

Novel Computing Hardwares in HEP

Heather M. Gray



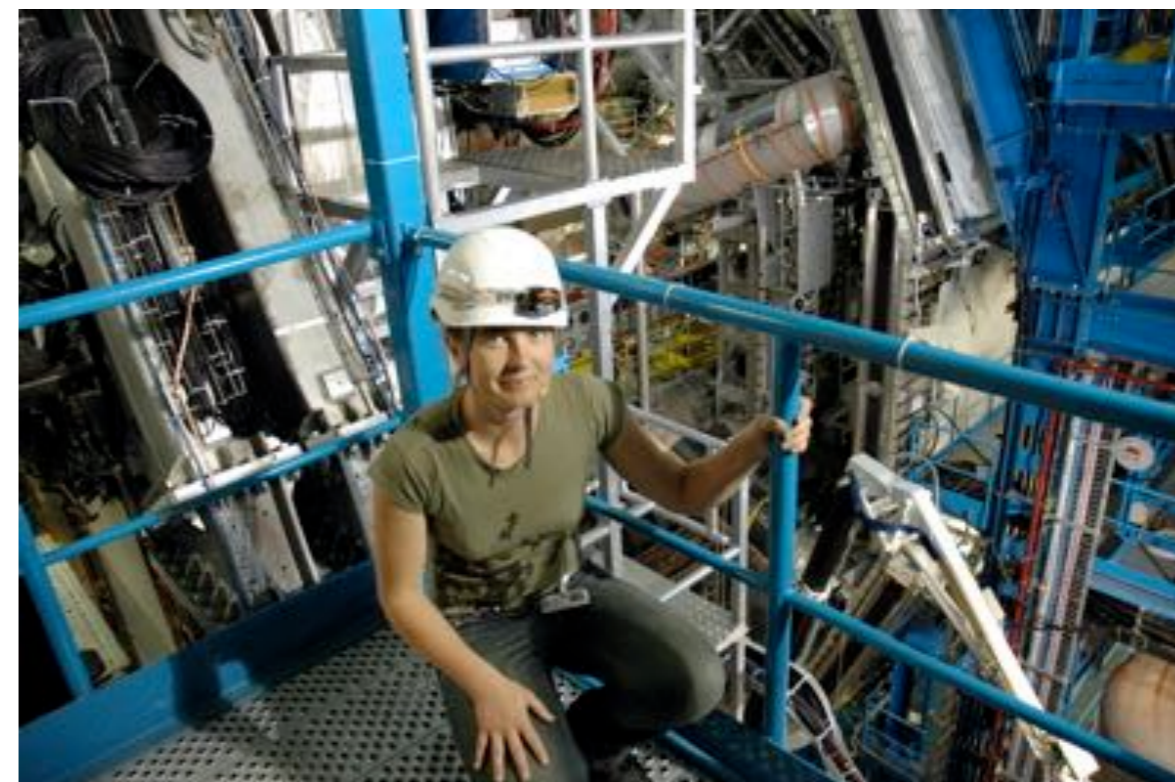
Berkeley
UNIVERSITY OF CALIFORNIA

CHACAL 2024



Introduction

- Associate Professor of Physics at **UC Berkeley** and Faculty Scientist at Lawrence Berkeley Laboratory
- BSc, BSc(Hons), MSc at UCT (2005)
- PhD at Caltech (2011)
- Research Fellow and Staff at CERN
- Member of the **ATLAS experiment** since 2005
 - Tracking Convener
 - Simulation Convener
 - (Currently) Data Preparation Coordinator
- Primary expertise: SM & Higgs physics, tracking, software, ...



Berkeley
UNIVERSITY OF CALIFORNIA

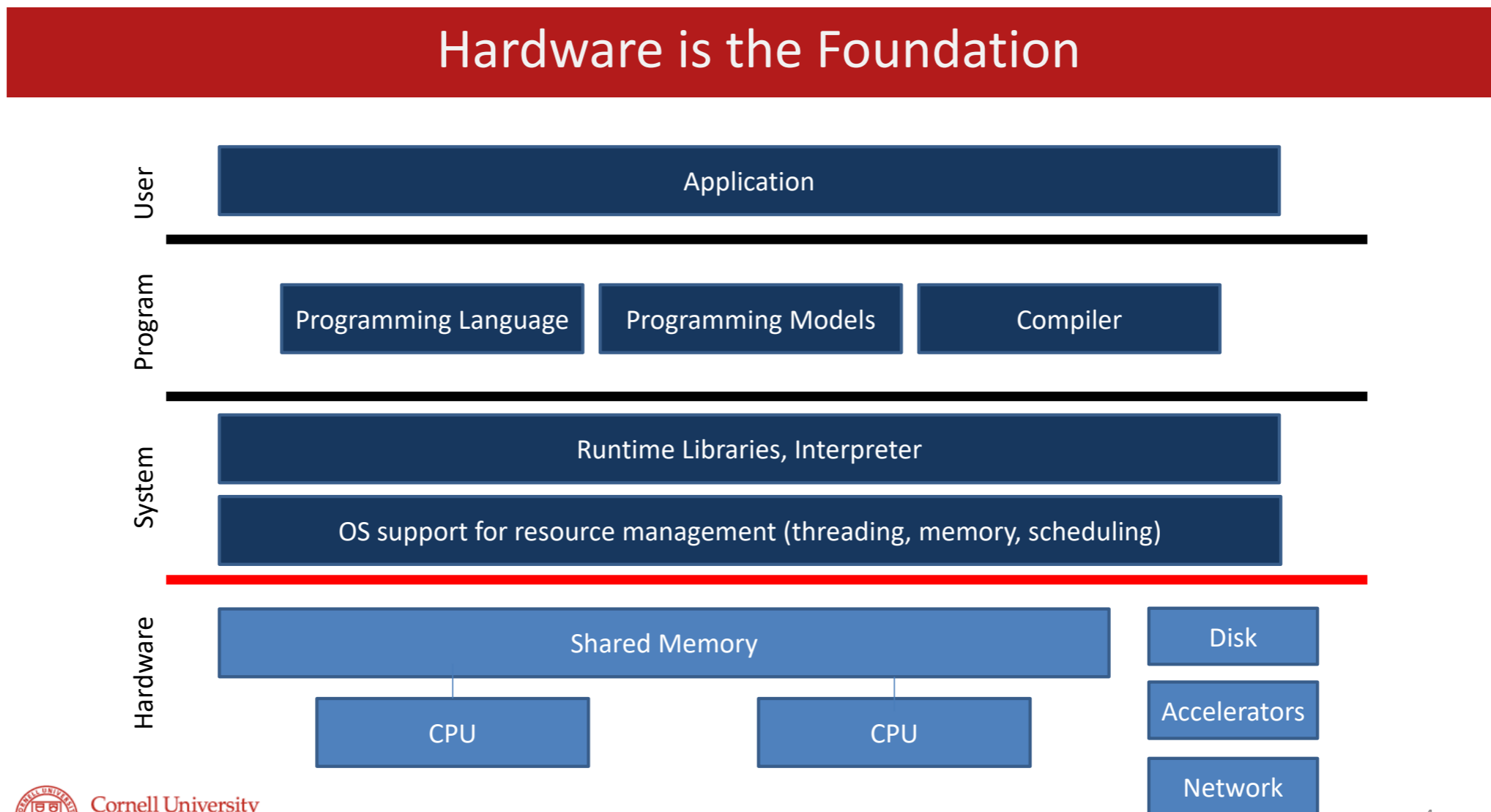


Lecture Outline

- Introduction & HEP Computing Challenge
- **Part 1: Novel Computing Hardware**
 - Hardware Accelerators
 - GPUs
 - FPGAs
 - New Computing Paradigms
 - GPU Programming Taster
- **Part 2: Application to HEP**
 - Event Generation
 - Simulation
 - Reconstruction
 - Trigger
- Conclusion

Hardware

- Lectures at this school have been primarily focused on **software**
- These lectures provide an introduction to the ongoing evolution of **computer hardware** and illustrate how it provides **new opportunities** in HEP



Traditional Computing Architectures

- Computing in HEP currently relies primarily upon large numbers of **Central Processing Units (CPUs)**
- CPUs are silicon-based microprocessors
 - Can perform an extensive variety of tasks
- **Operations** are typically performed **serially**
- **Programs** are sequences of operations processed in a sequential order



