Processor cache : few 10 clock cycles
  - DDR memory: ~100/~1000 clock cycles
  - Hard drives: even longer

- **Software engineers spend a lot of time restructuring their algorithms**
  - Increase the spatial locality of data in memory to increase the cache hit rate and decrease the processor time spent per instruction

Program execution on FPGA

FPGA is an inherently parallel processing fabric capable of implementing any logical and arithmetic function that can run on a processor

- Main difference: **Vivado HLS compiler**
  - Transforms software descriptions into RTL, is not hindered by the restrictions of a cache and a unified memory space
- Computation of z is compiled by Vivado HLS into several LUTs required to achieve the size of the output operand
- E.g.: In C code, variable a, b, and z are defined with the short data type (16-bit data container)
  - Variables gets implemented as 16 LUTs by Vivado HLS

General rule: 1 LUT is equivalent to 1 bit of computation

Execution steps on FPGA

- Vivado HLS compiler exercises the capabilities of the FPGA fabric using following processes:
  - Scheduling
  - Pipelining
  - Dataflow

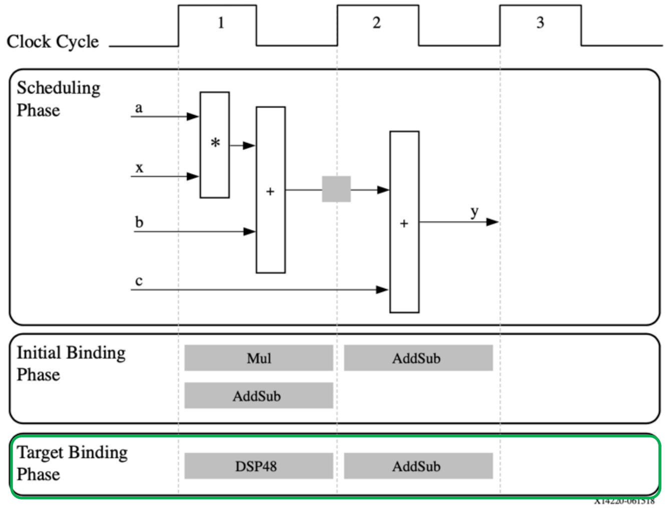
Transparent to the user, these processes are integral stages of the software compilation process that extract the best possible circuit-level implementation of the software application.

Scheduling

Process of identifying the data and control dependencies between different operations

- To Determine which operation occur during each clock cycle based on:
    - Length of the clock cycle or clock frequency
    - Time it takes for the operation to complete, as defined by the target device
    - User-specified optimization directives

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



Pipelining

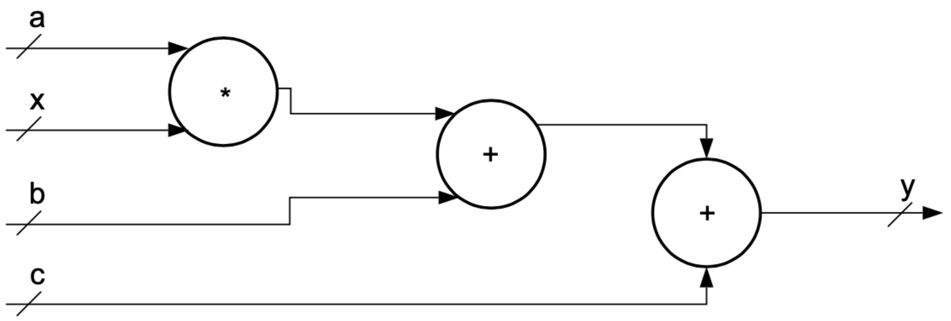
Technique to avoid data dependencies and increase the level of parallelism

- Preserving the original functionality, required circuit is divided into a chain of independent stages
- All stages in the chain run in parallel on the same clock cycle
- The only difference is the source of data for each stage
- Each stage in the computation receives its data values from the result computed by the preceding stage during the previous clock cycle

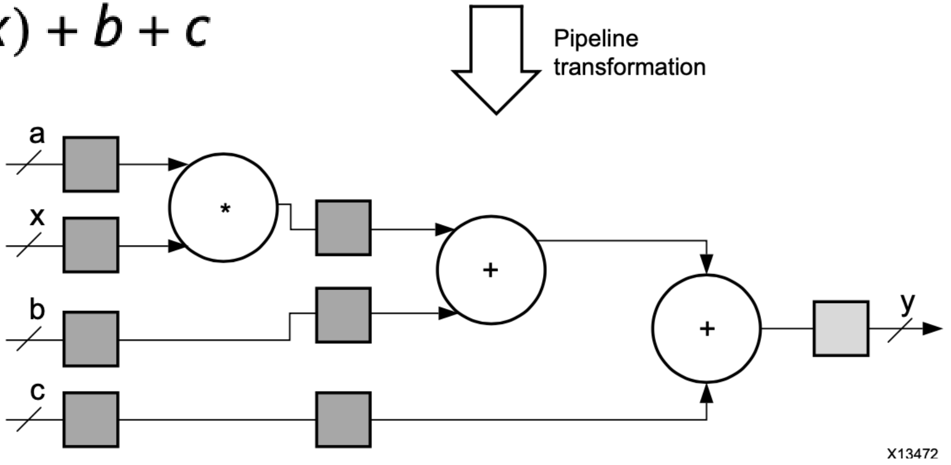
$$y = (a \times x) + b + c$$

Pipelining

C implementation



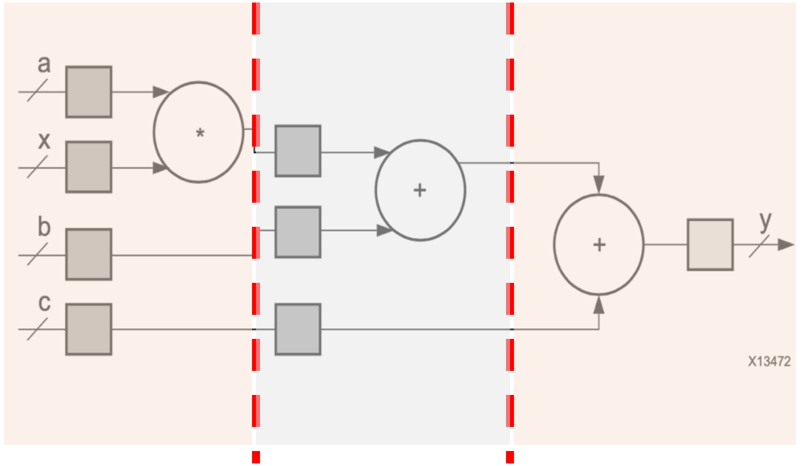
$$y = (a \times x) + b + c$$



Pipelined implementation

Pipelining

- Boxes: registers implemented by FF blocks
- Each box column counted as single clock cycle
- Result in 3 clock cycles.
- Addition of registers, leads to separated compute sections for each block
    - Multiplier & two adders can run in parallel and reduce latency

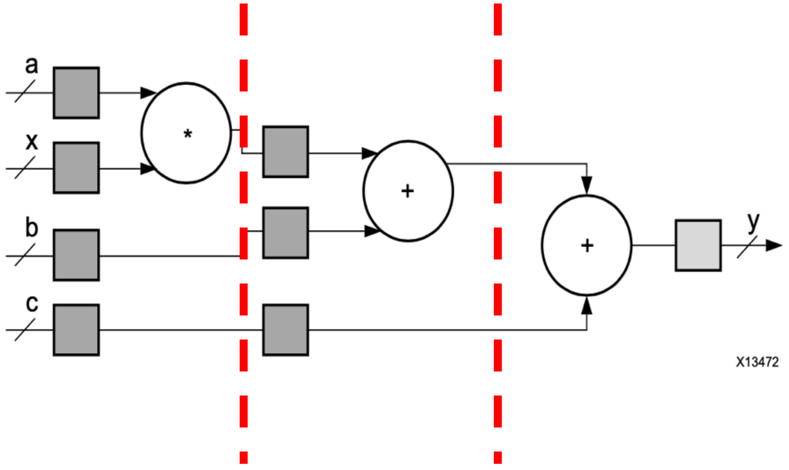


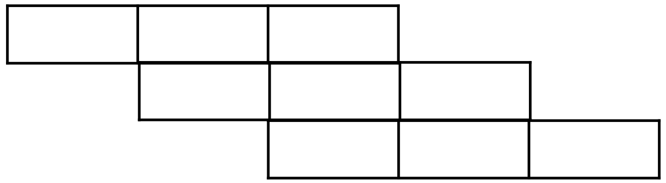
Pipelining

- Both sections of the datapath run in parallel
  - Essentially computing the **y** and **y’** in parallel
  - **y’** result of the next execution

  - First computation of y: pipeline fill time = 3 CLK

  - After this initial computation, a new value of y is available at the output on every clock cycle, because the computation pipeline contains overlapped data sets for the current and subsequent y computations



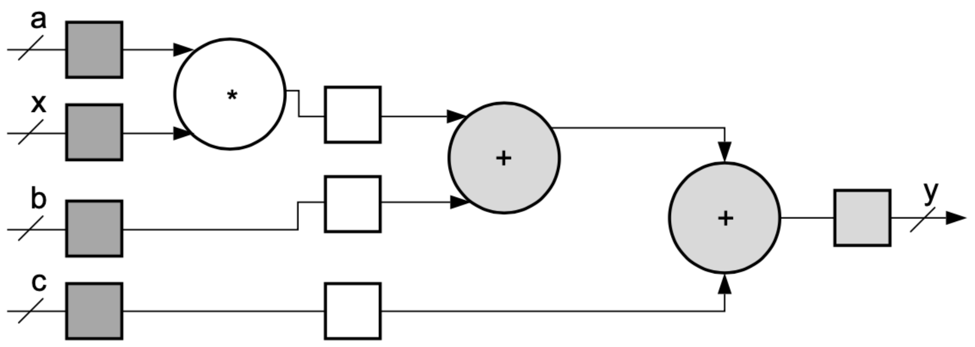


Pipelining

- Raw data: dark gray,
- Semi-computed data: white
- Final data: light gray

All exist simultaneously & each stage result is captured in its own set of registers

Although the latency for such computation is in multiple cycles, there is new result with every cycle



a(i)
x(i)
b(i)
c(i)
	a(i-1)*x(i-1)
	b(i-1)
	c(i-1)
		a(i-2)*x(i-2)+b(i-2)
		+c(i-2)

Dataflow

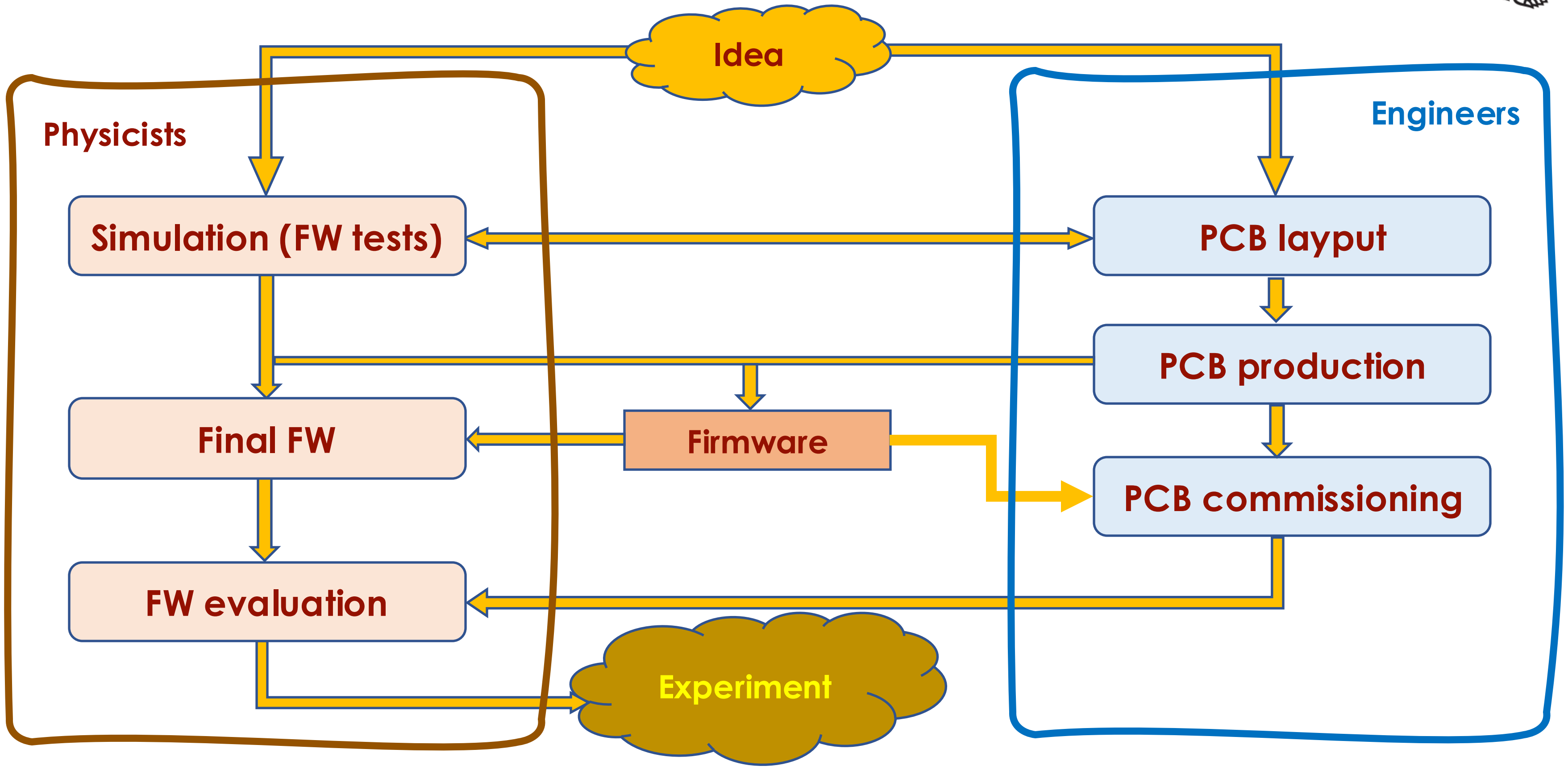
Similar to pipelining but parallelism at coarse-grain level

- Parallel execution of functions within a single program
  - By evaluating the interactions between different functions of a program based on their inputs and outputs

- Case-1: Independent (simplest)
  - Separate resources for different functions and run the blocks independently
- Case-2: Dependent (complex)
  - One function provides result for another function (*consumer-producer scenario*)

Why in HEP we need to know so much about FPGAs?

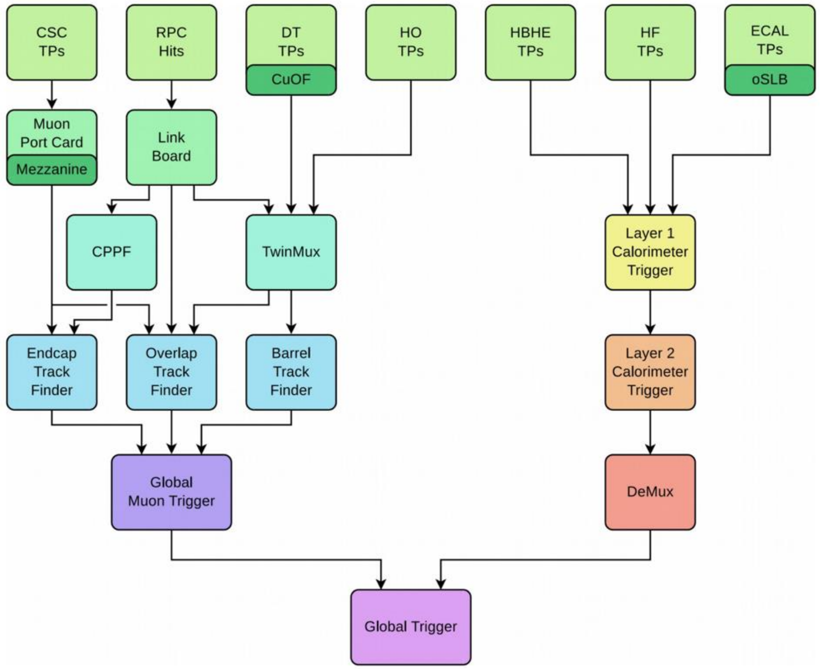
Workflow during FPGA development



Save $$$!!!