Image Credit



Image Credit

microprocessor: logic and control on a single integrated circuit

CPU Components

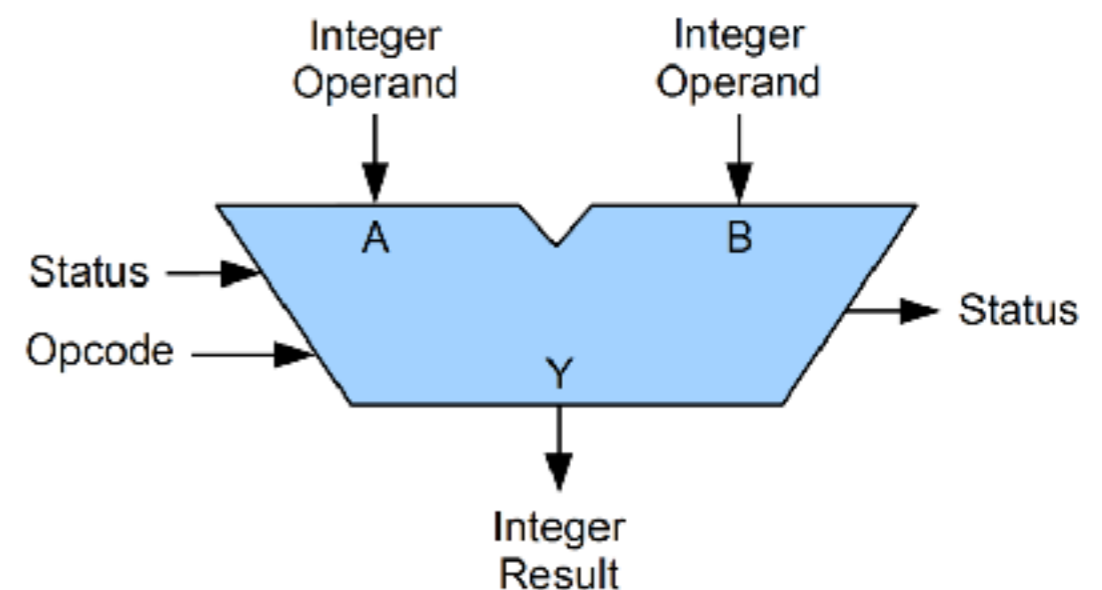
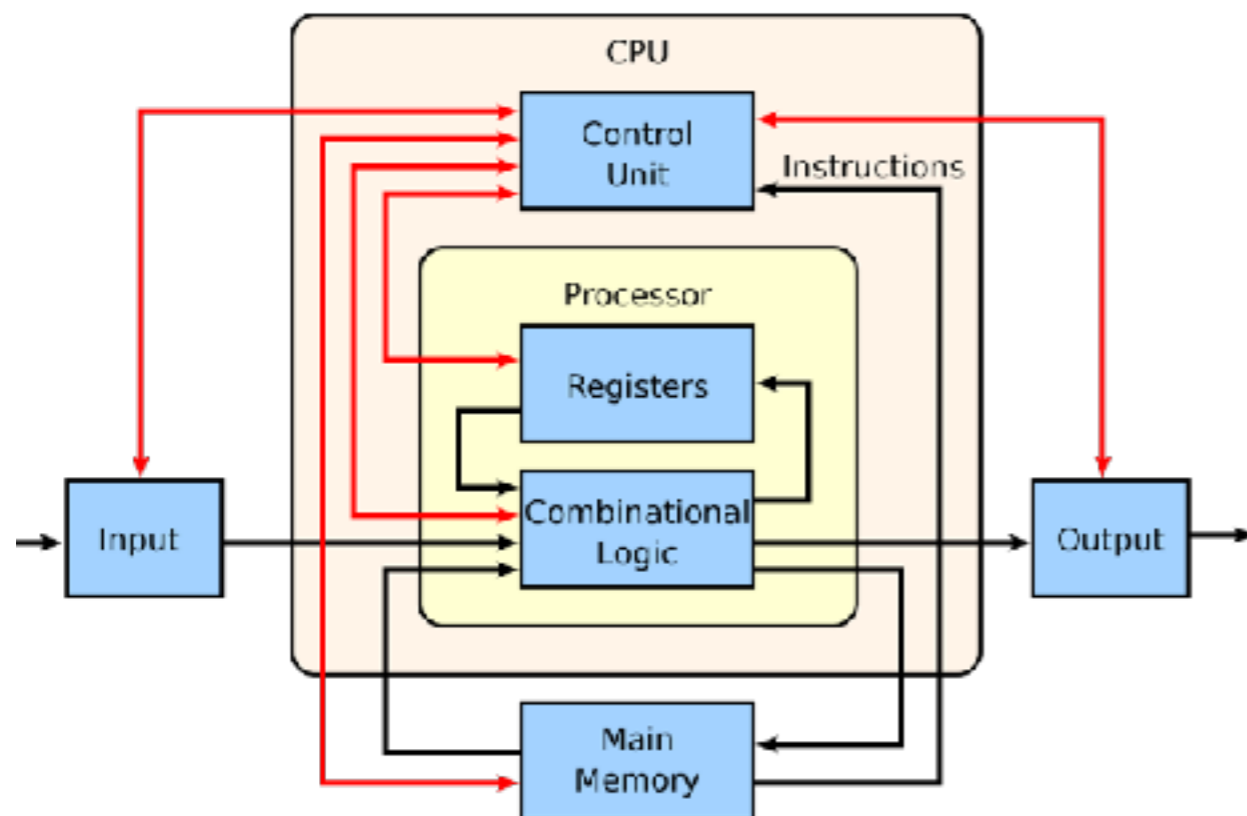
- CPUs consist of three principal components

- **Control Unit**

- Directs the processor operation
- Manages computer resources

- **Arithmetic Logic Unit (ALU)**

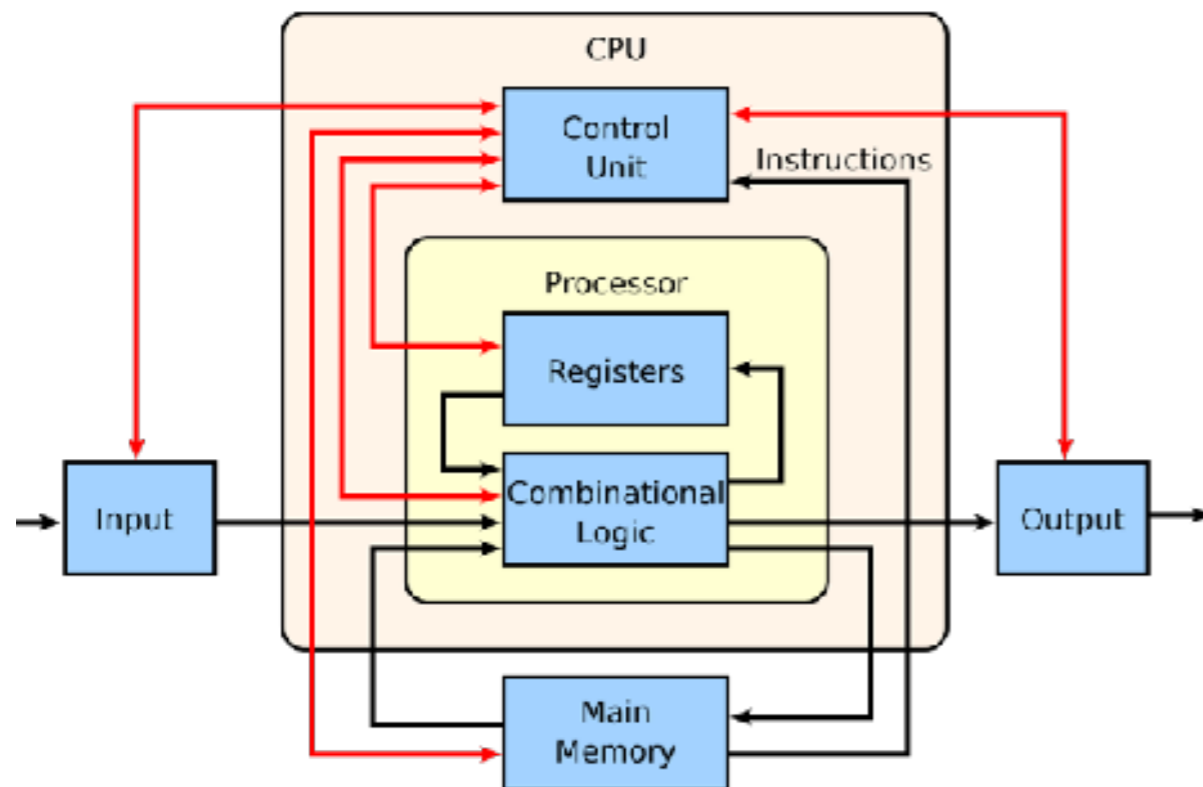
- Performs integer arithmetic and logical operations



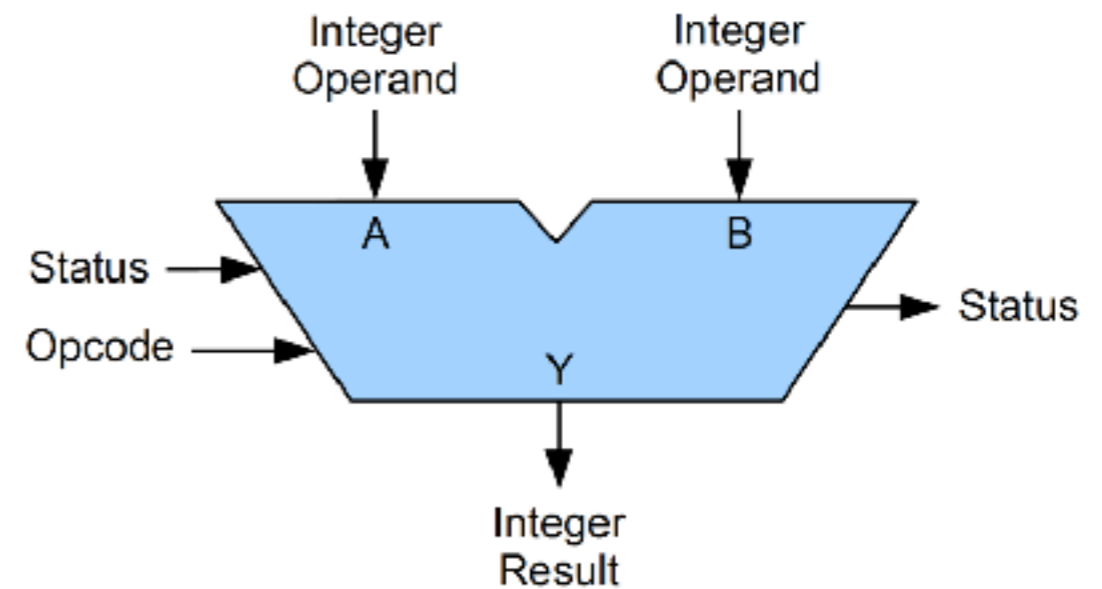
CPU Components

- **Registers**

- Rapidly accessible storage locations accessible to processors
 - Supplies operands and stores results from ALU
- CPUs are implemented on **integrated circuit (IC) microprocessors**
 - Each IC chip can have one or multiple CPUs (known as cores)
 - Multi-core processors are chips with multiple CPU
 - Multithreaded cores can be used to make virtual CPUs



Source



Source

CPU Architectures: x86 and ARM

- Most personal computers sold today have CPUs based on the **x86** architecture
 - x86 is a set of **complex instruction set computers (CISC)** initially developed by Intel
- Most smartphones and tablets use **ARM (Advanced RISC machines)**
 - **RISC**: Reduced instruction set computers
- Reason: CISC can handle more **complex tasks** and **calculations** but RISC has better **power efficiency**
- Both are closed source owned by Intel and ARM respectively

Moore's Law

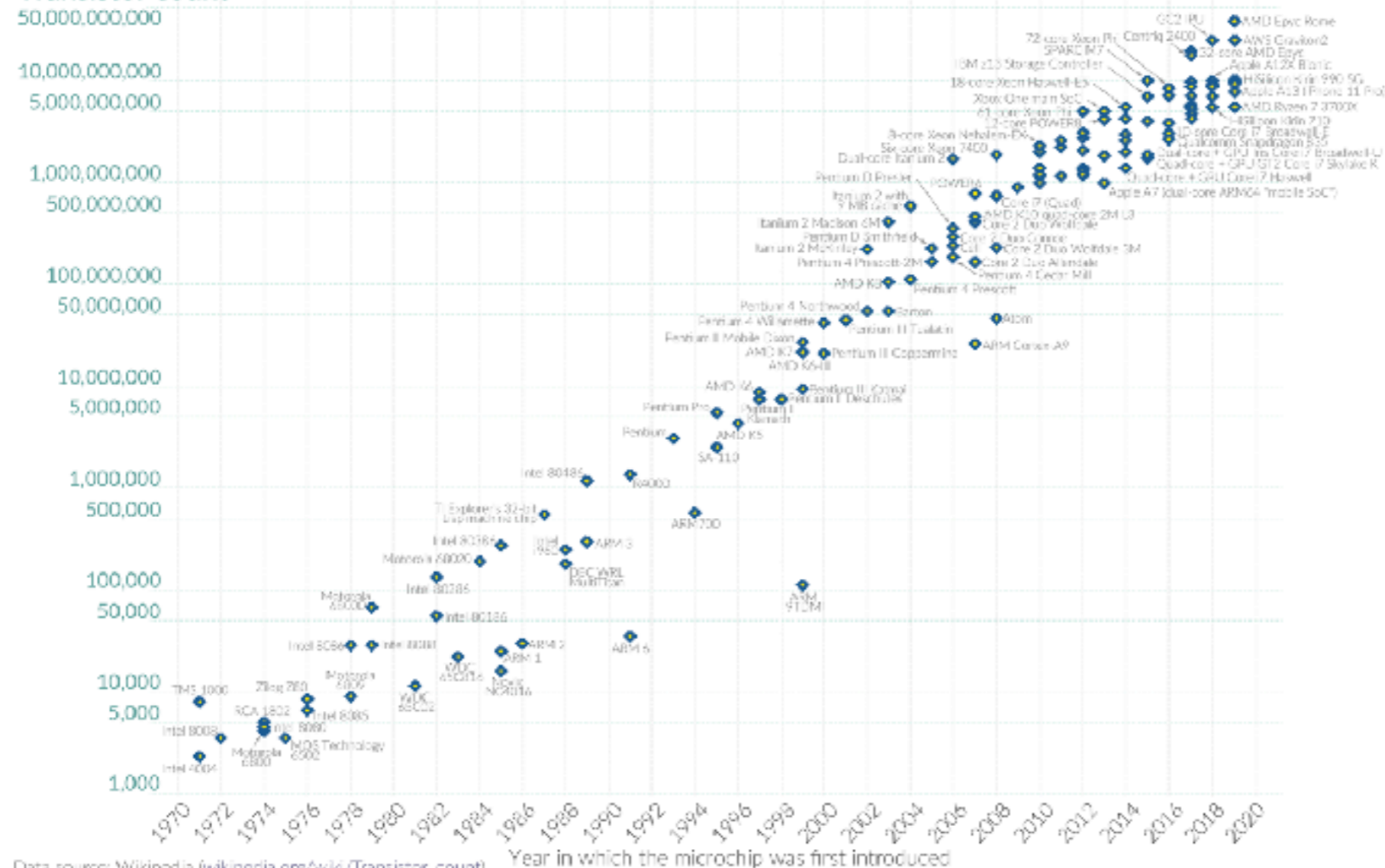
- **Moore's law:** the number of transistors in an integrated circuit doubles approximately every two years
- Observation named after Gordon Moore (founder of Intel)

Moore's Law: The number of transistors on microchips doubles every two years

Our World
in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Transistor count



Data source: Wikipedia ([wikipedia.org/wiki/Transistor_count](https://en.wikipedia.org/wiki/Transistor_count))

OurWorldinData.org – Research and data to make progress against the world's largest problems.

Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.

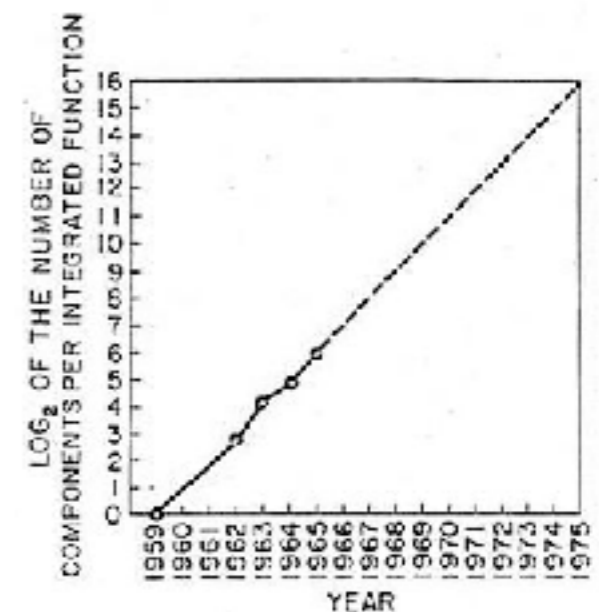
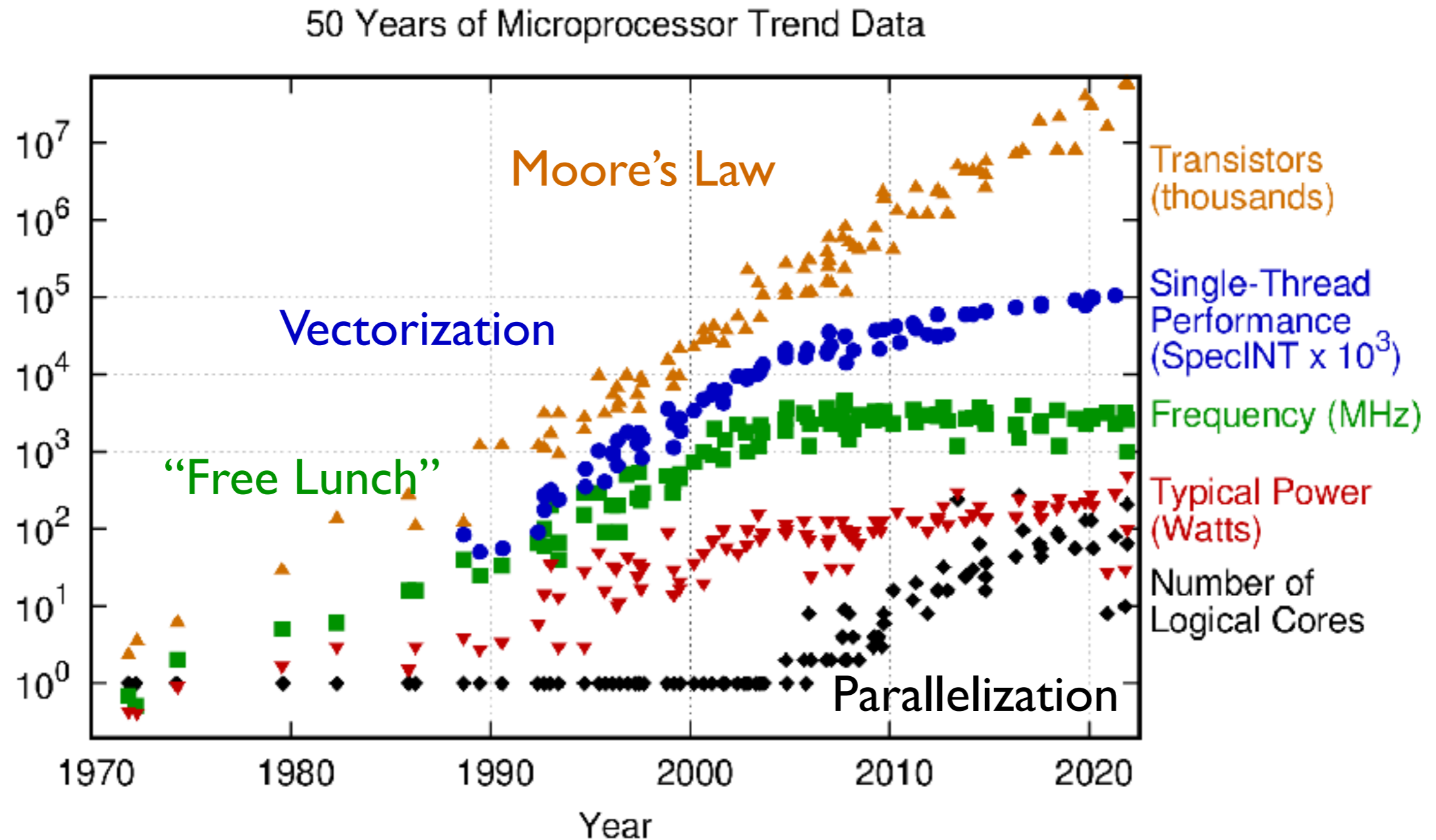