- Like our resources, each FPGA has limited resources
- FPGAs are expensive
- Need to design most optimal logic to have efficient functionality to meet the requirements

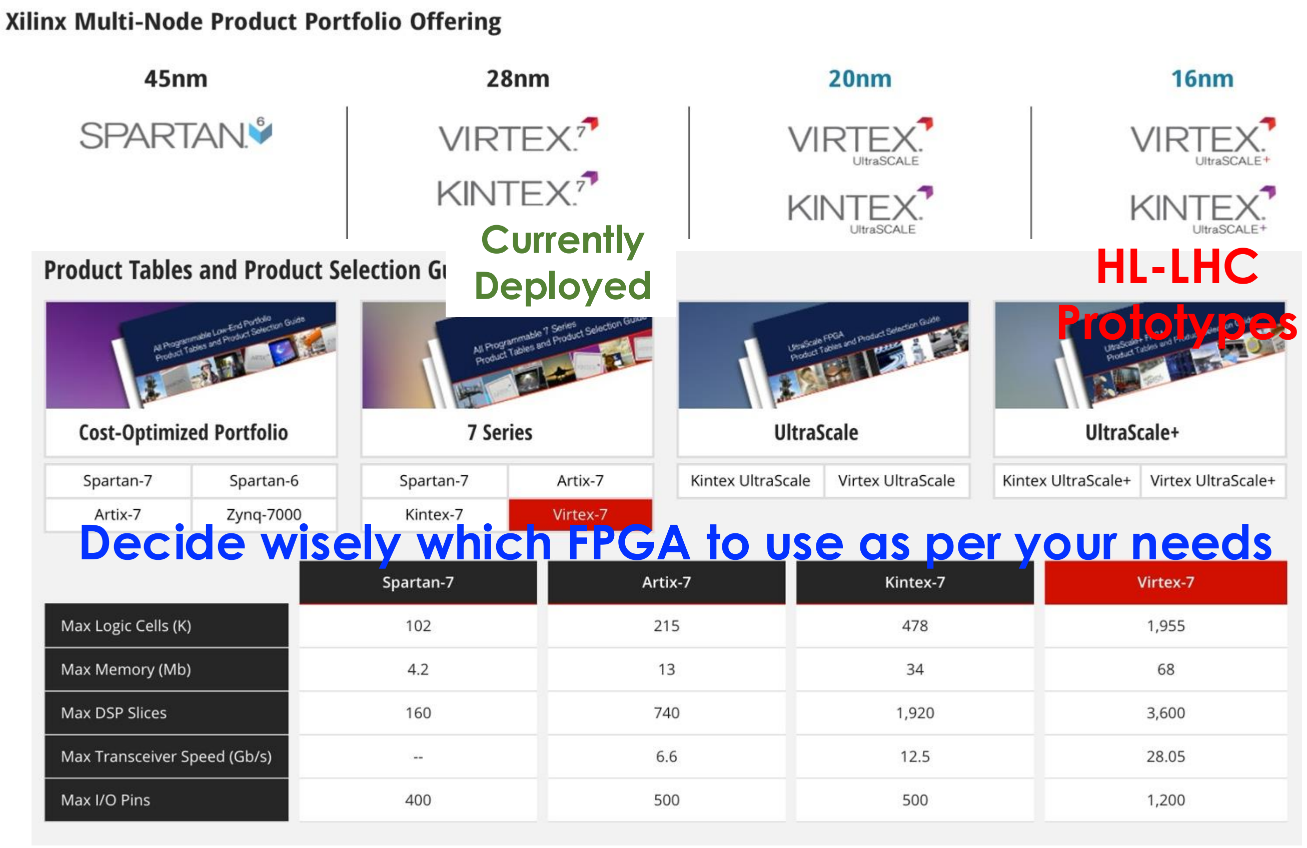
Back to Trigger!



To make decision in μs

- We have parallel/Pipelined system
- Feed Forward Algorithms (no backward loops)
- Highly distributed
- Parallelism in FPGA
- Parallelism in Logic

Xilinx FPGAs – Phase-1 choice: V7 690T



Xilinx Multi-Node Product Portfolio Offering

45nm	28nm	20nm	16nm
SPARTAN 6	VIRTEX 7	VIRTEX UltraSCALE	VIRTEX UltraSCALE+
	KINTEX 7	KINTEX UltraSCALE	KINTEX UltraSCALE+

Product Tables and Product Selection Gu

Cost-Optimized Portfolio		7 Series		UltraScale		UltraScale+	
Spartan-7	Spartan-6	Spartan-7	Artix-7	Kintex UltraScale	Virtex UltraScale	Kintex UltraScale+	Virtex UltraScale+
Artix-7	Zynq-7000	Kintex-7	Virtex-7				

	Spartan-7	Artix-7	Kintex-7	Virtex-7
Max Logic Cells (K)	102	215	478	1,955
Max Memory (Mb)	4.2	13	34	68
Max DSP Slices	160	740	1,920	3,600
Max Transceiver Speed (Gb/s)	--	6.6	12.5	28.05
Max I/O Pins	400	500	500	1,200

Currently Deployed

HL-LHC Prototypes

Decide wisely which FPGA to use as per your needs

Speed grade: maximum propagation delay for critical paths in the FPGA fabric or I/O operations

Key Element: Multi-Gigabit Opto-electronics

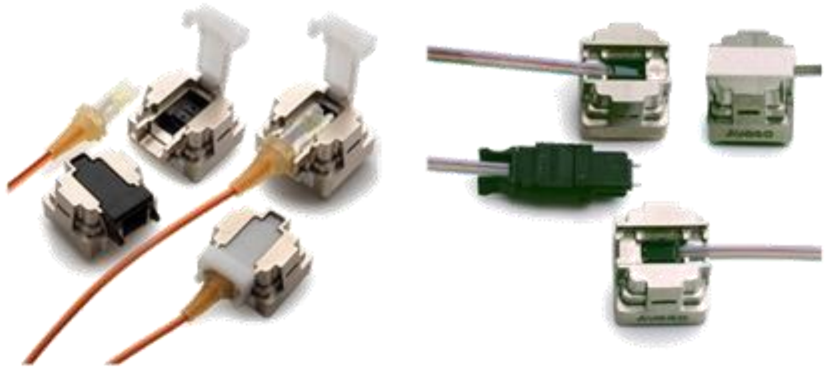


Figure 1. MiniPOD™ Transmitter and Receiver Modules with a) Round Cable and b) Flat Cable: shown with and without dust covers (White = Tx, Black = Rx).

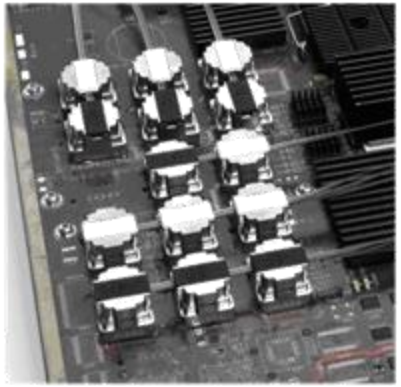


Figure 2. MiniPOD™ Transmitter and Receiver flat ribbon cable modules in a tiled arrangement example.

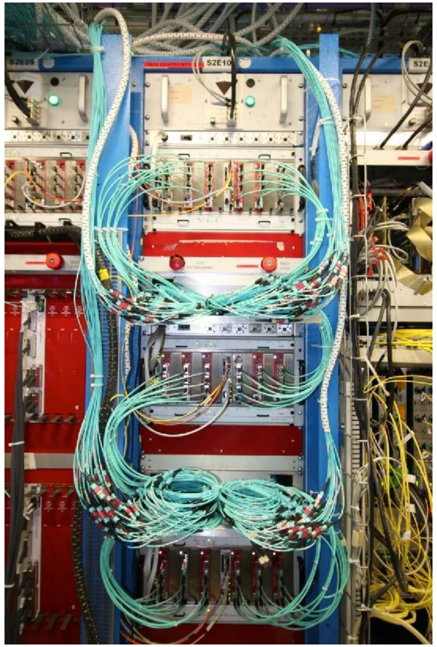
Key Product Parameters

The Avago Technologies MiniPOD™ modules operate at 850 nm and are compliant to the Multi-mode Fiber optical specs in clause 86 and relevant electrical specs in annex 86A of the IEEE 802.3ba specifications.

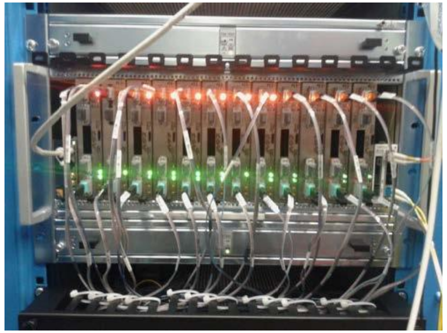
Parameter	Value	Units	Notes
Data rate per lane	10.3125	Gbps	As per 802.3ba: 100GBASE-SR10 and nPPI specifications
Number of operational lanes	12		100GbE operation utilizes the middle ten lanes (Rx and Tx) of the 12 physically defined lanes
Link Length	100 150	m m	OM3, 2000 MHzMHz•km 50 µm MMF OM4, 4700 MHz•km 50 µm MMF

CMS Level-1 Trigger Hardware

Calo Layer-1
CTP7: 18 boards

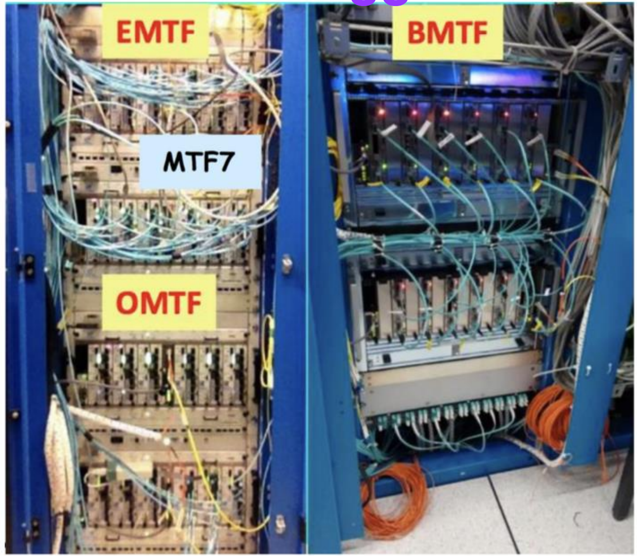


Calo Layer-2
CTP7: 10 boards



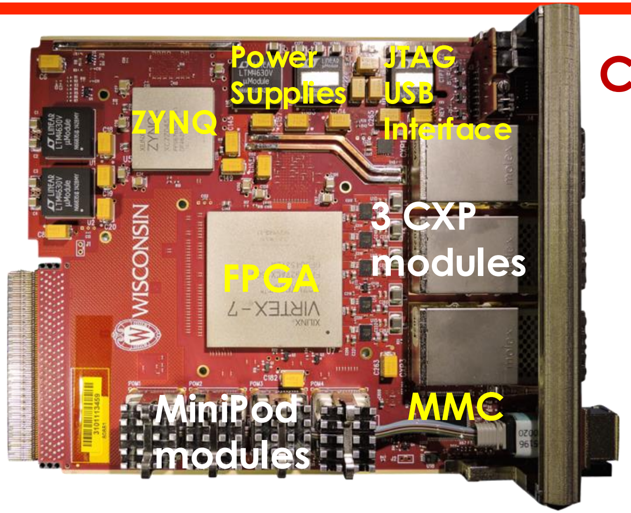
Time-Multiplexed Trigger

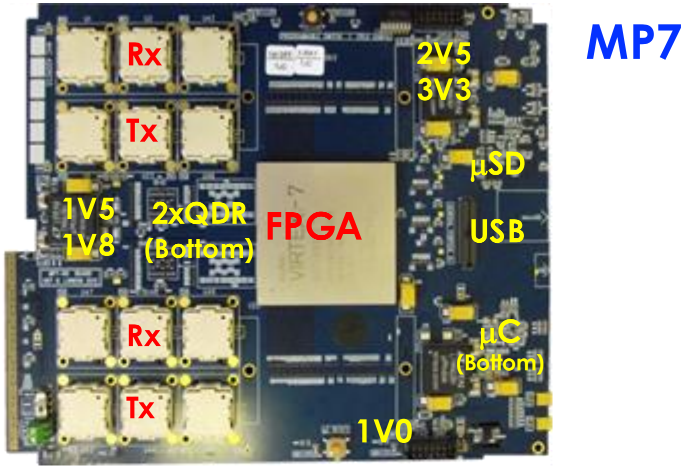
Muon Trigger



- Virtex7 FPGA used a main processor
- uTCA Form factor & infrastructure
- DAQ, slow control & monitoring

Trigger Processor Boards

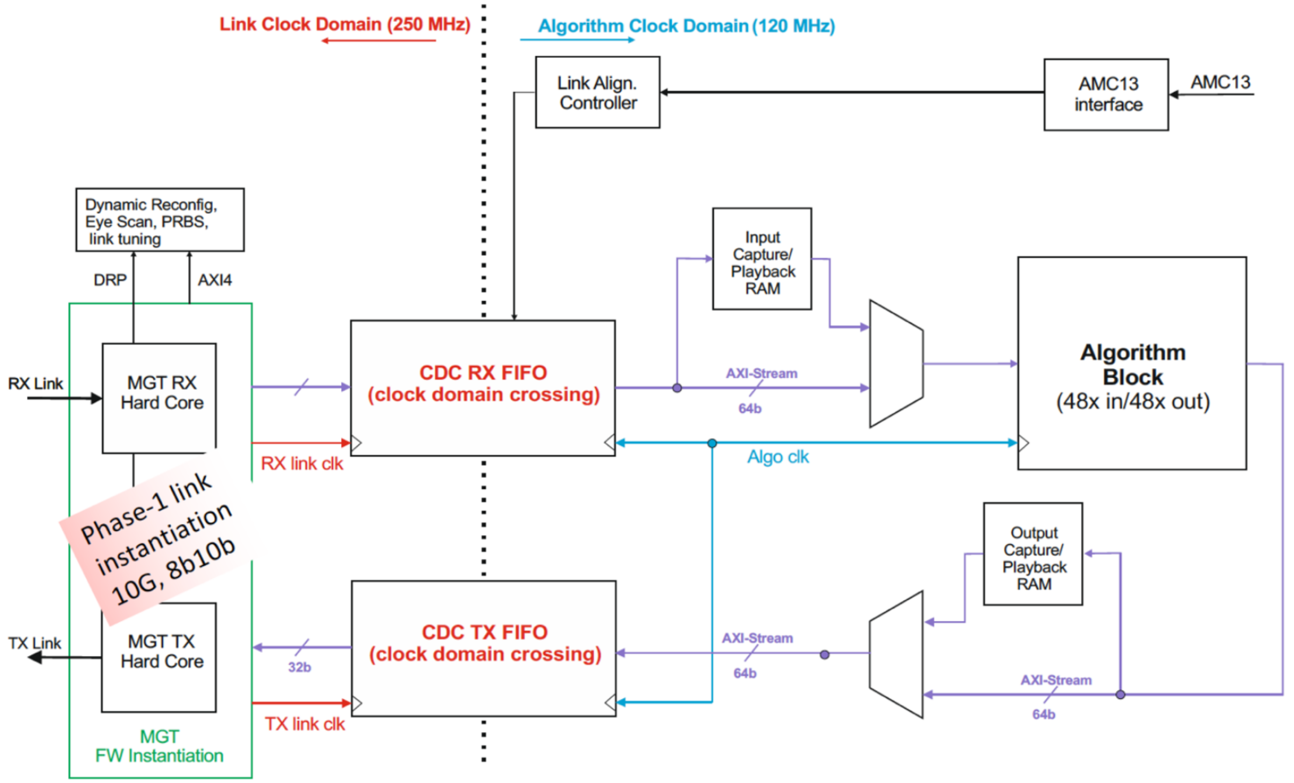




Calorimeter Trigger Processor(CTP7 – left), and Master Processor (MP7 - right)

- **CTP7 (Layer-1) – mTCA Single Virtex 7 FPGA, 67 optical inputs, 48 outputs, 12 RX/TX backplane**
  - Virtex 7 allows 10 Gb/s link speed on 3 CXP(36 TX & 36 RX) and 4 MiniPODs (31 RX & 12 TX)
  - ZYNQ processor running Xilinx PetaLinux for service tasks, including virtual JTAG cable
- **MP7 (Layer-2) – mTCA Single Virtex 7 FPGA, up to 72 input & output links**
  - Virtex 7 has 72 input and output links at 10 Gb/s
  - Dual 72 or 144MB QDR RAM clocked at 500 MHz

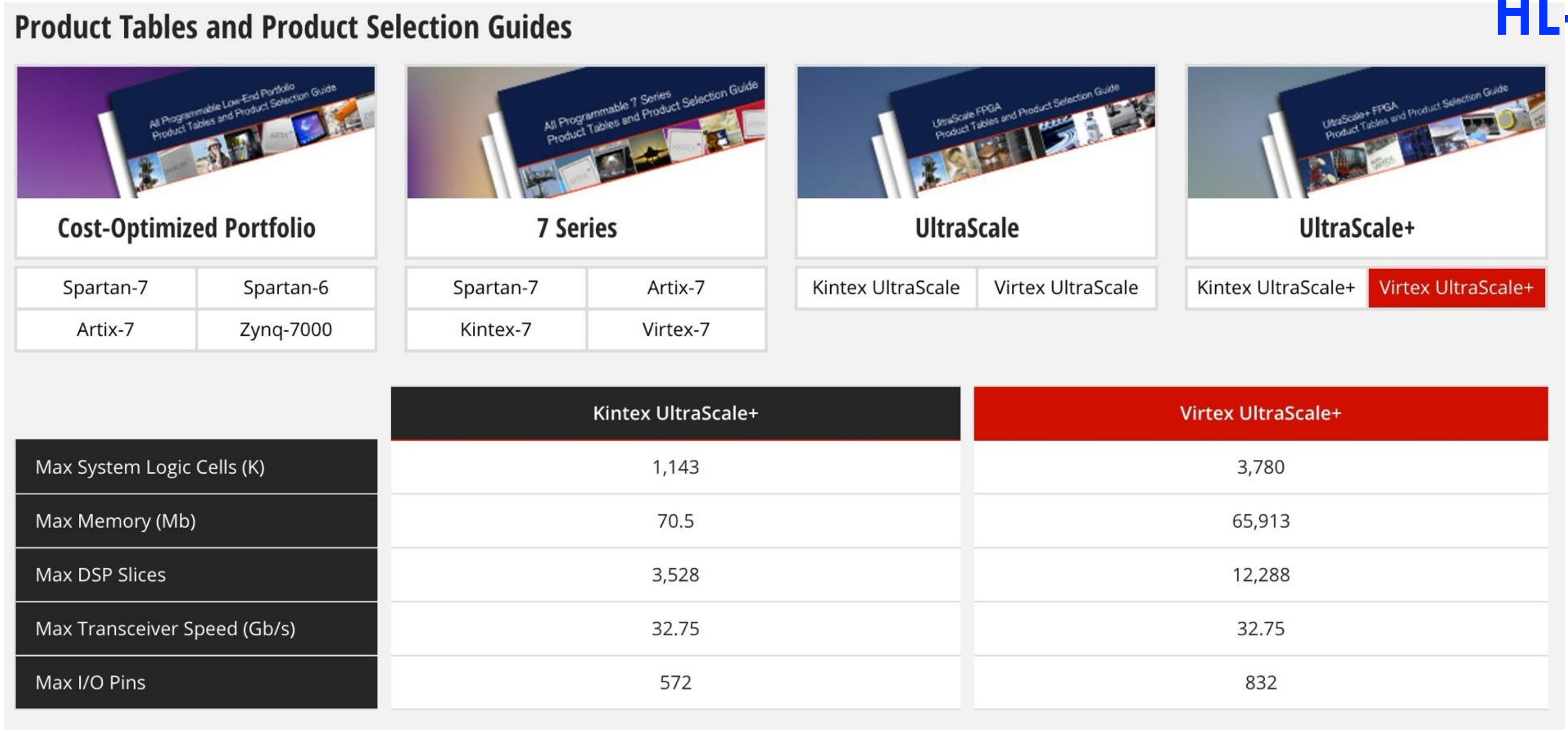
CTP7 FW Infrastructure



Link Clock Domain (250 MHz)
Algorithm Clock Domain (120 MHz)
Link Align. Controller
AMC13 interface
AMC13
Dynamic Reconfig, Eye Scan, PRBS, link tuning
DRP
AXI4
RX Link
MGT RX Hard Core
CDC RX FIFO (clock domain crossing)
RX link clk
Input Capture/ Playback RAM
AXI-Stream
64b
Algorithm Block (48x in/48x out)
Algo clk
Phase-1 link instantiation 10G, 8b10b
Output Capture/ Playback RAM
TX Link
MGT TX Hard Core
32b
CDC TX FIFO (clock domain crossing)
AXI-Stream
64b
AXI-Stream
64b
TX link clk
MGT FW Instantiation

Xilinx FPGAs – Phase-2 choice: VU13P

HL-LHC



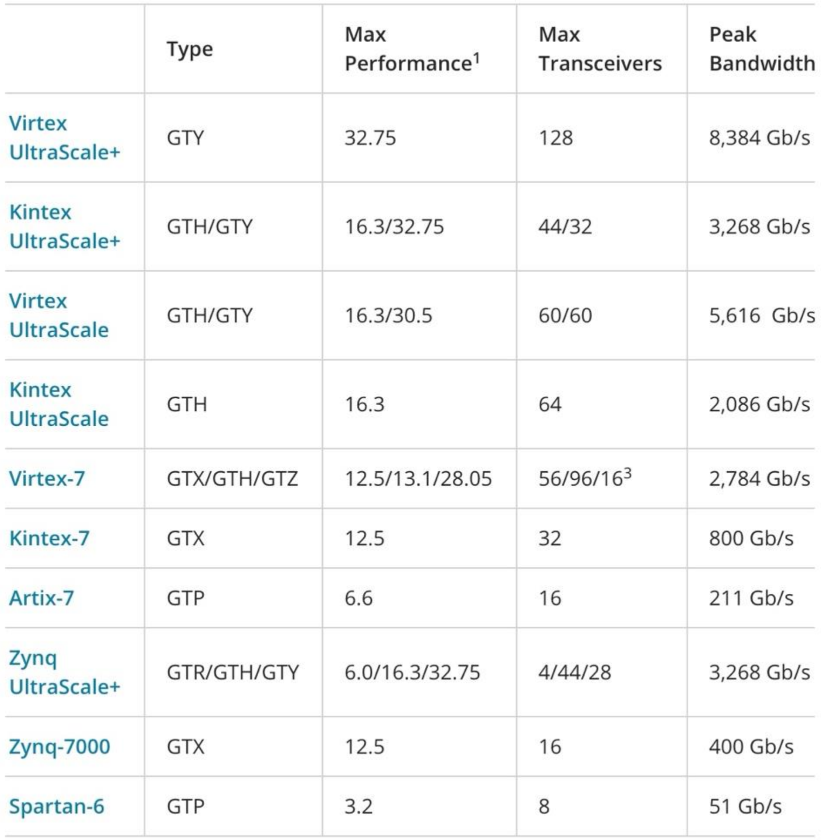
Product Tables and Product Selection Guides

Cost-Optimized Portfolio		7 Series		UltraScale		UltraScale+	
Spartan-7	Spartan-6	Spartan-7	Artix-7	Kintex UltraScale	Virtex UltraScale	Kintex UltraScale+	Virtex UltraScale+
Artix-7	Zynq-7000	Kintex-7	Virtex-7				

	Kintex UltraScale+	Virtex UltraScale+
Max System Logic Cells (K)	1,143	3,780
Max Memory (Mb)	70.5	65,913
Max DSP Slices	3,528	12,288
Max Transceiver Speed (Gb/s)	32.75	32.75
Max I/O Pins	572	832

Multi-gigabit-per-second serial links

HL-LHC

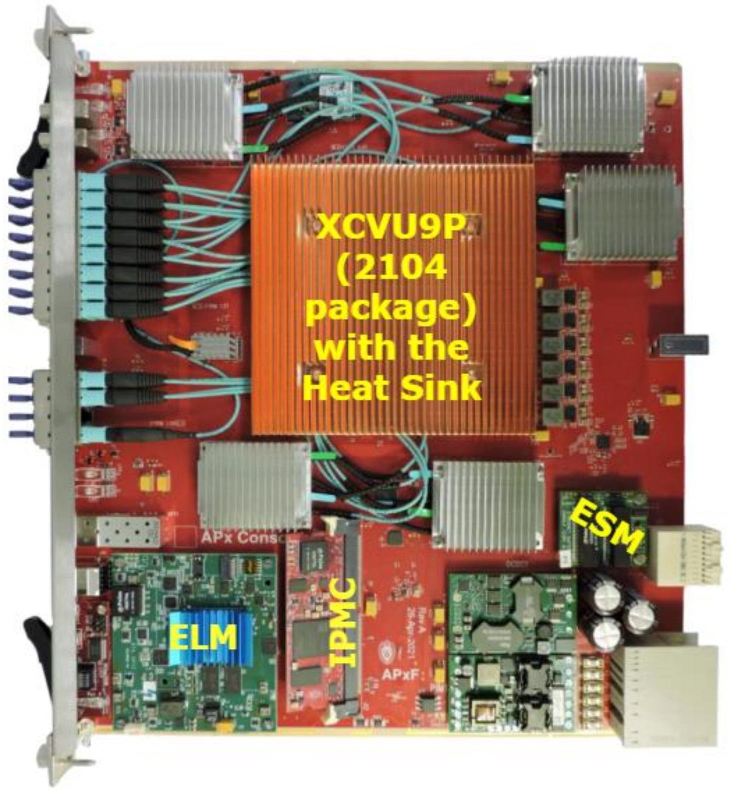


	Type	Max Performance[1]	Max Transceivers	Peak Bandwidth
Virtex UltraScale+	GTY	32.75	128	8,384 Gb/s
Kintex UltraScale+	GTH/GTY	16.3/32.75	44/32	3,268 Gb/s
Virtex UltraScale	GTH/GTY	16.3/30.5	60/60	5,616 Gb/s
Kintex UltraScale	GTH	16.3	64	2,086 Gb/s
Virtex-7	GTX/GTH/GTZ	12.5/13.1/28.05	56/96/16[3]	2,784 Gb/s
Kintex-7	GTX	12.5	32	800 Gb/s
Artix-7	GTP	6.6	16	211 Gb/s
Zynq UltraScale+	GTR/GTH/GTY	6.0/16.3/32.75	4/44/28	3,268 Gb/s
Zynq-7000	GTX	12.5	16	400 Gb/s
Spartan-6	GTP	3.2	8	51 Gb/s

HL-LHC ← 25 Gbps

LHC → 10 Gbps

Advanced Processor Prototype for HL-LHC

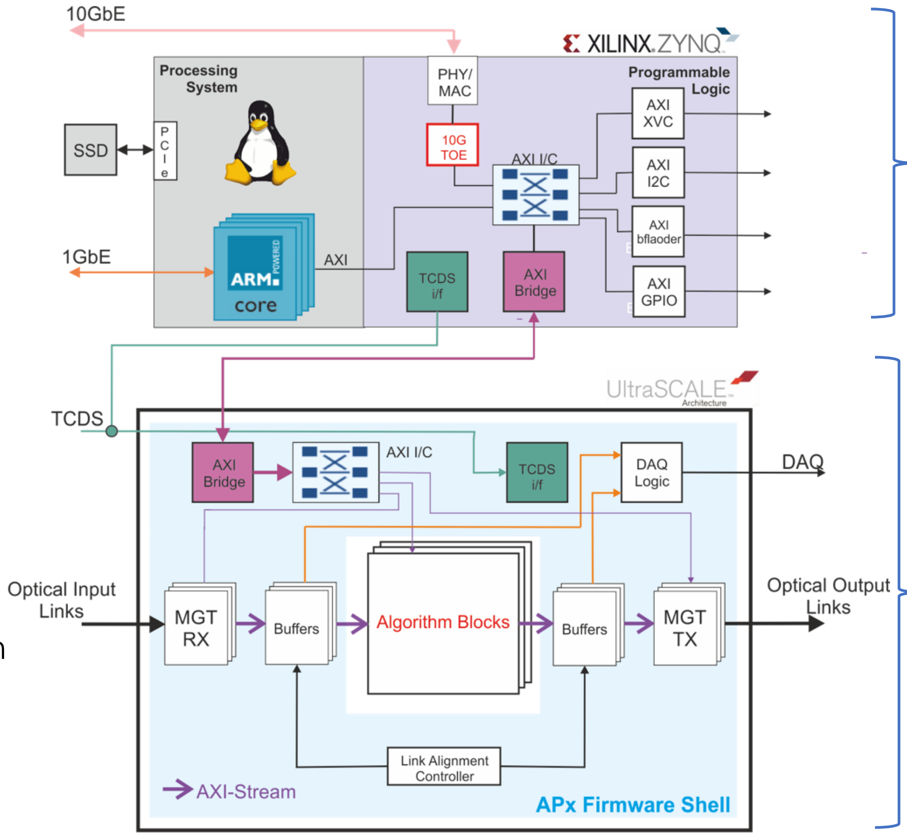


- Wisconsin APxF Board
- **Xilinx VU13P or VU9P FPGA**
- ZYNQ-IPMC
  (ATCA IPMI controller)
- ELM (ZYNQ-based embedded Linux endpoint)
- ESM (GbE switch)
- High efficiency heatsinks
- Front-panel inputs
- 25G Samtec Firefly positions loaded – 10x12 + 1x4
  (124 25 Gbps links)

Latenct budget for HL-LHC: 12.5 μs

APx – Firmware/Software

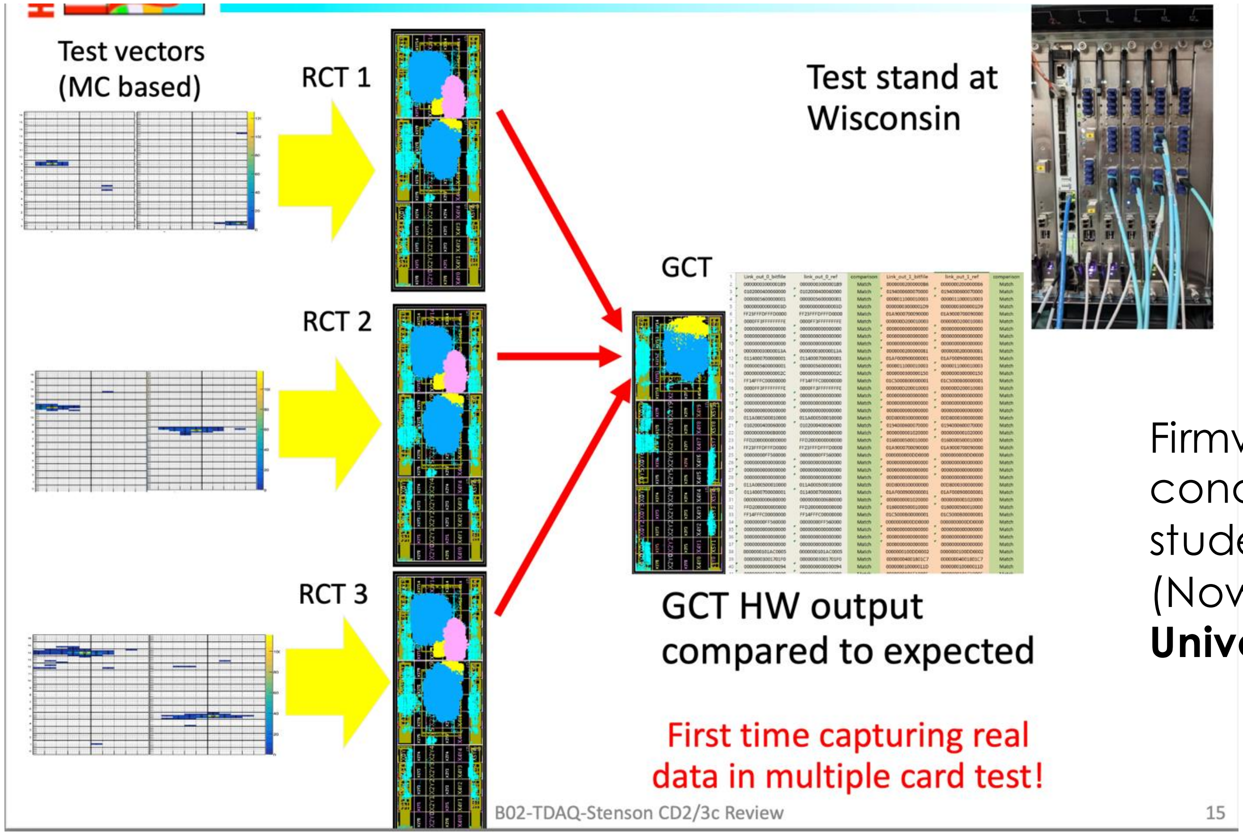
- A new paradigm for firmware development
  - Core firmware written in VHDL by engineers
  - Gigabit link support
  - Data exchange between SLRs within chip
  - Test buffers
  - Clock and control
- Physics
  - Algorithmic firmware in high-level languages like C++ written by physicists



ELM:
Control endpoint, providing complete board overhead functionality

Processing FPGA:
- I/O
- data processor
- DAQ

Trigger Upgrade FW testing


Test vectors (MC based)

RCT 1

RCT 2

RCT 3

GCT

Test stand at Wisconsin

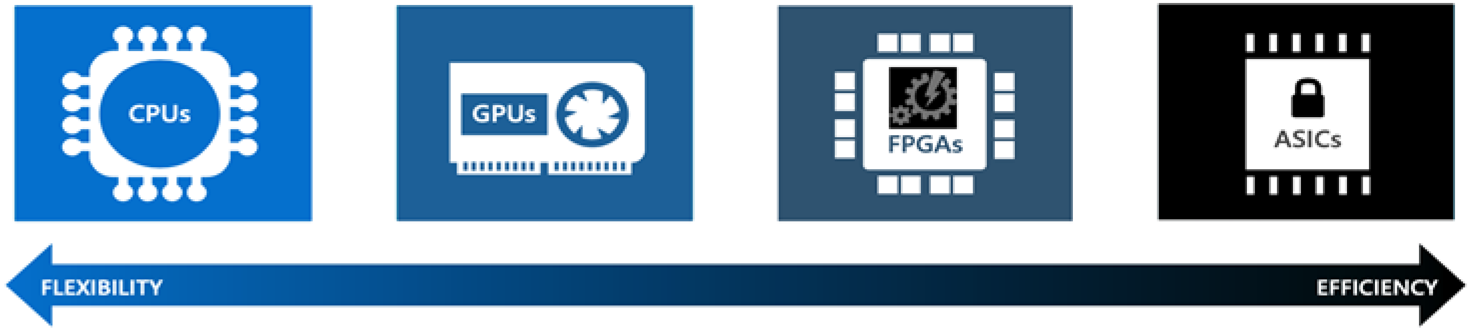
GCT HW output compared to expected

First time capturing real data in multiple card test!