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

Source

Moore's Law

This simple observation hides a more complex reality



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
 New plot and data collected for 2010-2021 by K. Rupp

Image Credit

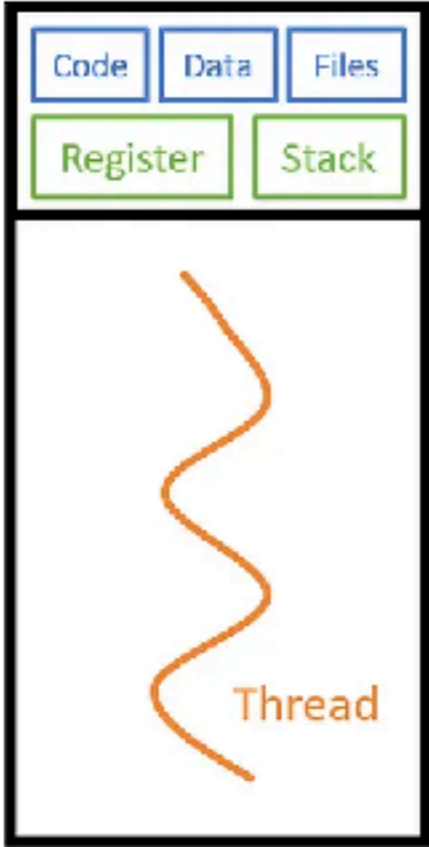
Slides after

Vectorization

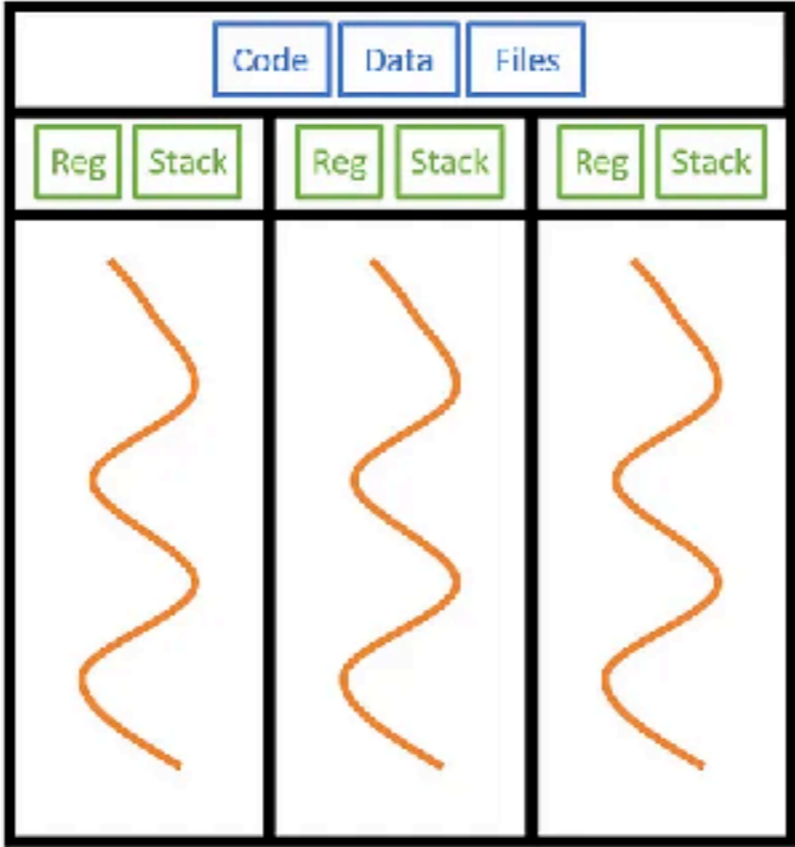
- Most modern processors provide **vectorization**
 - Also known as Single Instruction Multiple Data (**SIMD**)
 - Data is stored in vector registers
- Instructions must be executed in **lockstep**
 - No synchronization is needed
- Multithreading used threads instead of vectors
 - Also known as Single Instruction Multiple Threads (**SIMT**)
- Synchronization is required between threads as the execution time can vary

Categorization introduced in the 1960s by Flynn (and then modified by others)

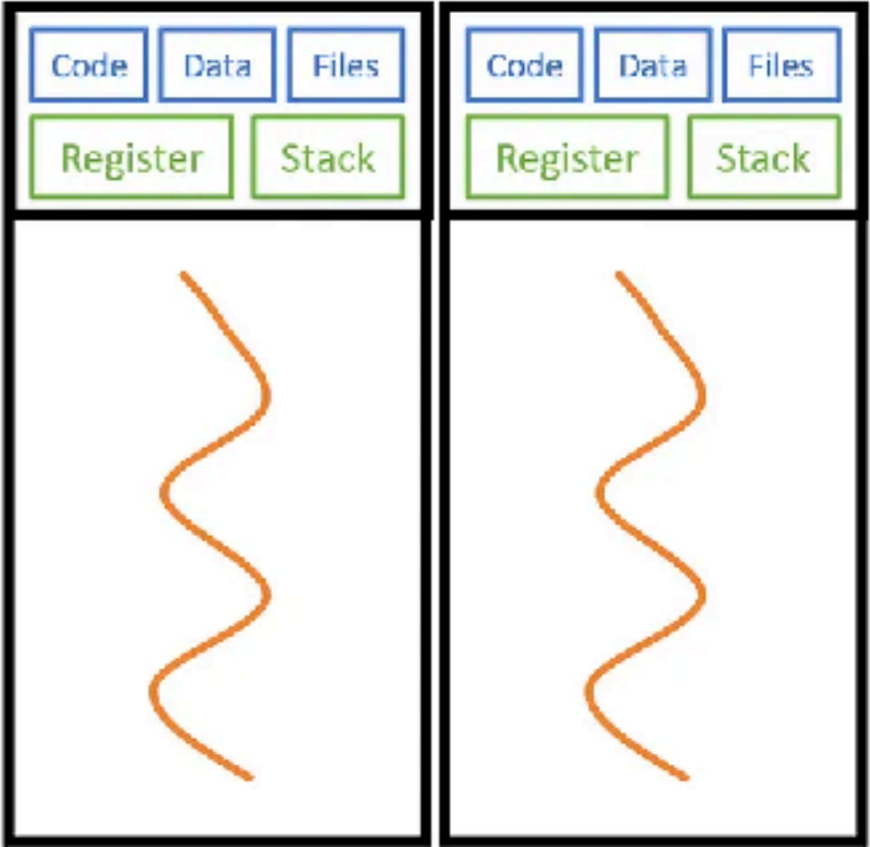
Parallelism Strategies



Single Processor Single Thread



Single Processor Multithread



Multiprocessing

Image Credit

Server processor evolution

	Irwin-dale	Wood-crest	Gaines-town	Haswell	Broad- well	Skylake	Ice Lake	AMD Epyc
Year	2005	2006	2009	2015	2016	2017	2021	2022
Cores	1	2	4	18	24	28	40	64 (128 SMT)
Freq (GHz)	3.8	3.0	3.33	2.1	2.2	2.5	2.3 (3.4 boost)	2.2 (3.5 boost)
LL Cache	L2 (2MB)	L2 (4MB)	L3 (8MB)	L3 (45MB)	L3 (60MB)	L3 (38MB)	L3 (60MB)	L3 (768MB)

Evolution of server processors
<https://ark.intel.com> / <https://amd.com>

Image Credit

Computing In HEP

- As you've seen these past two weeks, software and computing are used **everywhere** in high-energy physics
- Controls accelerator and detectors, trigger on interesting events, simulate the physics and the detectors, reconstruct the data, perform physics analysis, etc, etc.
- High-energy physics has also been responsible for **driving** many developments in computing, e.g. the world wide web



Source

HEP Computing Challenge

- Historically, HEP has relied on **Moore's Law** to meet computing needs* despite **flat budgets**
- In addition, computing needs are about to **grow rapidly** with the advent of the HL-LHC and will grow **even further** with future colliders
- Computing needs cannot be met through “business as usual”