B02-TDAQ-Stenson CD2/3c Review 15

Firmware prepared and tests conducted by former UoH PhD student **Piyush Kumar** (Now Research Engineer @ **University of Notre Dame, USA**

Comparison of CPU/GPU/FPGA/ASICS



CPU/FPGA Advantages

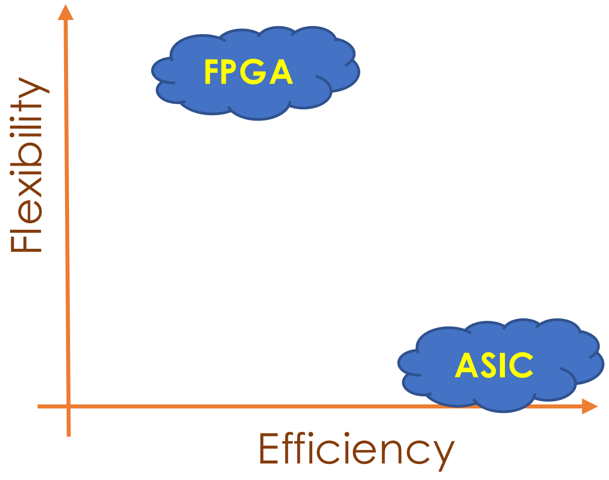
CPU advantages	FPGA Advantages
• Better with floating point numbers • Programming a CPU is normally easier than programming an FPGA (does not require to understand digital electronics) • Faster compilation • Easier code portability • Lower unit cost	• More versatile & adaptable • More flexible input/output • Parallel processing • Better with multi-clock systems • Better with time-critical operations • Power Efficient • Faster than processors

More and more often, FPGAs and CPUs (or GPUs) are complementary:
They co-exist in the same system and perform different tasks

ASICS

ASIC: Application **S**pecific **I**ntegrated **C**ircuit

FPGAs were originally popular for prototyping ASICs, but now also for high performance computing



FPGA/ASIC Advantages

FPGA Advantages	ASIC Advantages
Faster time-to-market - no layout, masks or other manufacturing steps are needed **Lower constant/initial cost** **Simpler design cycle** - due to software that handles much of the routing, placement, and timing **More predictable project cycle** due to elimination of potential re-spins, wafer capacities, etc. **Re-programmability**: a new configuration can be uploaded	**Full custom capability (including analog) -** since device is manufactured to design specs **Lower unit costs** – For mass production **Smaller form factor** - since device is manufactured to design specs **Higher clock speeds**

Uses of FPGAs outside HEP

- **Telecommunication**
- Automotive
- **Aerospace and Defense**
- Medical Electronics
- **ASIC Prototyping**
- Audio
- **Broadcast**
- Consumer Electronics
- **Data Center**
- Distributed Monetary Systems
- **High Performance Computing**
- Industrial
- Scientific Instruments
- Security systems
- Video & Image Processing
- Digital signal processing
- **Bioinformatics**
- Controllers
- **Computer hardware emulation**
- Voice recognition
- **Cryptography**

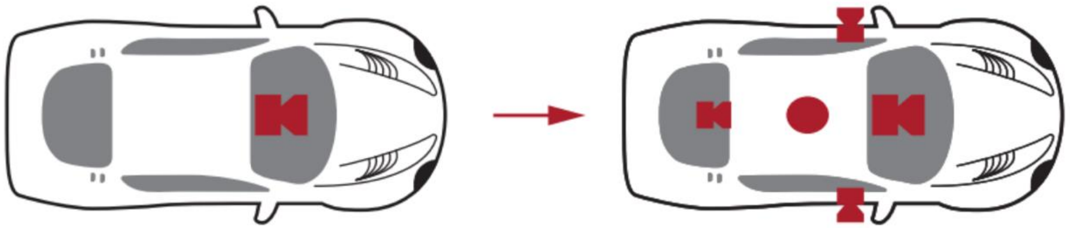
More Advanced Architectures

- Embedded FPGA System on Chip (SoC)

- High Bandwidth Memory (HBM) on Xilinx FPGA
  - A theoretical bandwidth up to 460 GB/s

- ACAP: Adaptive Compute Acceleration Platform
  - *A fully software-programmable, heterogeneous compute platform that combines Scalar Engines, Adaptable Engines, and Intelligent Engines to achieve dramatic performance improvements of up to 20X over today's fastest FPGA implementations and over 100X over today's fastest CPU implementations—for Data Center, wired network, 5G wireless, and automotive driver assist applications.*

ACAP Application



1x HD Camera

~10W

Sensor Fusion

4x HD Cameras

Radar

Ultrasound

LIDAR

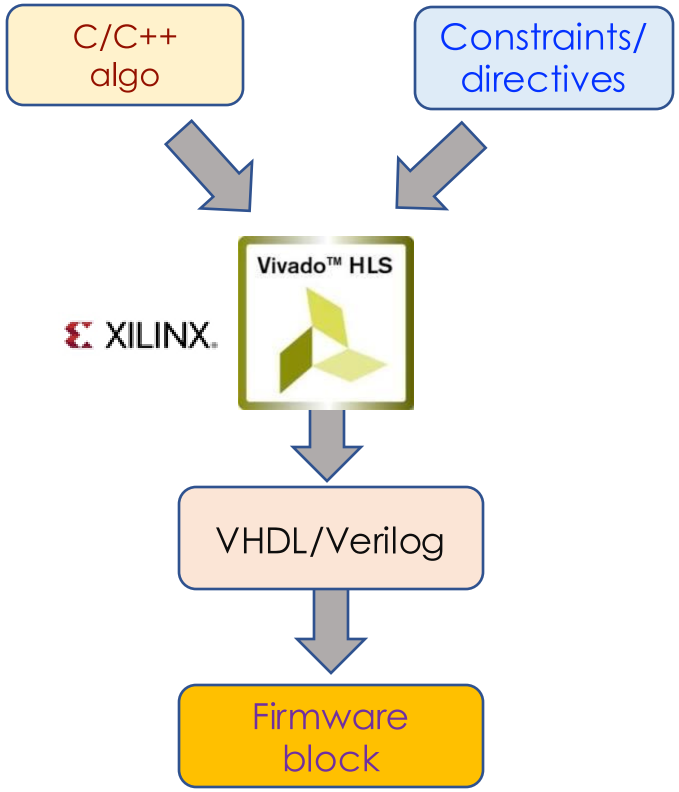
Machine Learning

~10W

WP505_13_092818

Xilinx ACAP Devices enable sensor fusion in small power envelopes

Path to firmware



High Level Synthesis (HLS)

- Compile from C/C++ to VHDL/Verilog
- Pre-processor directives and constraints used to optimize the design

Hardware Description Languages

- VHDL/Verilog
- Programming languages which describe electronic circuits

Drastic decrease in firmware development time!

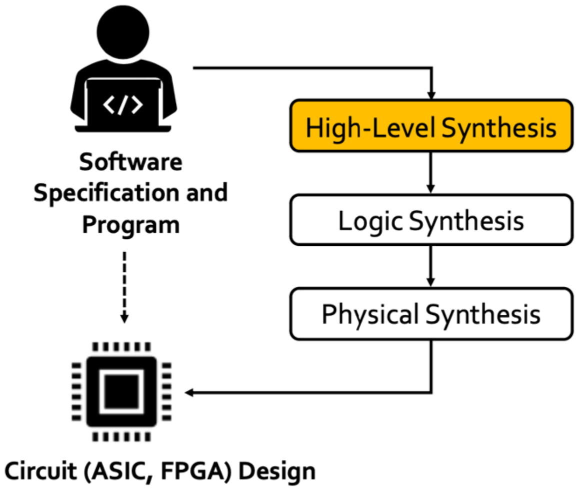
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf

High Level Synthesis

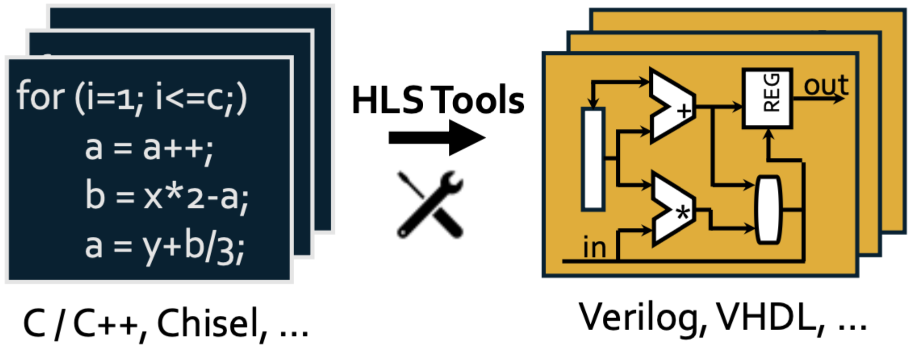
https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Getting-Started-with-Vitis-HLS

What is HLS?

HLS is an automated design process that transforms a high-level functional specification to an optimized register-transfer level (RTL) descriptions for efficient hardware implementation



Software Specification and Program
High-Level Synthesis
Logic Synthesis
Physical Synthesis
Circuit (ASIC, FPGA) Design

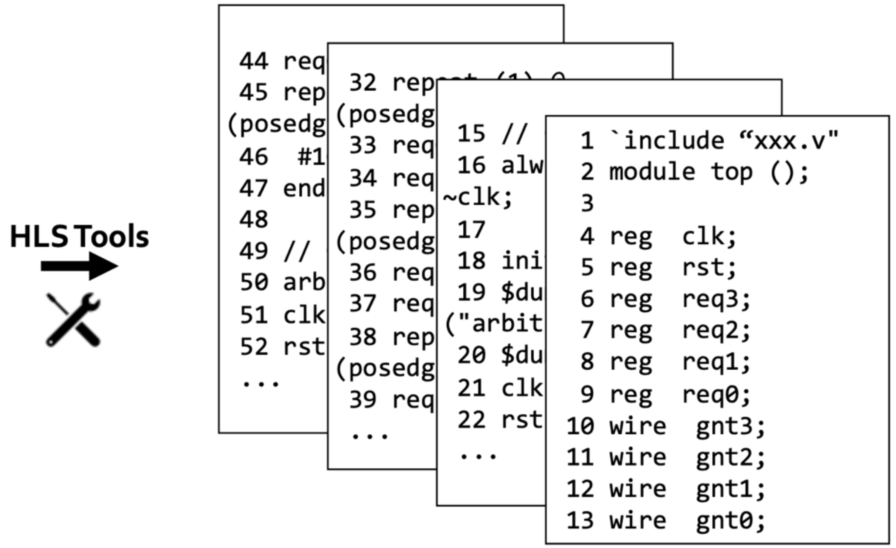


```
for (i=1; i<=c;)
        a = a++;
        b = x*2-a;
        a = y+b/3;
```

What is HLS?

```
for(int h = 0; h < H; h++ )
  for(int w = 0; w < W; w++)
    for(int m = 0; m < K; m++)
      for(int n = 0; n < K; n++)
        ...
```

Behavioral-level:
Expressive and Concise



```
req
rep
(posedg
  #1
end

//
arb
clk
rst
...
```

```
repeat (1) @
(posedg
req
req
rep
(posedg
req
req
rep
(posedg
req
...
```

```
//
alw
~clk;

ini
$du
("arbit
$du
clk
rst
...
```

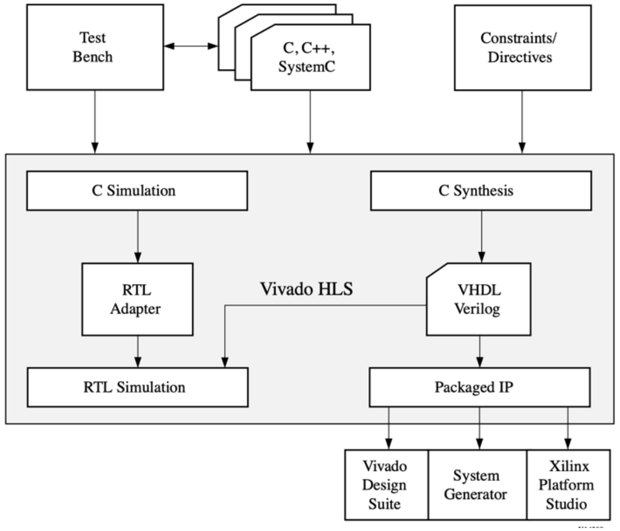
```
`include “xxx.v"
module top ();

reg  clk;
reg  rst;
reg  req3;
reg  req2;
reg  req1;
reg  req0;
wire  gnt3;
wire  gnt2;
wire  gnt1;
wire  gnt0;
```

Why use HLS?

- Productivity
  - Lower design complexity and faster simulation speed
  - Ease of use
- Portability
  - Single source -> multiple implementations (different target devices)
- Permutability
  - Much more optimization opportunities at higher level
  - Rapid design space exploration

HLS Design Flow

- Compile, execute (simulate), and debug the C algorithm
- Synthesize the C algorithm into an RTL implementation, optionally using user optimization directives
- Generate comprehensive reports and analyze the design
- Verify the RTL implementation using a pushbutton flow
- Package the RTL implementation into a selection of IP formats



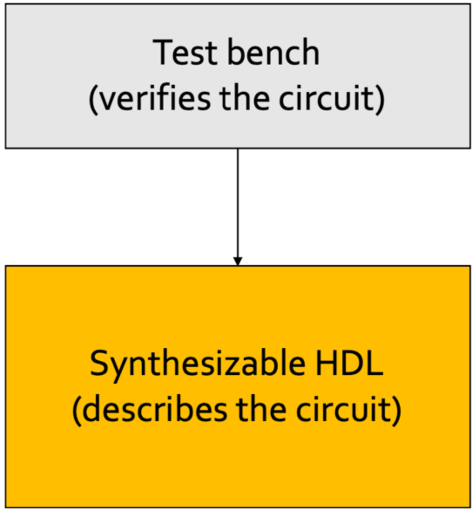
Simulation and Synthesis

The two major purposes of HDLs are logic simulation and synthesis:

- **During simulation**, inputs are applied to a module, and the outputs are checked to verify that the module operates correctly
- **During synthesis,** the textual description of a module is transformed into logic gates

HDL code is divided into synthesizable modules and a test bench:

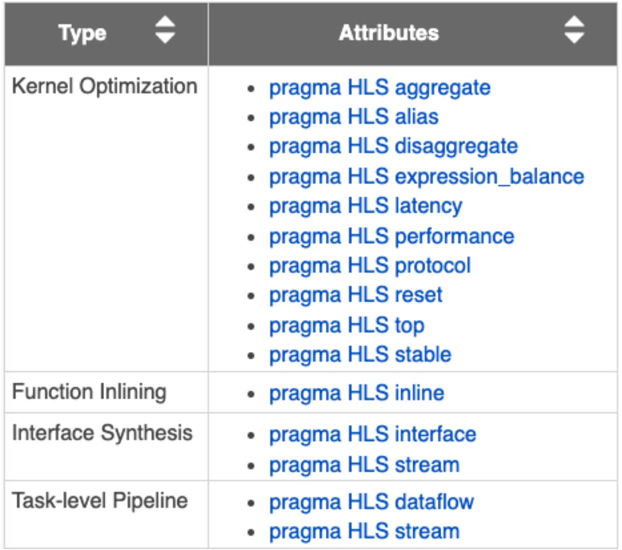
- The synthesizable modules describe the hardware
- The test bench checks whether the output results are correct (only for simulation and cannot be synthesized)



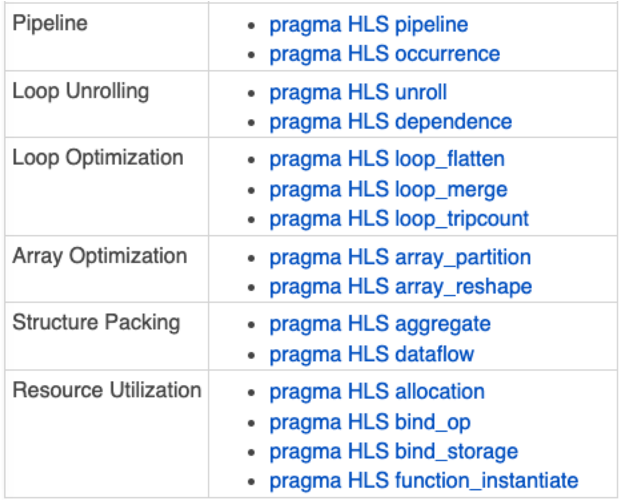
HLS Pragmas

"Pragmas": Instructions to tell your compiler how to build the hardware

- HLS tool provides different set of pragmas that can be used to optimize the design, reduce latency, improve performance etc. These pragmas can be directly added to the source code for the kernel.



Type	Attributes
Kernel Optimization	• pragma HLS aggregate • pragma HLS alias • pragma HLS disaggregate • pragma HLS expression_balance • pragma HLS latency • pragma HLS performance • pragma HLS protocol • pragma HLS reset • pragma HLS top • pragma HLS stable
Function Inlining	• pragma HLS inline
Interface Synthesis	• pragma HLS interface • pragma HLS stream
Task-level Pipeline	• pragma HLS dataflow • pragma HLS stream



Pipeline	• pragma HLS pipeline • pragma HLS occurrence
Loop Unrolling	• pragma HLS unroll • pragma HLS dependence
Loop Optimization	• pragma HLS loop_flatten • pragma HLS loop_merge • pragma HLS loop_tripcount
Array Optimization	• pragma HLS array_partition • pragma HLS array_reshape
Structure Packing	• pragma HLS aggregate • pragma HLS dataflow
Resource Utilization	• pragma HLS allocation • pragma HLS bind_op • pragma HLS bind_storage • pragma HLS function_instantiate

https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/HLS-Pragmas

Pragma HLS array_partition

- Partitions an array into smaller arrays or individual elements and provides the following:
  - Results in RTL with multiple small memories or multiple registers instead of one large memory
  - Effectively increases the amount of read and write ports for the storage
  - Potentially improves the throughput of the design
  - Requires more memory instances or registers

Syntax:
Place the pragma in the C source within the boundaries of the function where the array variable is defined

#pragma HLS array_partition variable=<name> <type> factor=<int> dim=<int>

Ex: Pragma HLS array_partition

```
#pragma HLS array_partition variable=AB block factor=4
```

- This example partitions the 13 element array, AB[13], into four arrays using block partitioning:
  - Because four is not an integer factor of 13:
  - Three of the new arrays have three elements each,
  - One array has four elements (AB[9:12])

```
#pragma HLS array_partition variable=AB block factor=2 dim=2
```

- This example partitions dimension two of the two-dimensional array, AB[6][4] into two new arrays of dimension [6][2]:

Pragma HLS unroll

- Unroll loops to create multiple independent operations rather than a single collection of operations
- **UNROLL** pragma transforms loops by creating multiples copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel
- Loops in the C/C++ functions are kept rolled by default
  - When loops are rolled, synthesis creates the logic for one iteration of the loop, and the RTL design executes this logic for each iteration of the loop in sequence
- **UNROLL** pragma allows the loop to be fully or partially unrolled
  - Fully unrolling the loop creates a copy of the loop body in the RTL for each loop iteration, so the entire loop can be run concurrently
  - Partially unrolling a loop lets you specify a factor *N*

Ex: Pragma HLS unroll

#pragma HLS unroll factor=<N> region skip_exit_check

The following example fully unrolls loop_1 in function *foo*

Place the pragma in the body of loop_1 as shown:

```
loop_1: for(int i = 0; i < N; i++) {
  #pragma HLS unroll
  a[i] = b[i] + c[i];
}
```

This example specifies an unroll factor of 4 to partially unroll loop_2 of function foo, and removes the exit check:

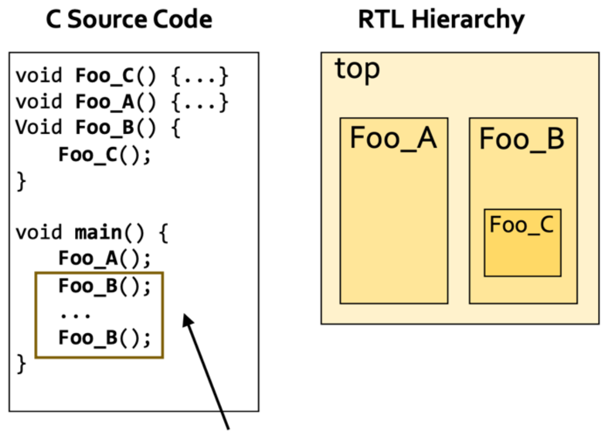
```
void foo (...) {
  int8 array1[M];
  int12 array2[N];
  ...
  loop_2: for(i=0;i<M;i++) {
    #pragma HLS unroll skip_exit_check factor=4
    array1[i] = ...;
    array2[i] = ...;
    ...
  }
  ...
}
```

Basic Mapping Rule C/C++ ➛ RTL

C Constructs		RTL Components
Functions	→	Modules
Arguments	→	I/O Ports
Operators (+, *)	→	Functional units (adder, multiplier)
Scalars	→	Wires or registers
Arrays	→	Memory
Control flows	→	Control logics (Finite State Machine)

Basic Mapping Rule C/C++ ➤ RTL

C Constructs		RTL Components
Functions	→	Modules
Arguments	→	I/O Ports
Operators (+, *)	→	Functional units (adder, multiplier)
Scalars	→	Wires or registers
Arrays	→	Memory
Control flows	→	Control logics (Finite State Machine)



```
void Foo_C() {...}
void Foo_A() {...}
Void Foo_B() {
    Foo_C();
}

void main() {
    Foo_A();
    Foo_B();
    ...
    Foo_B();
}
```

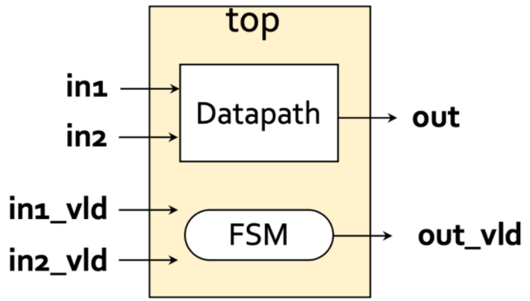
Resource Sharing: Only one **instance** of Foo_B written to the hardware

Basic Mapping Rule C/C++ ➛ RTL

C Constructs		RTL Components
Functions	→	Modules
Arguments	→	I/O Ports
Operators (+, *)	→	Functional units (adder, multiplier)
Scalars	→	Wires or registers
Arrays	→	Memory
Control flows	→	Control logics (Finite State Machine)

C Source Code

```
void top(int* in1, int* in2, int* out) {
    *out = *in1 + *in2;
}
```

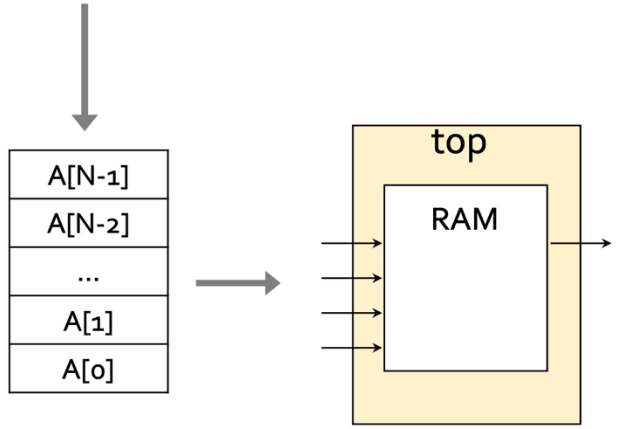


Basic Mapping Rule C/C++ ➤ RTL

C Constructs		RTL Components
Functions	→	Modules
Arguments	→	I/O Ports
Operators (+, *)	→	Functional units (adder, multiplier)
Scalars	→	Wires or registers
Arrays	→	Memory
Control flows	→	Control logics (Finite State Machine)

C Source Code

```
for (i = 0; i < N; i++)
  A[i+x] = A[i] + i;
```



Deterministic at Compile time

On an FPGA, memory maps to a physical address space

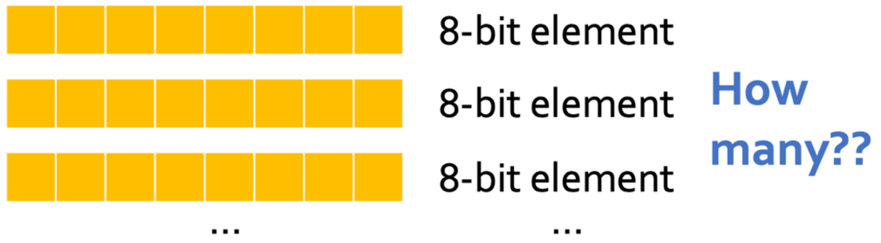
Everything must be decided at compile time – your hardware cannot be changed while running!

- Adding one more piece of memory after the circuit is built?

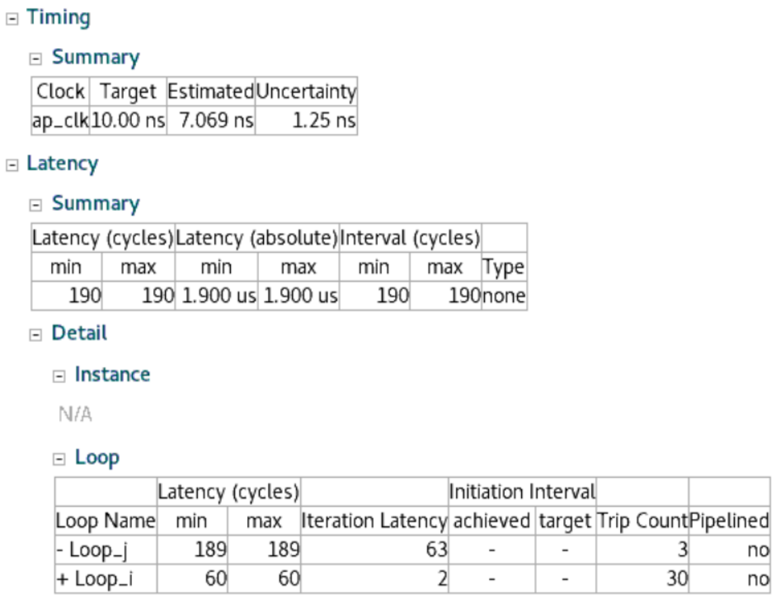
```
int mem[var];
```

```
int mem* = malloc(var * sizeof(int));
```

```
reg [0:7] mem [var:0];
```



Lets run some examples (ex1)



⊟ Timing

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	7.069 ns	1.25 ns

⊟ Latency

⊟ Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
190	190	1.900 us	1.900 us	190	190	none

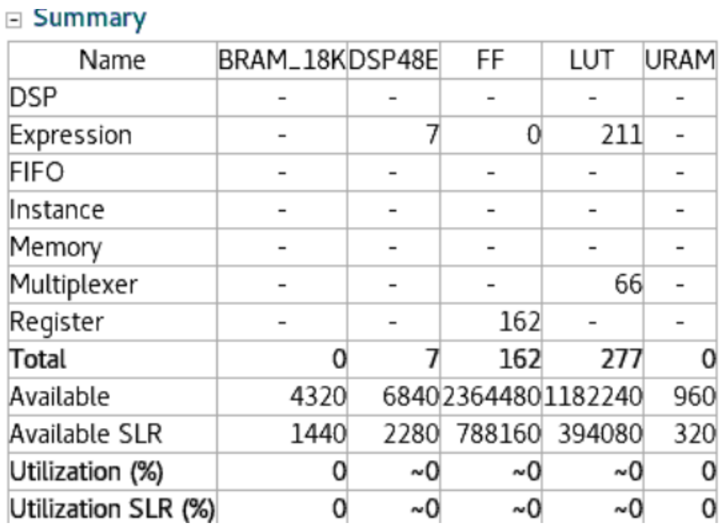
⊟ Detail

⊟ Instance

N/A

⊟ Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop_j	189	189	63	-	-	3	no
+ Loop_i	60	60	2	-	-	30	no



⊟ Summary

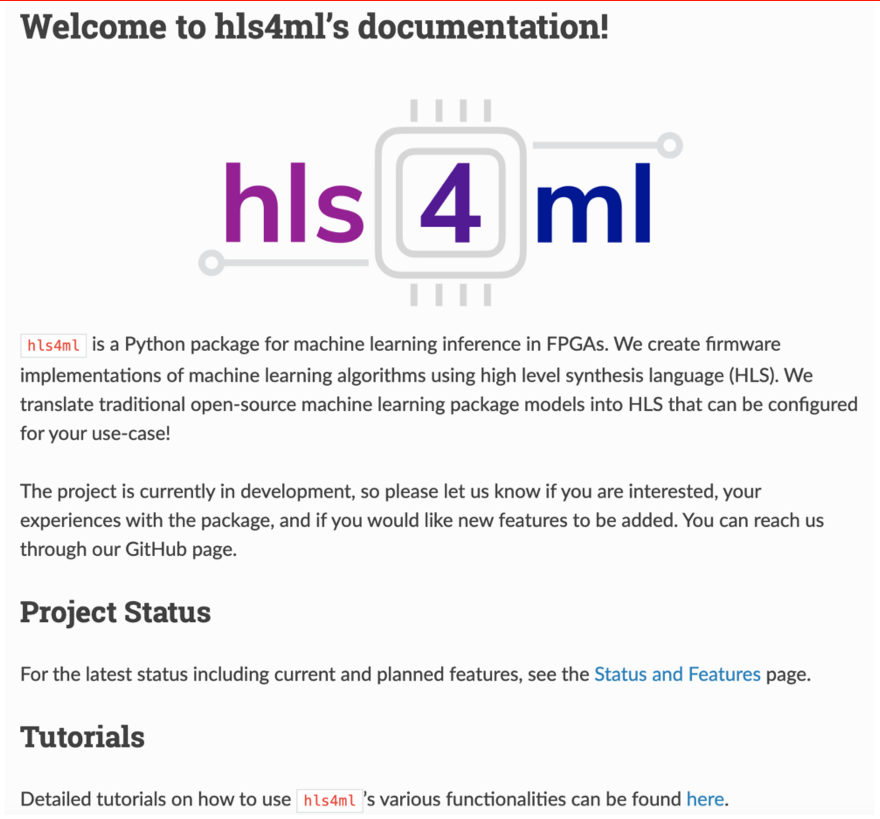
Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	7	0	211	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	66	-
Register	-	-	162	-	-
Total	0	7	162	277	0
Available	4320	6840	2364480	1182240	960
Available SLR	1440	2280	788160	394080	320
Utilization (%)	0	~0	~0	~0	0
Utilization SLR (%)	0	~0	~0	~0	0

Is Machine Learning Possible on FPGAs?

hls4ml

hls4ml

https://fastmachinelearning.org/hls4ml/



Welcome to hls4ml's documentation!

`hls4ml` is a Python package for machine learning inference in FPGAs. We create firmware implementations of machine learning algorithms using high level synthesis language (HLS). We translate traditional open-source machine learning package models into HLS that can be configured for your use-case!

The project is currently in development, so please let us know if you are interested, your experiences with the package, and if you would like new features to be added. You can reach us through our GitHub page.