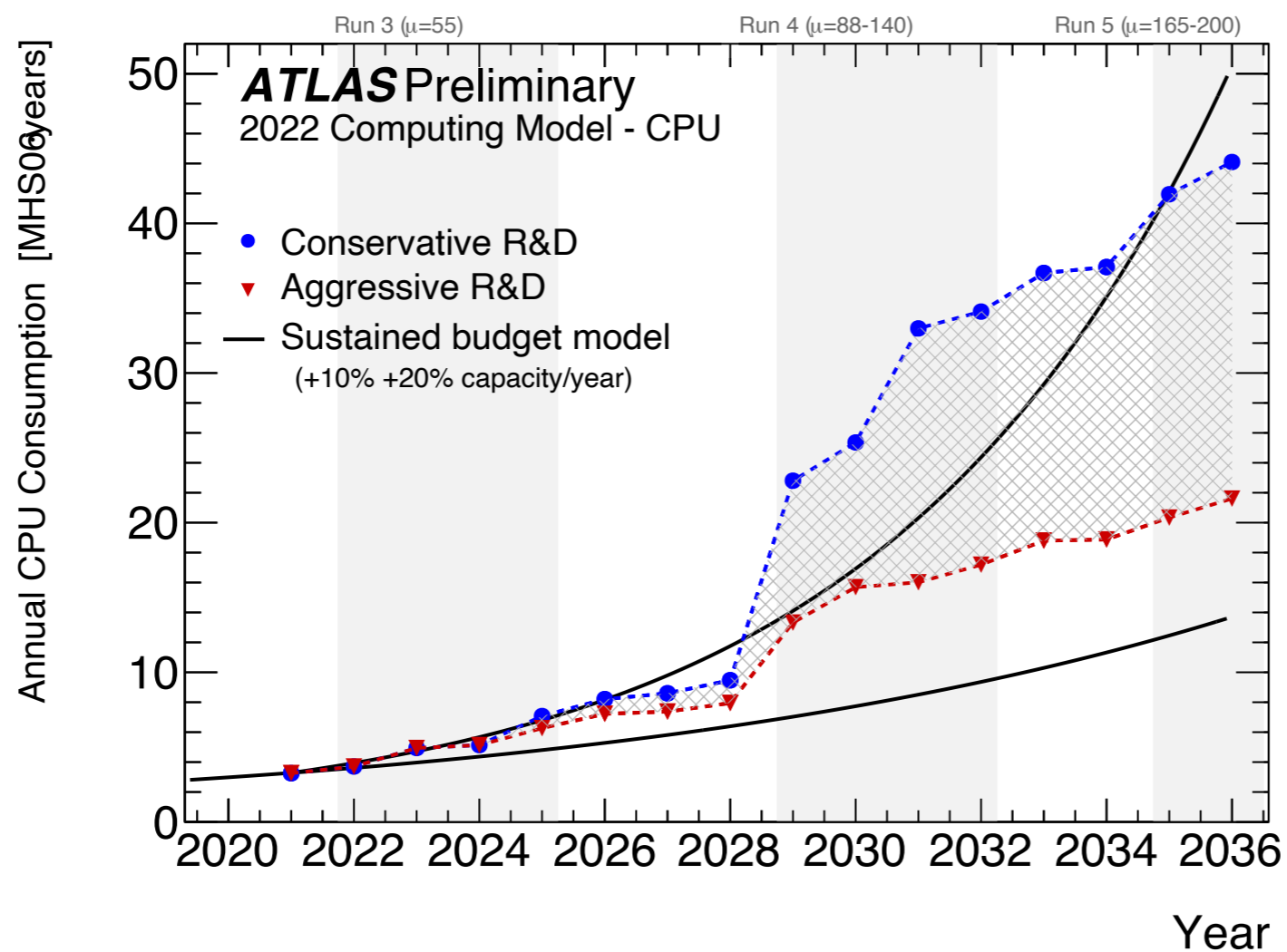


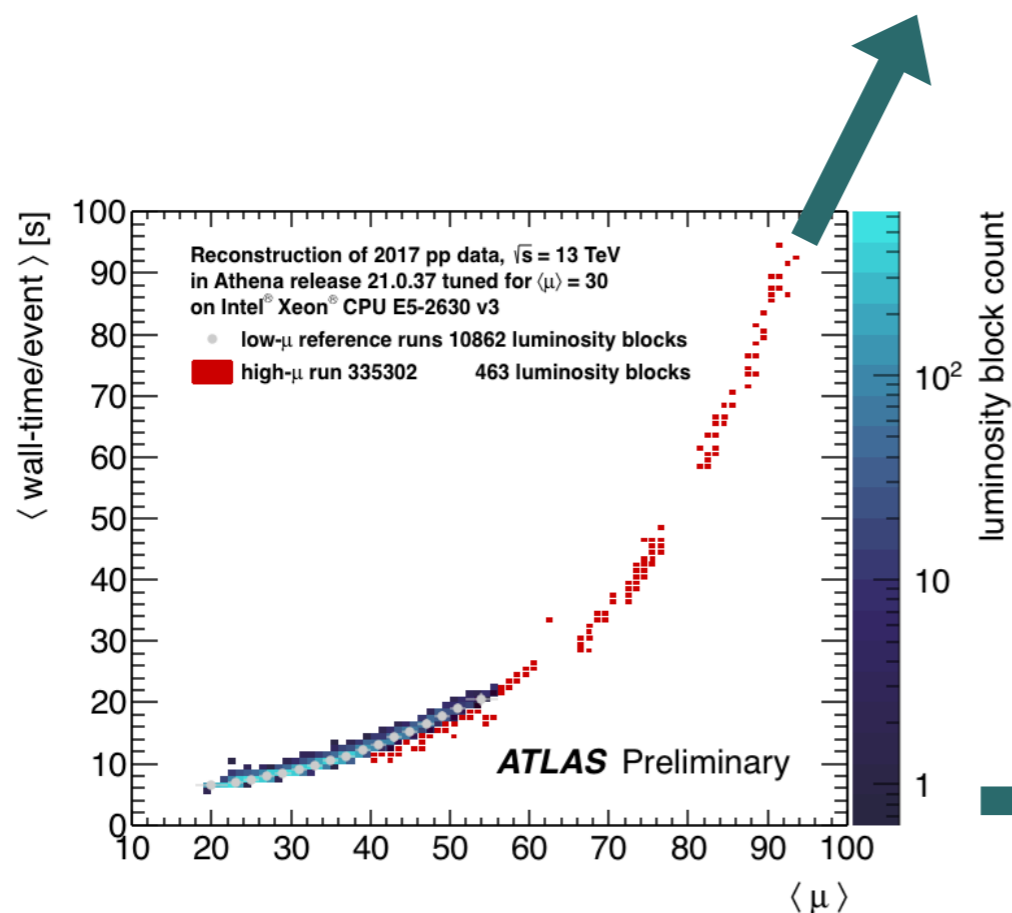
Image Credit

*computing needs =
offline computing
needs

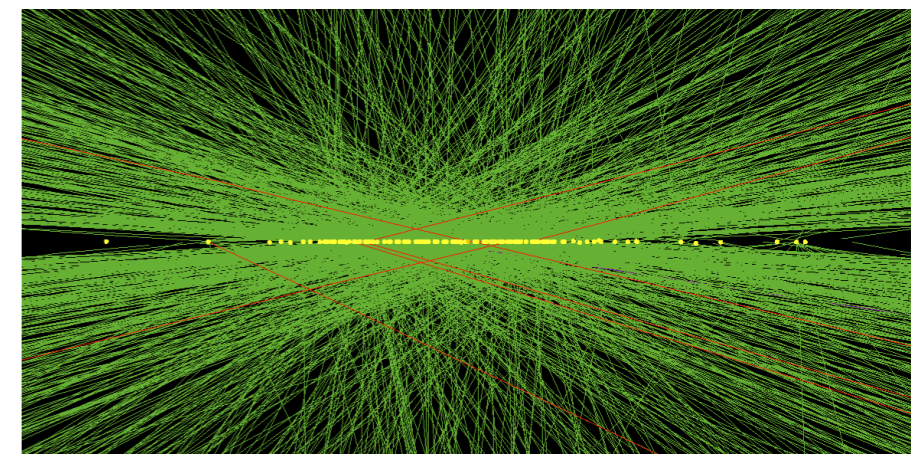
Disk space is also a
challenge, but not
something I'll cover
today

Example: Reconstruction Challenge

- For ATLAS and CMS, the HL-LHC will bring
 - **5-7x** increase in luminosity (LHC accelerator upgrade)
 - **4-5x** increase in event size (new detectors)
 - **10x** increase in event rate (trigger upgrade)
- The problem will be far worse at **future colliders** such as FCC-hh with up to 1000 (!) additional collisions (pile up) per bunch crossing