Project Status

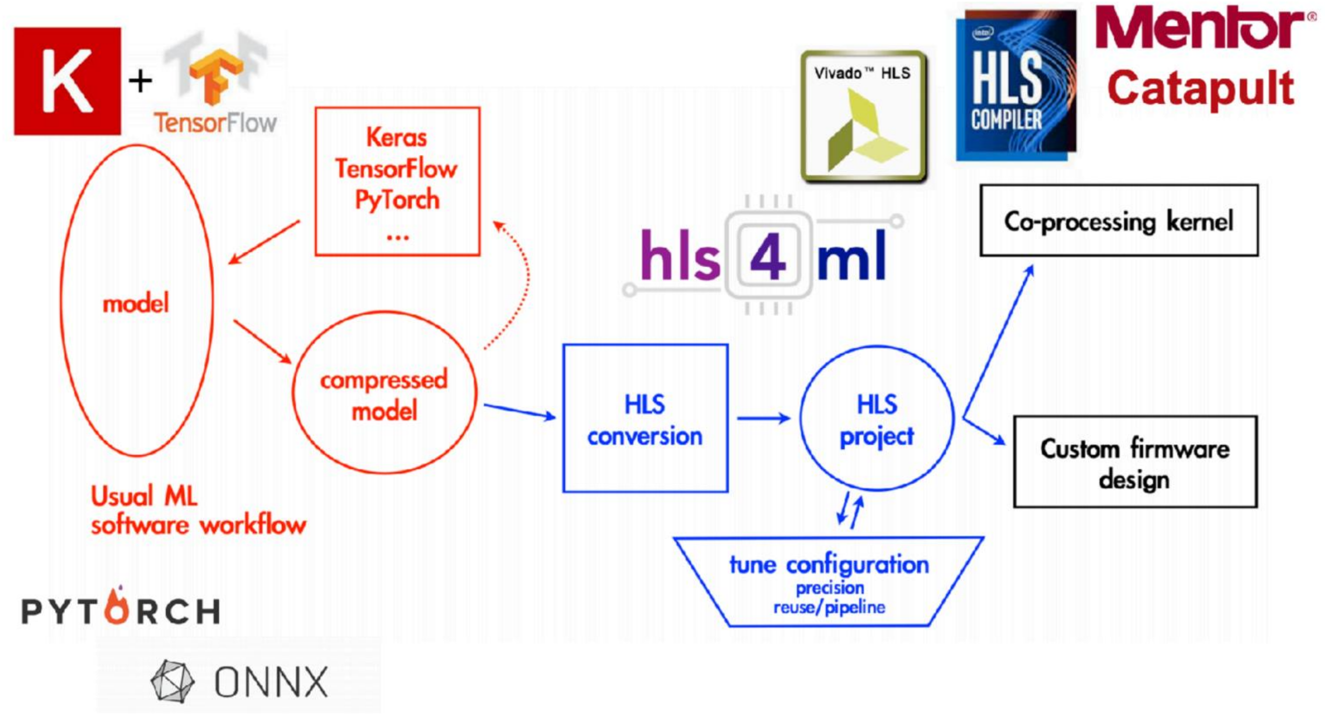
For the latest status including current and planned features, see the Status and Features page.

Tutorials

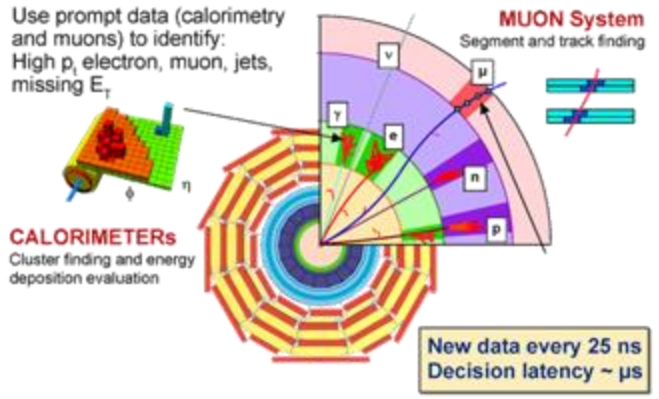
Detailed tutorials on how to use `hls4ml`'s various functionalities can be found here.

- hls4ml is a software package for automatically creating implementations of neural networks for FPGAs and ASICs
- Supports common layer architectures and model software (keras, tensorow, pytorch, ONNX)
- pip installable
- arXiv:1804.06913

hls4ml Workflow

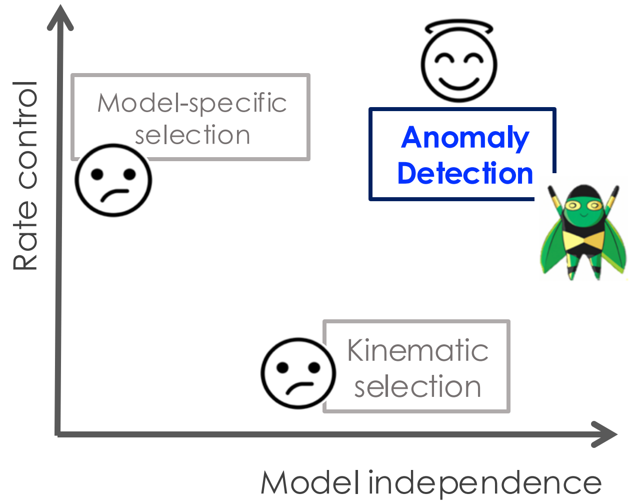


Machine Learning at Level-1 Trigger


Use prompt data (calorimetry and muons) to identify:
High p_t electron, muon, jets, missing E_T
MUON System
Segment and track finding
CALORIMETERs
Cluster finding and energy deposition evaluation
New data every 25 ns
Decision latency ~ µs

Traditional event selection at L1 based on object thresholds

➢ High-level and Data analysis selections limited to use those objects



ML decisions based on level-1 inputs themselves

- **Minimize human bias, completely data-driven**
- **ML can unearth unknown and complex correlation**
- **New physics searches in model-independent way**

CICADA

Calorimeter Image Convolutional Anomaly Detection Algorithm

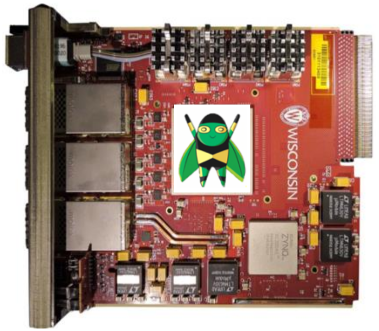
https://cicada.web.cern.ch/

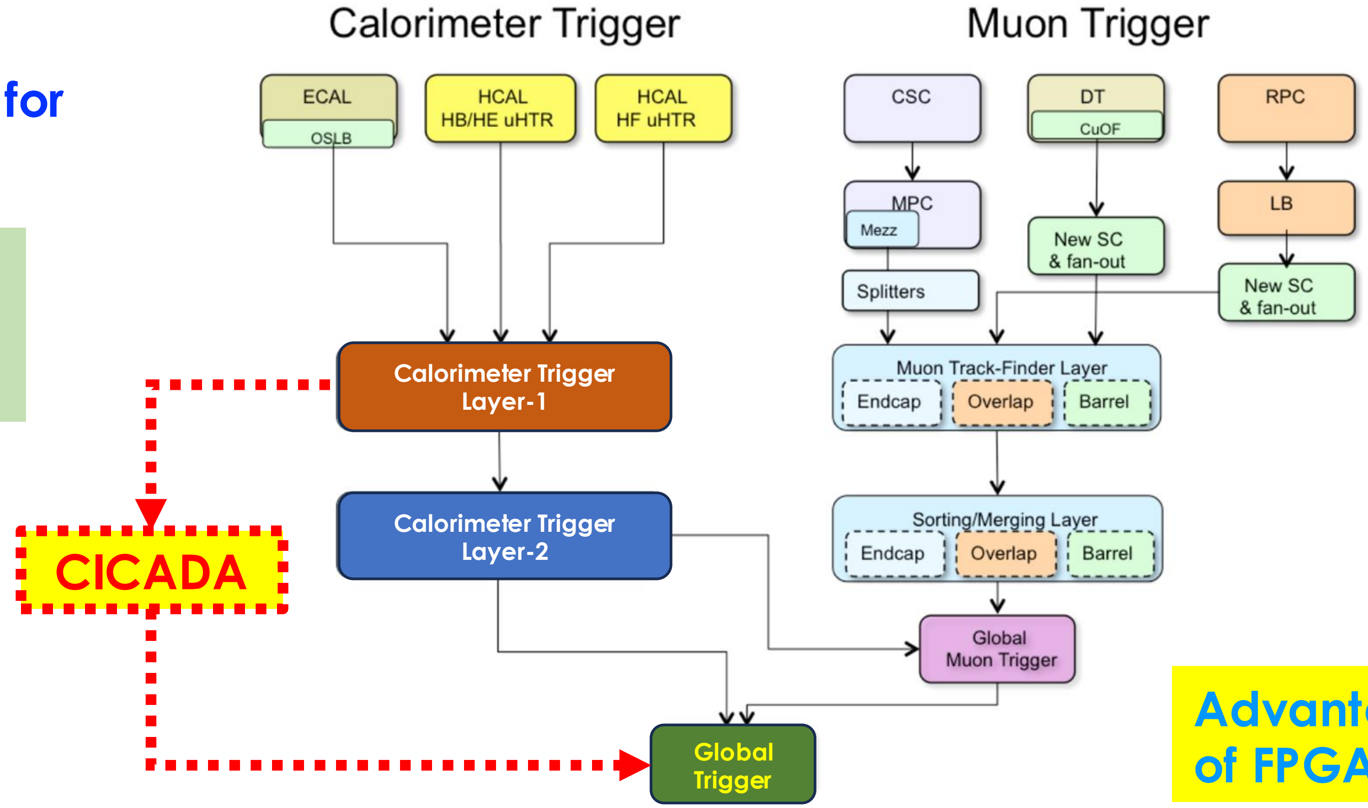
CMS-DP-2023-086

CICADA: New Addition in Run-3

ML-Based Improvements for Run-3

Currently taking Physics data @ CMS





Advantage of FPGA

CICADA: Inputs

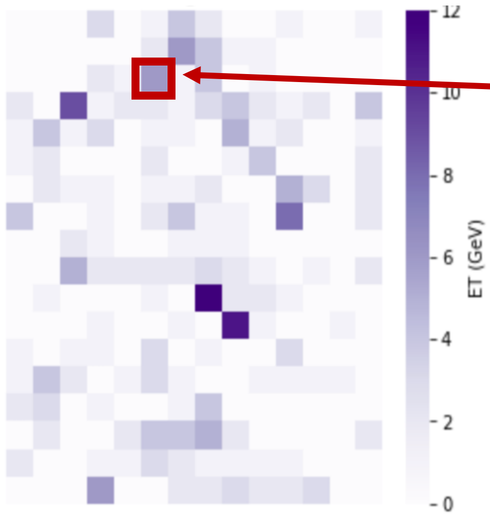
Anomaly Detection Algorithm to Select ~un-biased events for new physics searches

CICADA Inputs from CALO Layer-1

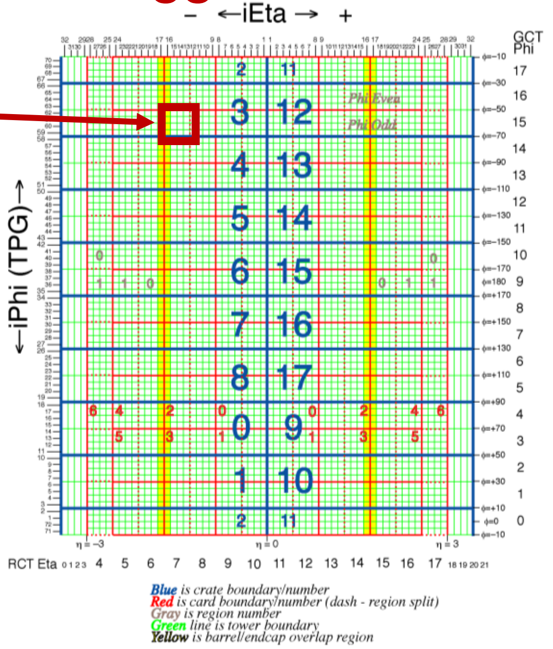
- 18 ϕ x 14 η regions, 252 regions in total
- Each region contains energy deposits from both ECAL and HCAL
- Summary of the energy distribution profile within the region

- Low level information not dependent on object reconstructions

One region = 4x4 trigger towers



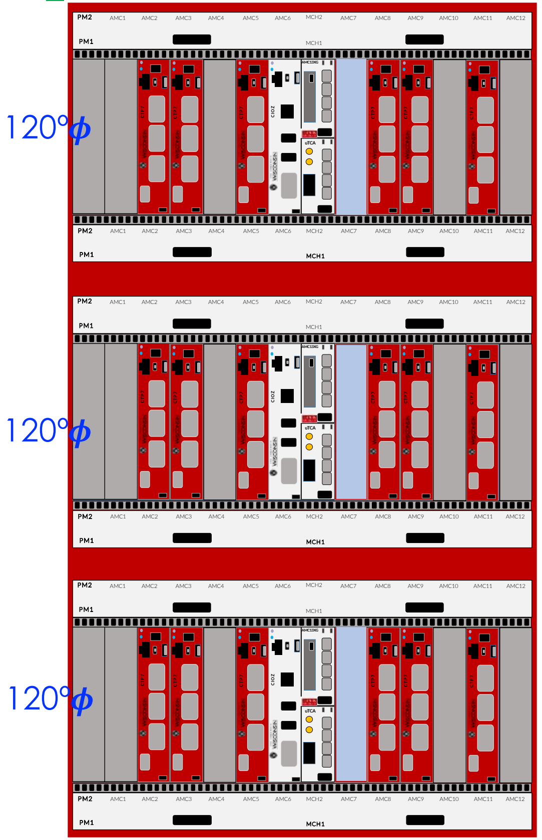
Calorimeter E_T deposit from One ZeroBias event



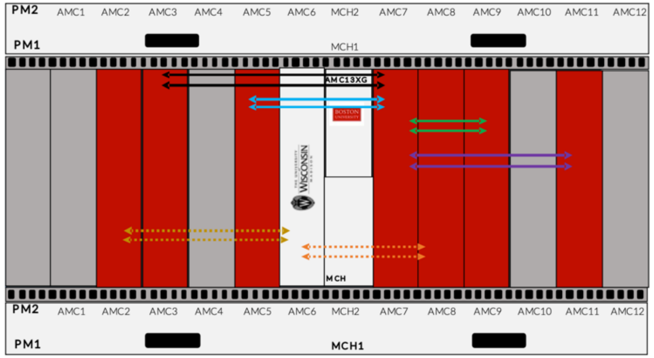
– ←iEta → +
GCT Phi
←iPhi (TPG)→
RCT Eta 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
Phi Even
Phi Odd
Blue is crate boundary/number
Red is card boundary/number (dash - region split)
Gray is region number
Green line is tower boundary
Yellow is barrel/endcap overlap region

Calo tower region map

CaloL1 Setup

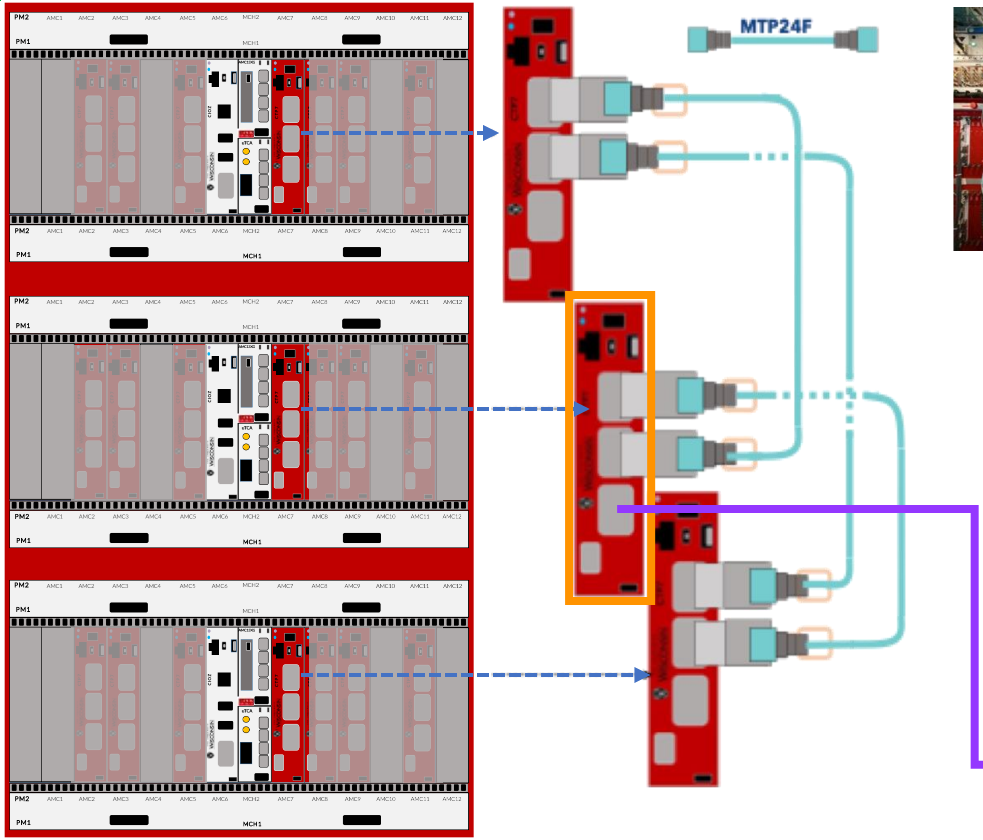


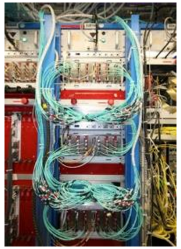
- Calo-Layer 1 Trigger consists of **3-μTCA crates** each equipped with **6-CTP7 cards**
- Each CTP7 cards receive information from the calorimeters (HCAL, ECAL, HF) and send calibrated E+H & E/H to next lyare

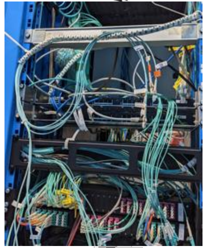


Backplane links

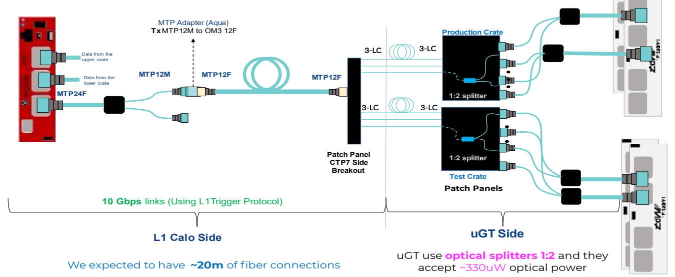
CICADA: Layer-1 to uGT







CICADA to uGT Fiber Path (Block Diagram Simplified)



We expected to have **~20m** of fiber connections

uGT use **optical splitters 1:2** and they accept ~330uW optical power

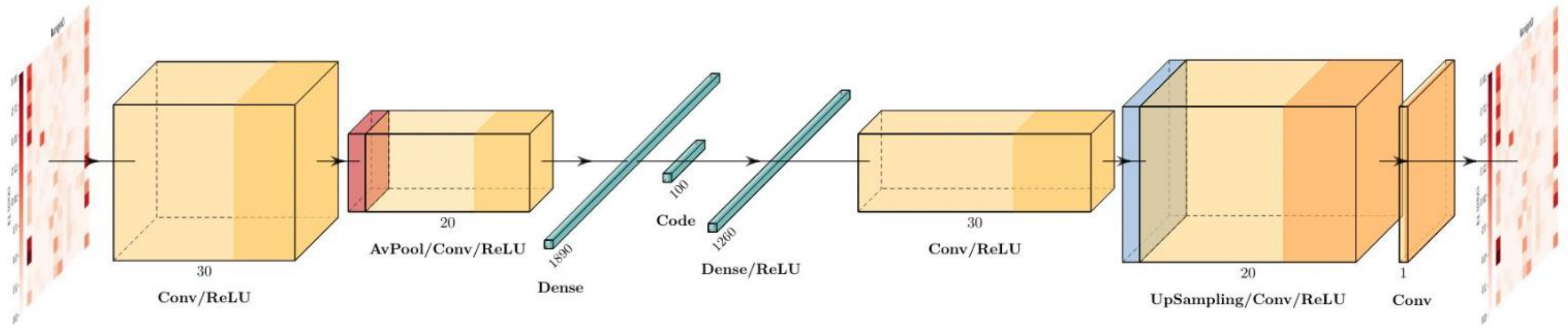
All data is collected in one card '<u>Summary Card</u>'

LC fibres

Global Trigger

CICADA: Auto-encoder Model

Model architecture: calo input → encoder → latent space → decoder → reconstructed input

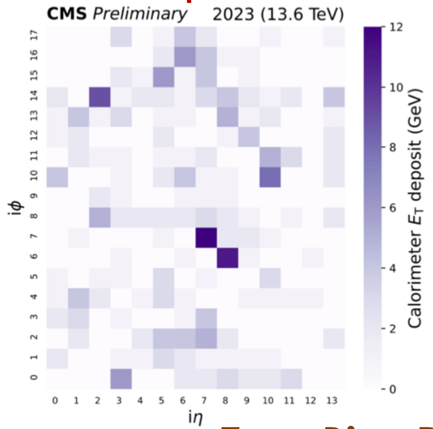


Autoencoder-based ***anomaly*** detection

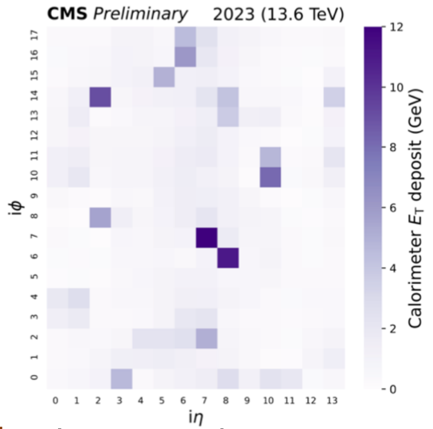
- Input is a 2D tensor from the Calo region energy information
- Encoder and decoder are Convolutional Neural Networks
- ***Unsupervised*** learning : train only on ZeroBias data to learn input reconstruction

CICADA: Event Reconstruction

Input

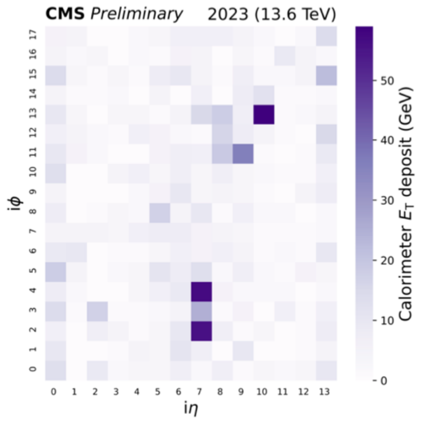


Reconstruction

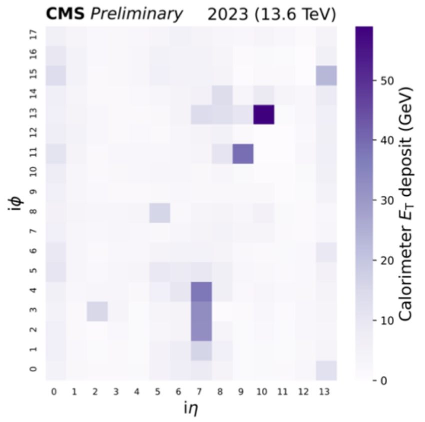


Zero Bias Data (Loss: 0.8)

Input



Reconstruction

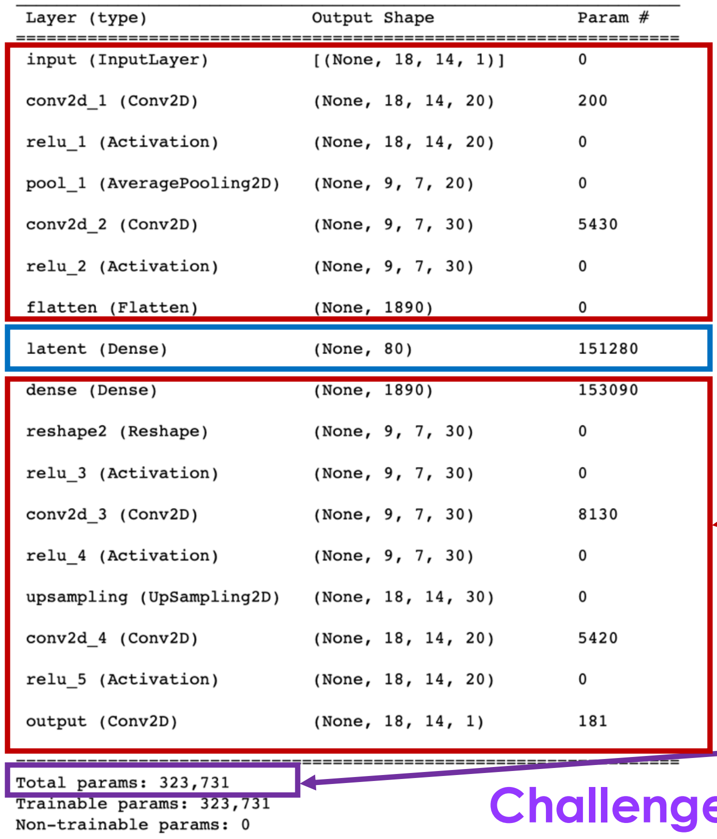


BSM Simulated Signal (Loss: 14.2)

Expectation:
- *Good reconstruction* on **normal events** (ZeroBias used for training)
- *Bad reconstruction* on anything else such as **BSM signals** (never seen during training)

Goal:
- **Anomaly Score:** Mean Squared Error, **MSE(input, output)**

CICADA: Naive Auto-encoder Model



```
Layer (type)                 Output Shape              Param #
===============================================================
input (InputLayer)           [(None, 18, 14, 1)]       0

conv2d_1 (Conv2D)            (None, 18, 14, 20)        200

relu_1 (Activation)          (None, 18, 14, 20)        0

pool_1 (AveragePooling2D)    (None, 9, 7, 20)          0

conv2d_2 (Conv2D)            (None, 9, 7, 30)          5430

relu_2 (Activation)          (None, 9, 7, 30)          0

flatten (Flatten)            (None, 1890)              0

latent (Dense)               (None, 80)                151280

dense (Dense)                (None, 1890)              153090

reshape2 (Reshape)           (None, 9, 7, 30)          0

relu_3 (Activation)          (None, 9, 7, 30)          0

conv2d_3 (Conv2D)            (None, 9, 7, 30)          8130

relu_4 (Activation)          (None, 9, 7, 30)          0

upsampling (UpSampling2D)    (None, 18, 14, 30)        0

conv2d_4 (Conv2D)            (None, 18, 14, 20)        5420

relu_5 (Activation)          (None, 18, 14, 20)        0

output (Conv2D)              (None, 18, 14, 1)         181

===============================================================
Total params: 323,731
Trainable params: 323,731
Non-trainable params: 0
```

Encoder(compressor)

Latent space (compressed input)

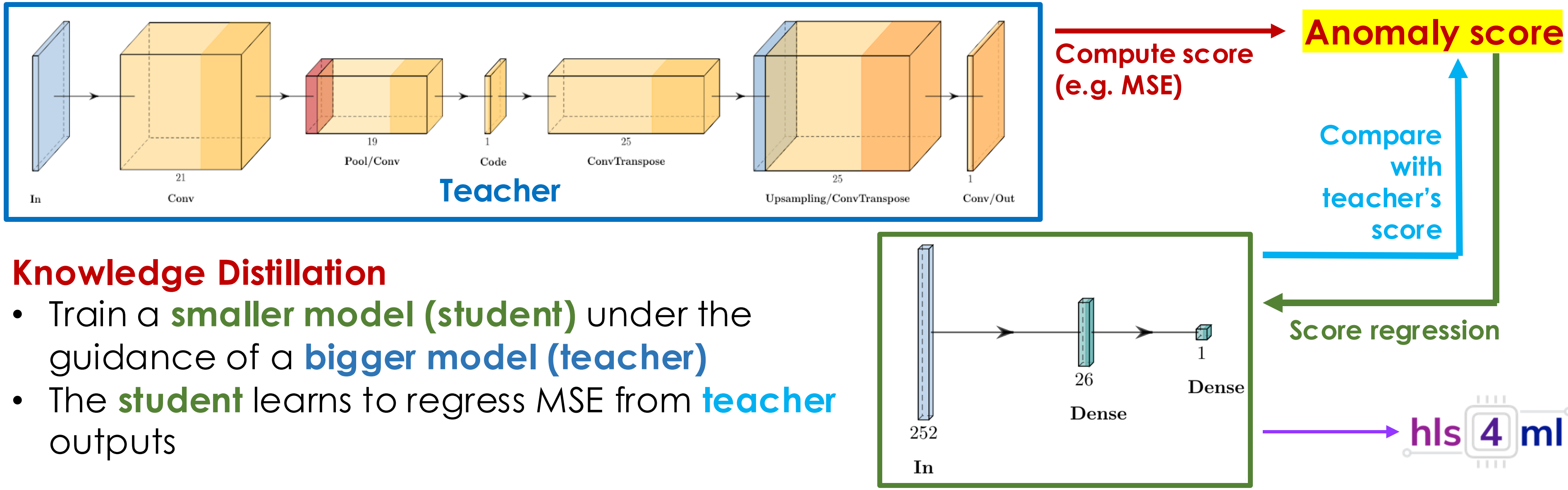
Decoder (decompressor)

Challenges!

324 K parameters model size

can't fit L1 constraints…

Knowledge Distillation + Quantization



Knowledge Distillation

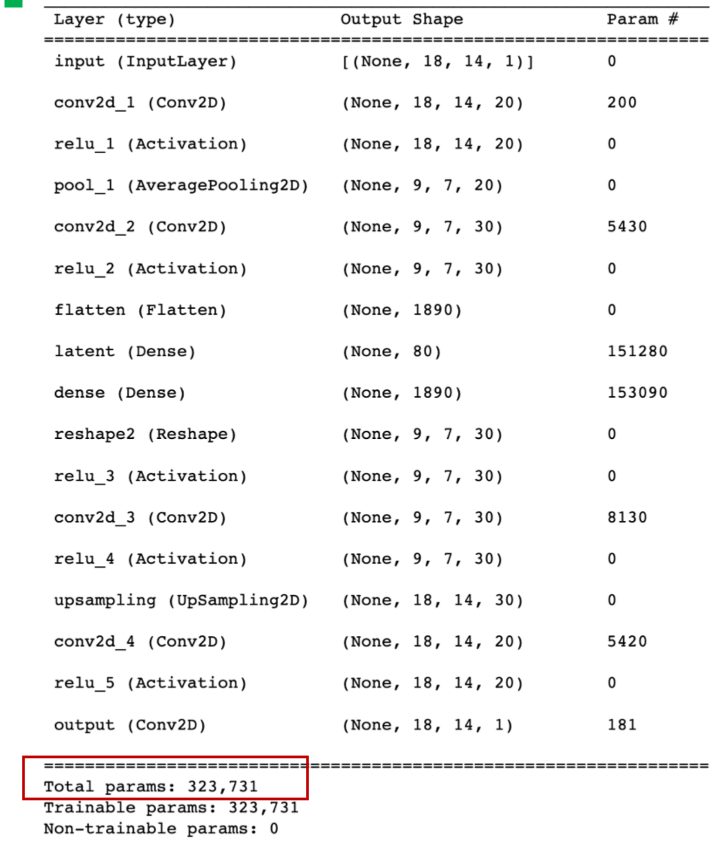
- Train a **smaller model (student)** under the guidance of a **bigger model (teacher)**
- The **student** learns to regress MSE from **teacher** outputs

Quantization-aware training (QKeras)

- Model weights quantized to fixed precision (e.g., 2 bits for integer, 4 bits for fraction)
- *Train a quantized model rather than quantize a trained model*

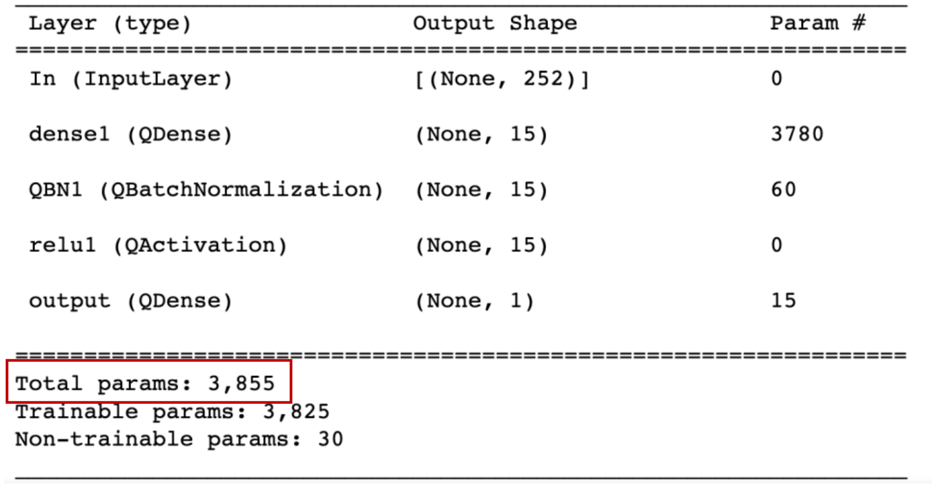
→ x10 reduction in resources/latency

CICADA: Teacher → Student Model



```
Layer (type)                Output Shape              Param #
=================================================================
input (InputLayer)          [(None, 18, 14, 1)]       0

conv2d_1 (Conv2D)           (None, 18, 14, 20)        200

relu_1 (Activation)         (None, 18, 14, 20)        0

pool_1 (AveragePooling2D)   (None, 9, 7, 20)          0

conv2d_2 (Conv2D)           (None, 9, 7, 30)          5430

relu_2 (Activation)         (None, 9, 7, 30)          0

flatten (Flatten)           (None, 1890)              0

latent (Dense)              (None, 80)                151280

dense (Dense)               (None, 1890)              153090

reshape2 (Reshape)          (None, 9, 7, 30)          0

relu_3 (Activation)         (None, 9, 7, 30)          0

conv2d_3 (Conv2D)           (None, 9, 7, 30)          8130

relu_4 (Activation)         (None, 9, 7, 30)          0

upsampling (UpSampling2D)   (None, 18, 14, 30)        0

conv2d_4 (Conv2D)           (None, 18, 14, 20)        5420

relu_5 (Activation)         (None, 18, 14, 20)        0

output (Conv2D)             (None, 18, 14, 1)         181

=================================================================
Total params: 323,731
Trainable params: 323,731
Non-trainable params: 0
```