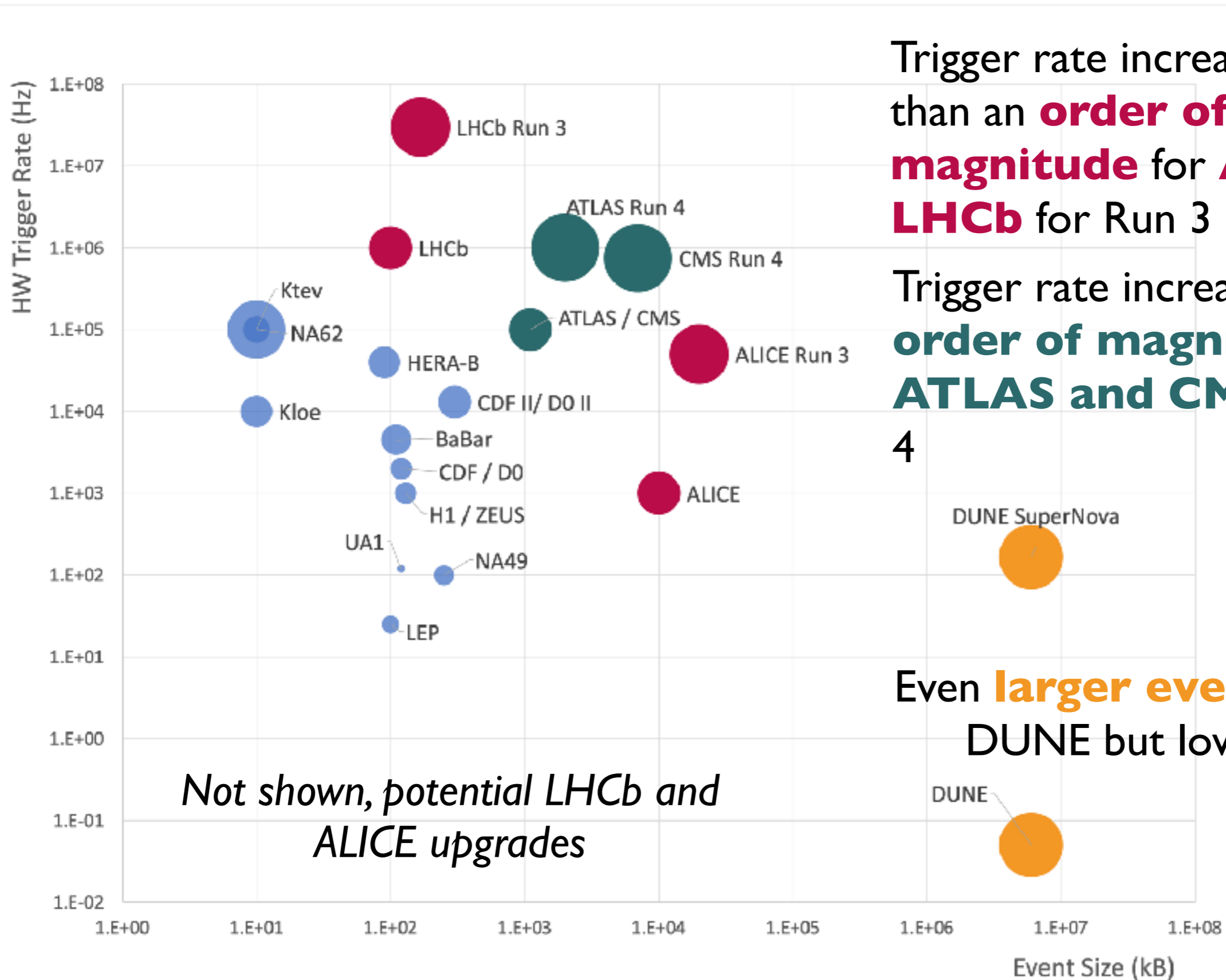


Pile up



● proton-proton collision vertex

Example: Trigger Challenge



Trigger rate increases by more than an **order of magnitude** for **ALICE and LHCb** for Run 3

Trigger rate increase by an **order of magnitude** for **ATLAS and CMS** for Run 4

Even **larger event** sizes for DUNE but lower rate

Not shown, potential LHCb and ALICE upgrades

Other Challenges

Challenges anticipated at each step of the **data processing** and **simulation**

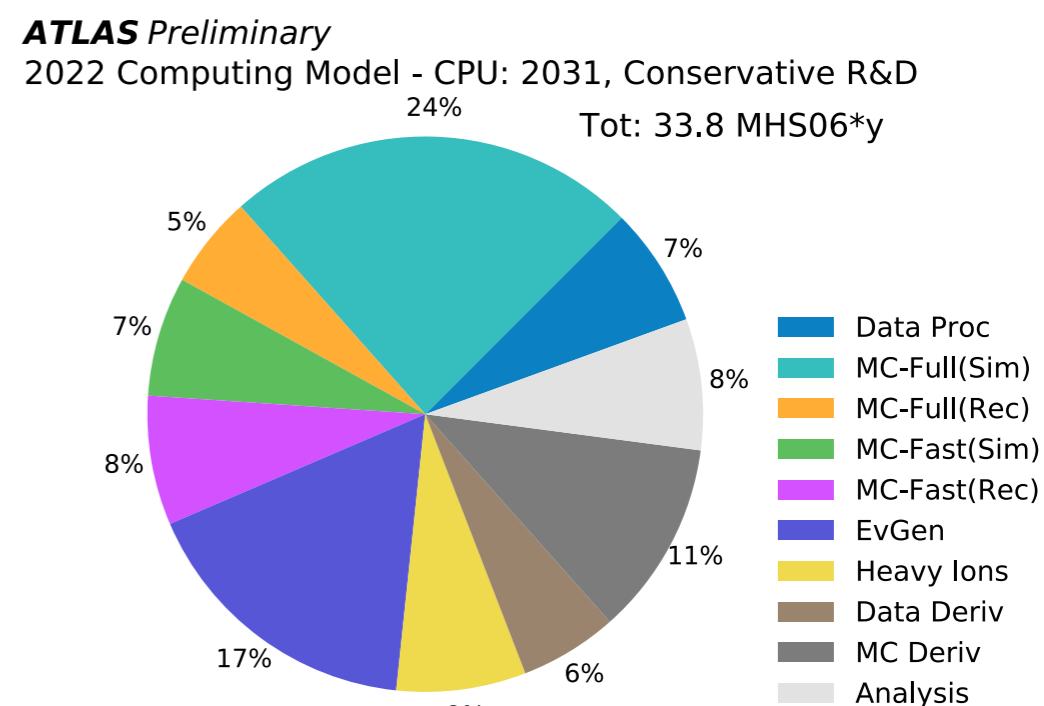
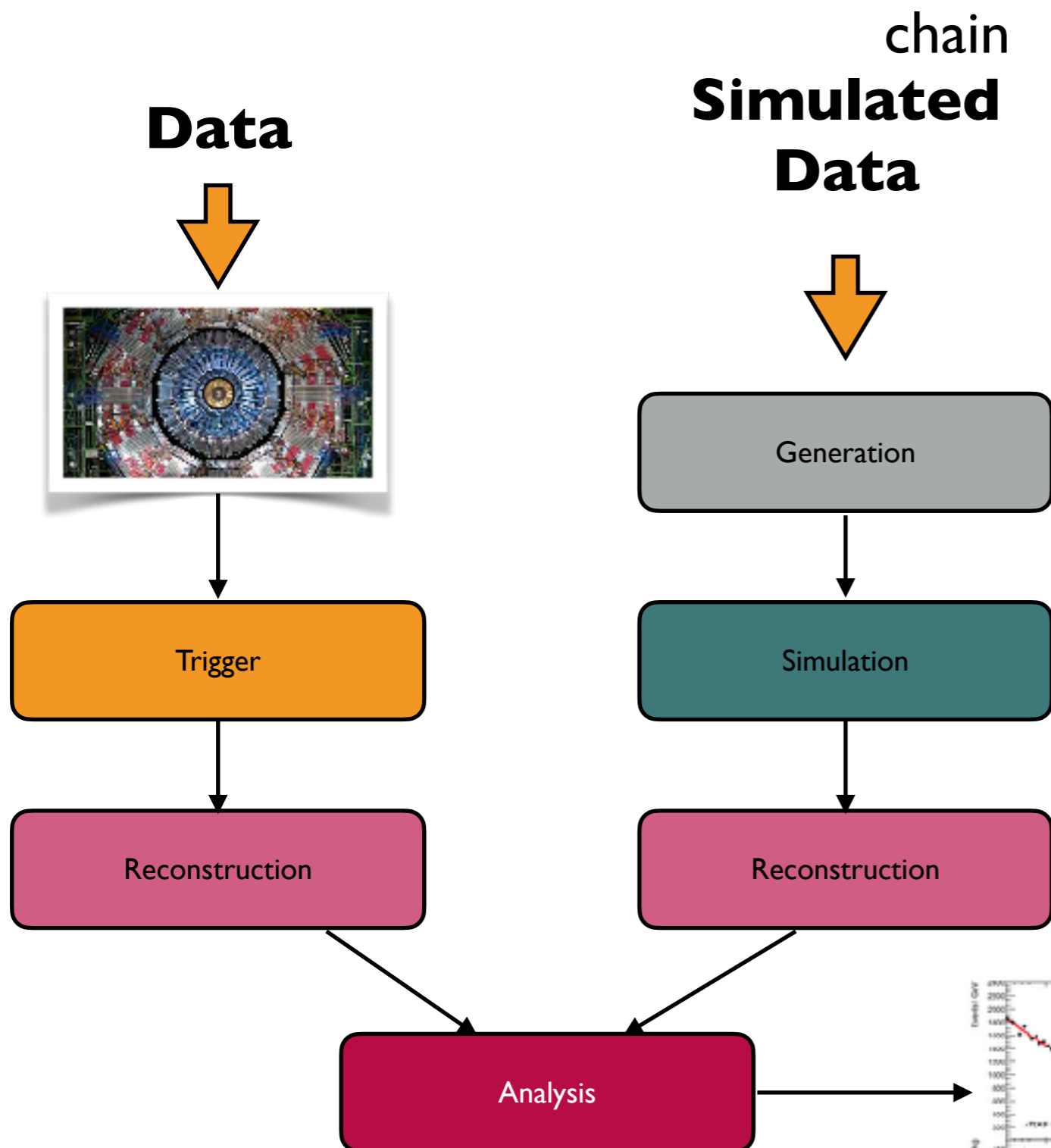
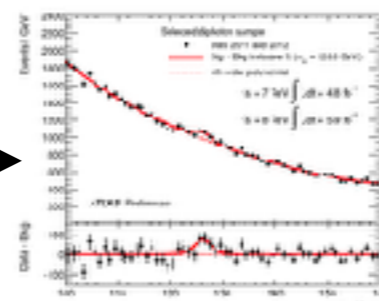