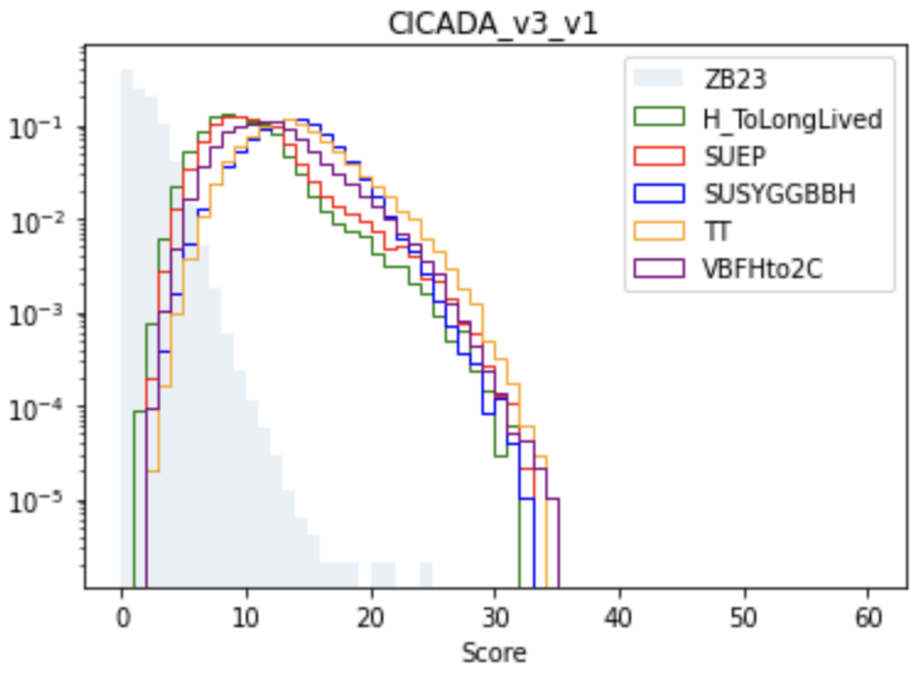
Student

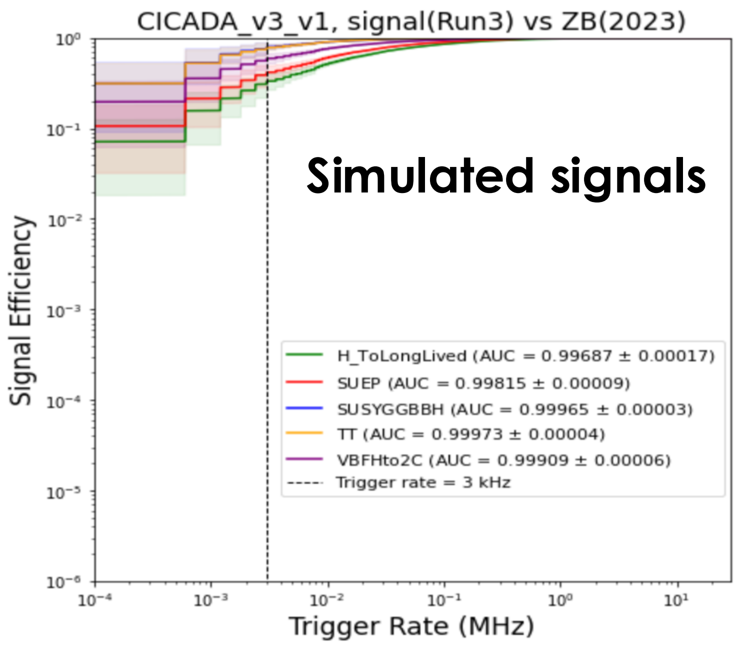


```
Layer (type)                    Output Shape              Param #
=================================================================
In (InputLayer)                 [(None, 252)]             0

dense1 (QDense)                 (None, 15)                3780

QBN1 (QBatchNormalization)      (None, 15)                60

relu1 (QActivation)             (None, 15)                0

output (QDense)                 (None, 1)                 15

=================================================================
Total params: 3,855
Trainable params: 3,825
Non-trainable params: 30
_________________________________________________________________
```

324K parameters go down to 3.8K parameters

CICADA: Physics Performance

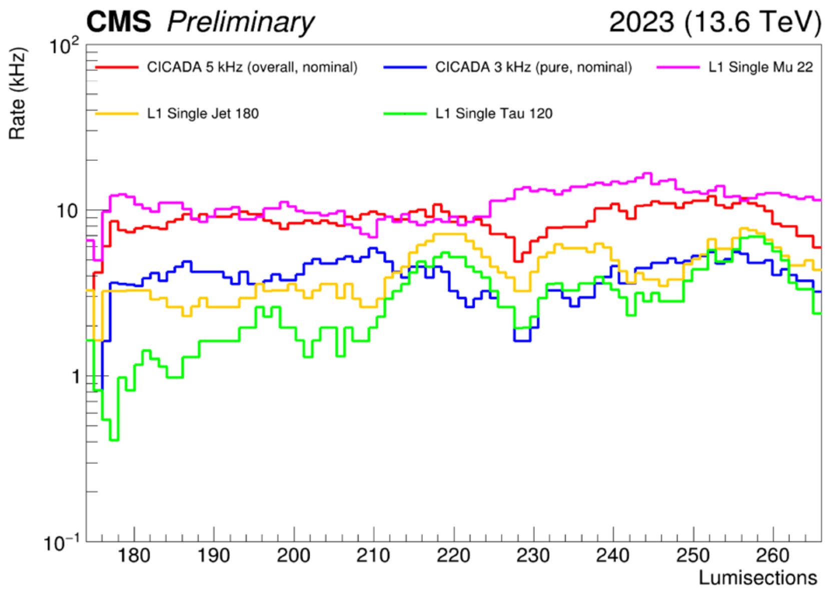


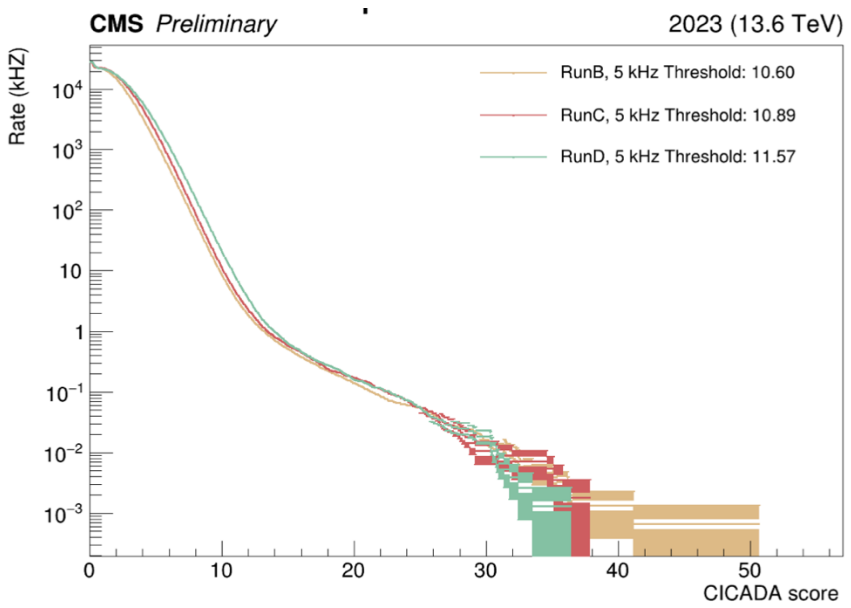


Simulated signals

- **Model trained on 2023 ZB, evaluated on 2023 Simulated signals**
- **Able to pick up a wide range of BSM signals**

CICADA: Rate Stability

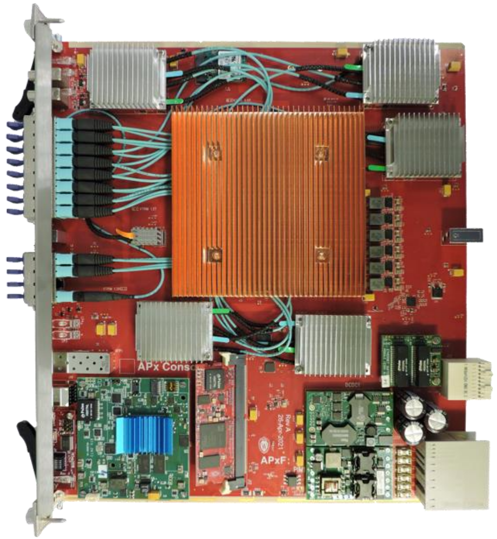




A Flexible trigger: tunable threshold for different rates, stable over the run

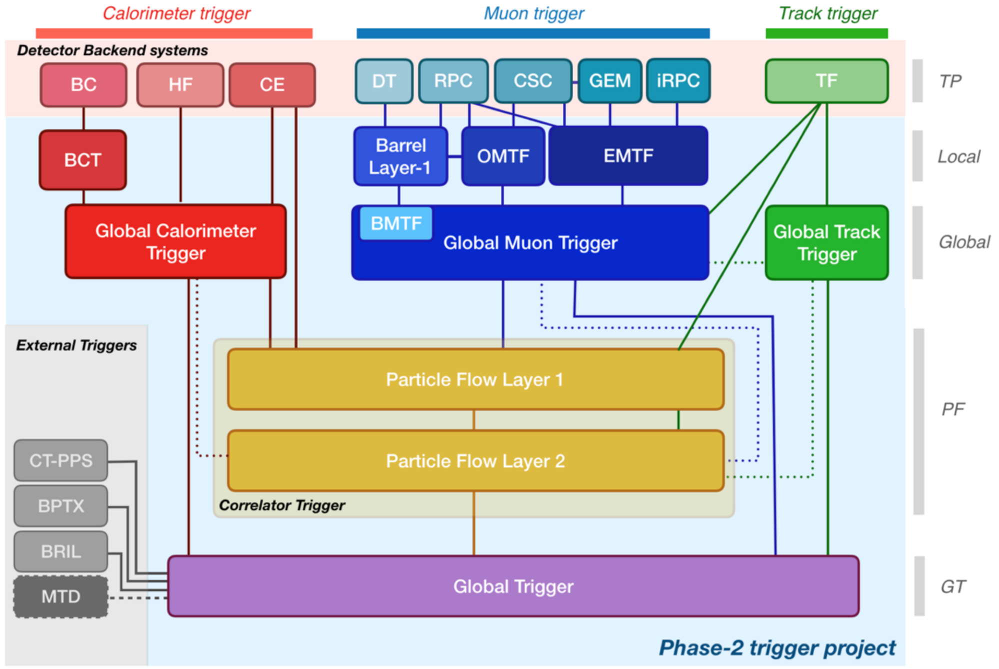
HL-LHC: Can be more adventurous

Wisconsin APxF Board



- Xilinx VU13P FPGA
- 25G Samtec Firefly optics (124 25 Gbps links)

CMS Upgrade to Level-1 Trigger



More resources available to implement ML based triggers

Concluding Remarks

- We in HEP may not be the pioneers of modern electronics technologies, but we are among those who drive their advancement most aggressively
- Progress in telecommunications and field-programmable logic devices is constantly leveraged to manage the growing demands of data processing
- A collaborative team of engineers and physicists has mastered the challenge of handling the massive data output from the LHC, using advanced telecommunications and field-programmable logic devices to facilitate groundbreaking discoveries in fundamental physics
- With advances in ML and FPGAs, more complex models can be implemented in future

ధన్యవాదాలు

धन्यवाद

Thank you

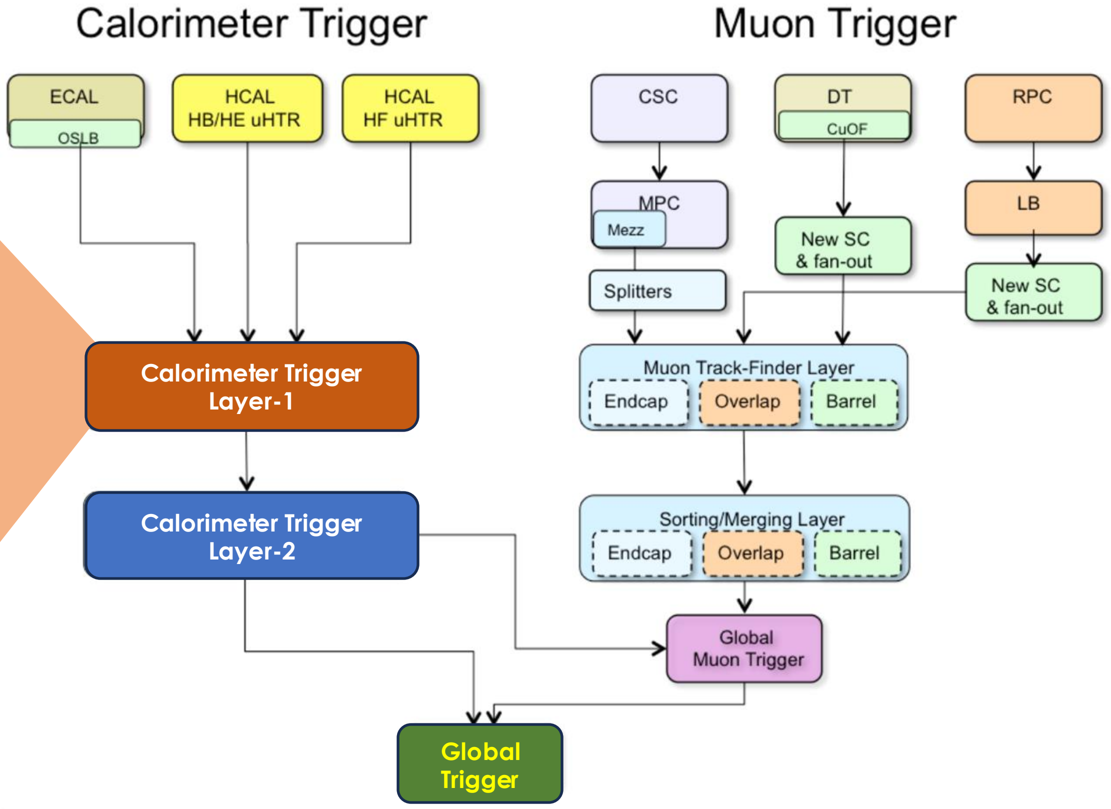
Extra Slides

Jargons

- **ICs - Integrated chip:** assembly of hundreds of millions of transistors on a minor chip
- **PCB:** Printed Circuit Board
- **LUT - Look Up Table aka 'logic'** - generic functions on small bitwidth inputs. Combine many to build the algorithm
- **FF - Flip Flops** - control the flow of data with the clock pulse. Used to build the pipeline and achieve high throughput
- **DSP - Digital Signal Processor** - performs multiplication and other arithmetic in the FPGA
- **BRAM - Block RAM** - hardened RAM resource. More efficient memories than using LUTs for more than a few elements
- **PCIe or PCI-E - Peripheral Component Interconnect Express**: is a serial expansion bus standard for connecting a computer to one or more peripheral devices
- **InfiniBand** is a computer networking communications standard used in high-performance computing that features very high throughput and very low latency
- **HLS** - High Level Synthesis - compiler for C, C++, SystemC into FPGA IP cores
- **HDL** - Hardware Description Language - low level language for describing circuits
- **RTL** - Register Transfer Level - the very low level description of the function and connection of logic gates
- **Latency** - time between starting processing and receiving the result
  - Measured in clock cycles or seconds
- **II - Initiation Interval** - time from accepting first input to accepting next input

CMS Level-1 Trigger

Wisconsin CTP7 Board
Xilinx's Virtex7 FPGA



What is CICADA ☺

A **“CICADA”** is an insect of the family “Cicadoidea”

- Cicadas are known for their loud vocalizations (typically during summer)

- Much of a cicada’s life cycle is actually spent underground, with a few famous American species (the “periodical cicada”) only emerging every 13 (*magicicada tredecim*) or 17 (*magicicada septendecim*) years

Source: https://kids.nationalgeographic.com/animals/invertebrates/facts/cicada