Image Credit



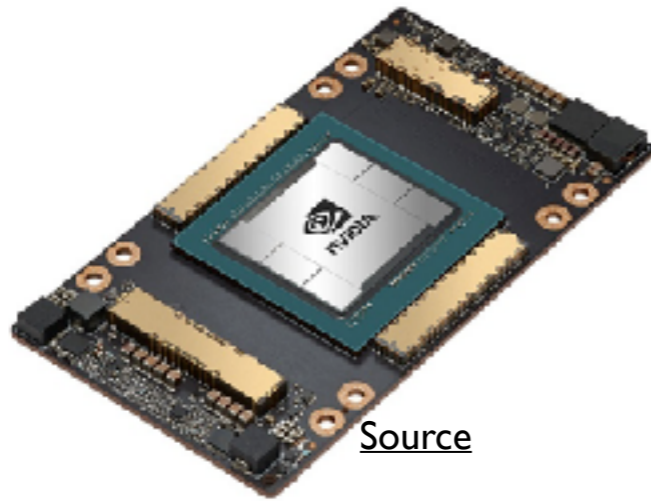
Beyond CPUs

- **Hardware accelerators** are custom-made hardware designed to perform specific functions more efficiently than CPUs
- Wide variety of hardware accelerators depending on the application
 - e.g. **GPU, FPGA, TPU**
- We use hardware accelerators frequently in our daily lives
 - e.g. graphics acceleration, encryption, machine learning, decoding video streams
- As we'll see later, a large fraction of the power in High Performance Centers (HPCs) comes from GPUs
- “New” computing paradigms
 - **Neuromorphic** computing, **quantum** computing....

Common Types of Hardware Accelerators

Graphical Processing Units (GPUs)

e.g. NVidia, AMD, Intel



Source



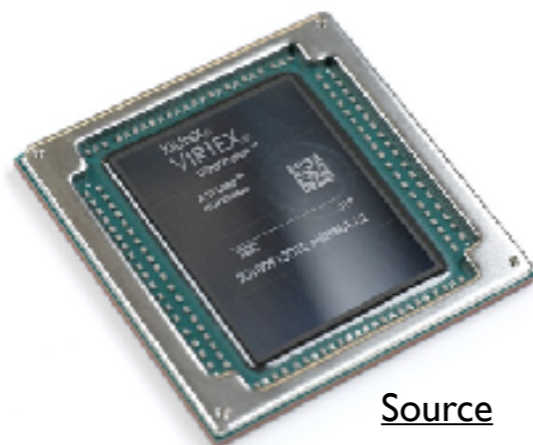
Source



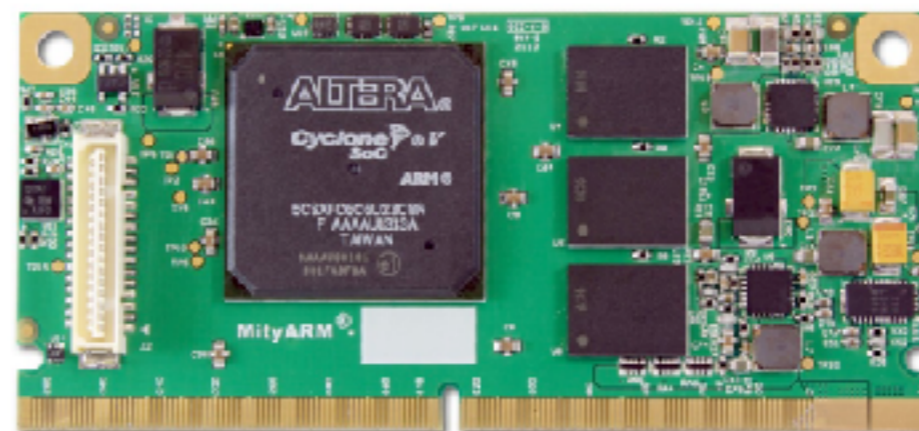
Source

Field Programmable Gate Arrays (FPGAs)

e.g. Xilinx, Altera



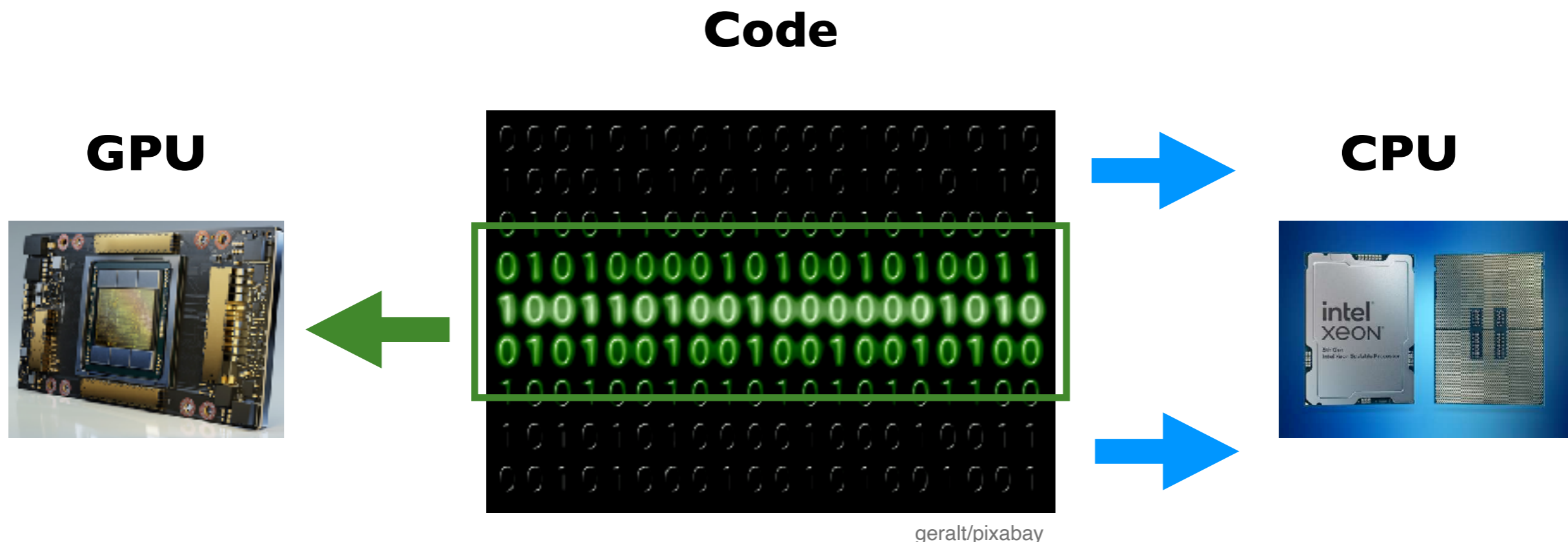
Source



Source

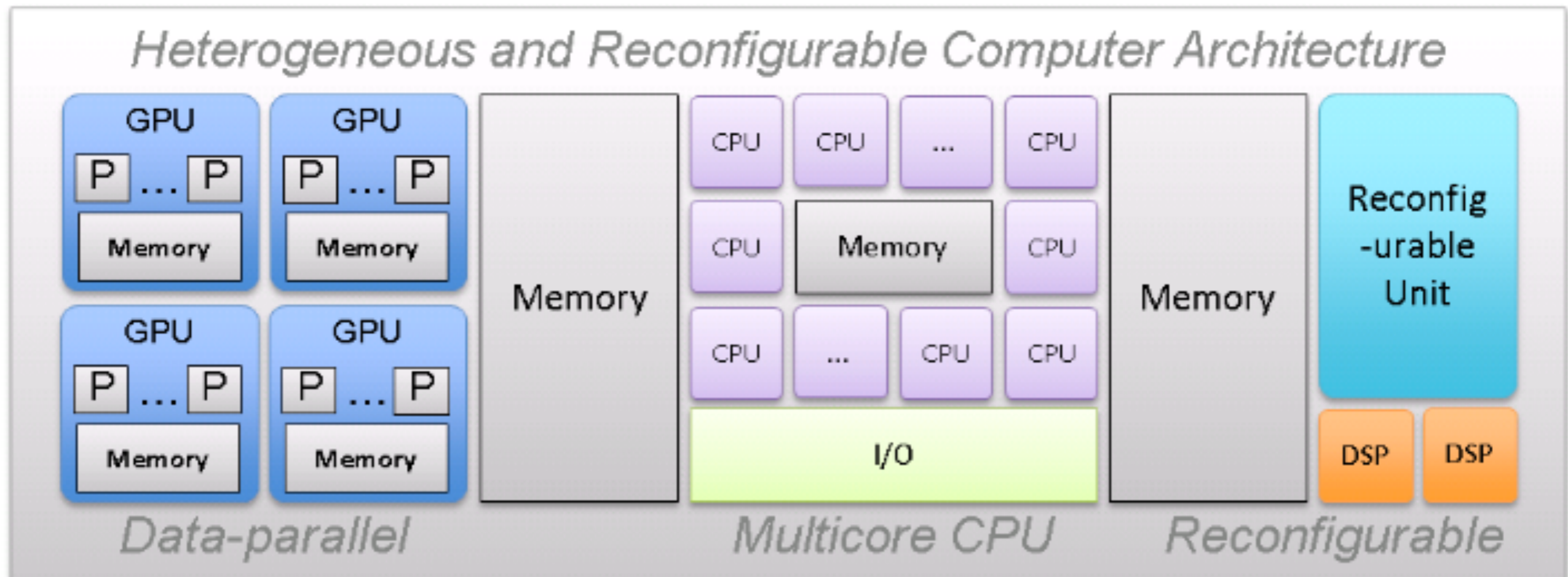
Using Hardware Accelerators

- One approach is to identify the most **computationally intensive** parts of the code and **parallelize** those to be executed on the accelerator
 - The rest of the (ideally **sequential**) code is executed on the **CPU**
- This requires data to be **transferred** to and from the CPU
 - Can be the **bottleneck** in execution



More generally: heterogeneous Computing

- More generally one can **mix and match** different processors, which is termed **heterogeneous computing**
- Different platforms can be used **concurrently**



Heterogeneity in super computers

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	Frontier - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,194.00	1,679.82	22,703
2	Aurora - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	4,742,808	585.34	1,059.33	24,687
3	Eagle - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Microsoft Azure United States	1,123,200	561.20	845.34	
4	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,348	442.01	537.21	29,899
5	LUMI - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107

AMD CPU & GPU

Intel CPU & GPU

Intel CPU & NVidia GPU

ARM

AMD CPU & GPU

Source



Not particularly heterogeneous in terms of operating system (**Linux**) or processor family (**95% Intel**)

Source

Slide Inspiration

Current and Future HPCs

Increasingly complex heterogeneous ecosystem

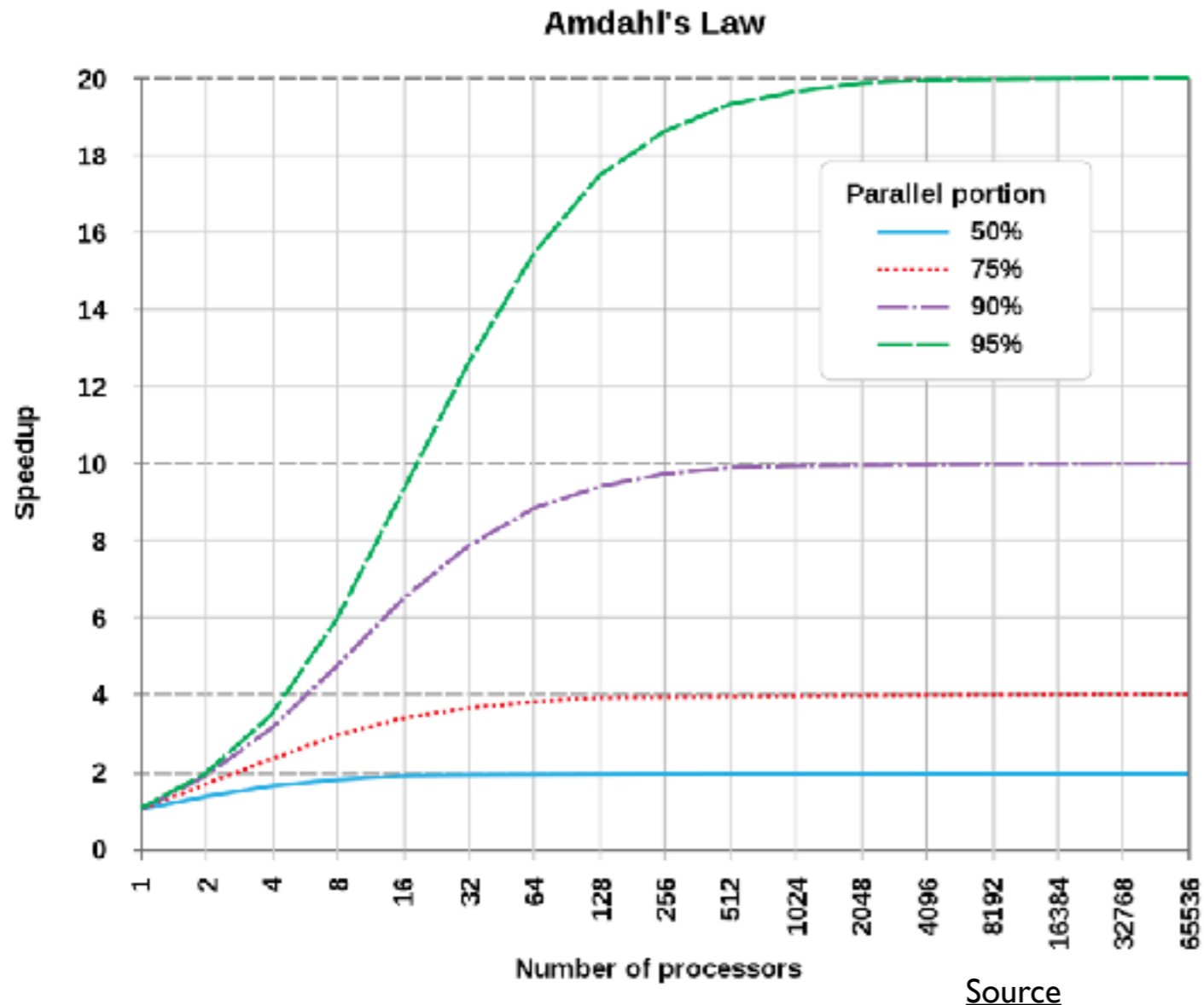
- A new golden age for computing architectures
- or -
a disaster?

		Accelerators				
		Intel	NVidia	AMD	FPGA	Other
CPU	Intel	Aurora	Cori Leonardo Piz Daint Tsukuba MareNostrum			
	AMD		Perlmutter	Frontier El Capitan		
	IBM		Summit Sierra MareNostrum			
	Arm		Wombat			Astra*
	Fujitsu					Fugaku

- Amazon EC2 P3
- Amazon Graviton2
- Google Cloud TPU
- Microsoft Azure
- Intel DevCloud

HPC = High Performance Computing

Parallelization: Amdahl's Law



Speed up in latency
 $1/(S + P/N)$

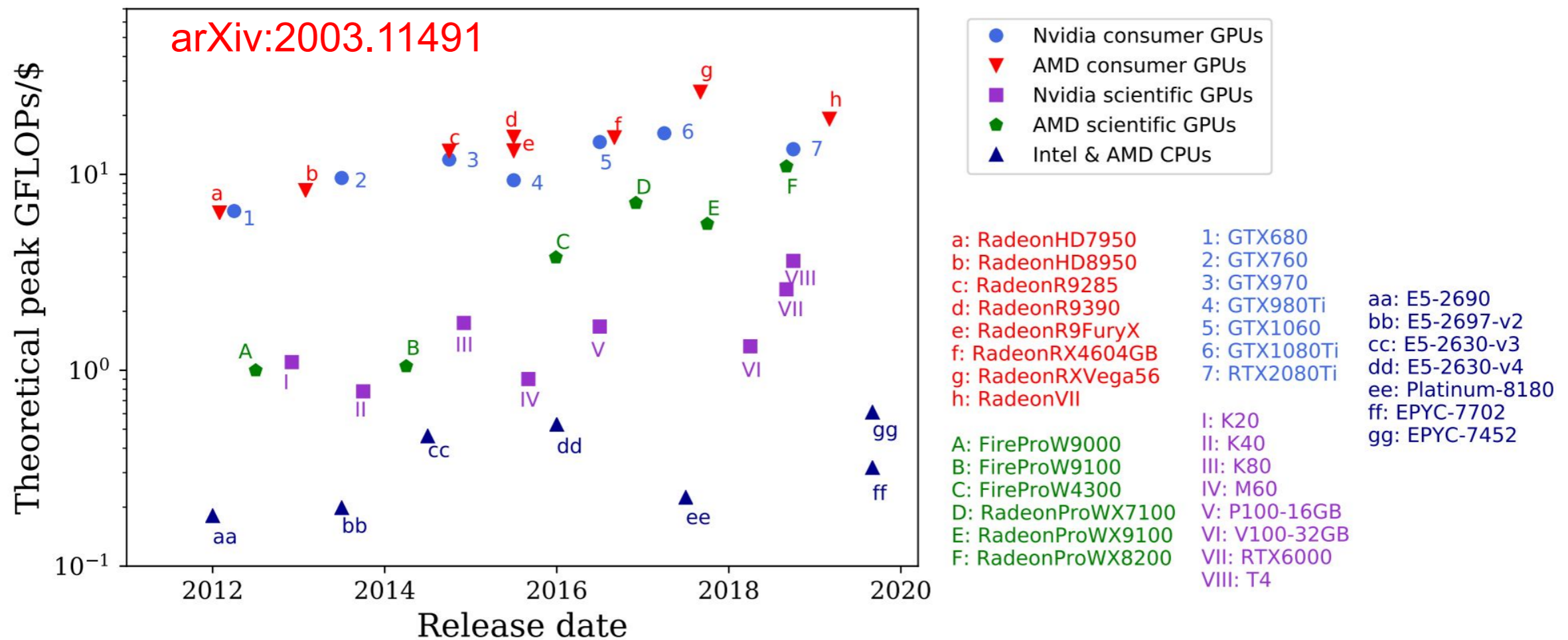
S: sequential part of program

P: parallel part of program

N: number of processors

Import: Speedup depends on how much of the problem can be parallelized

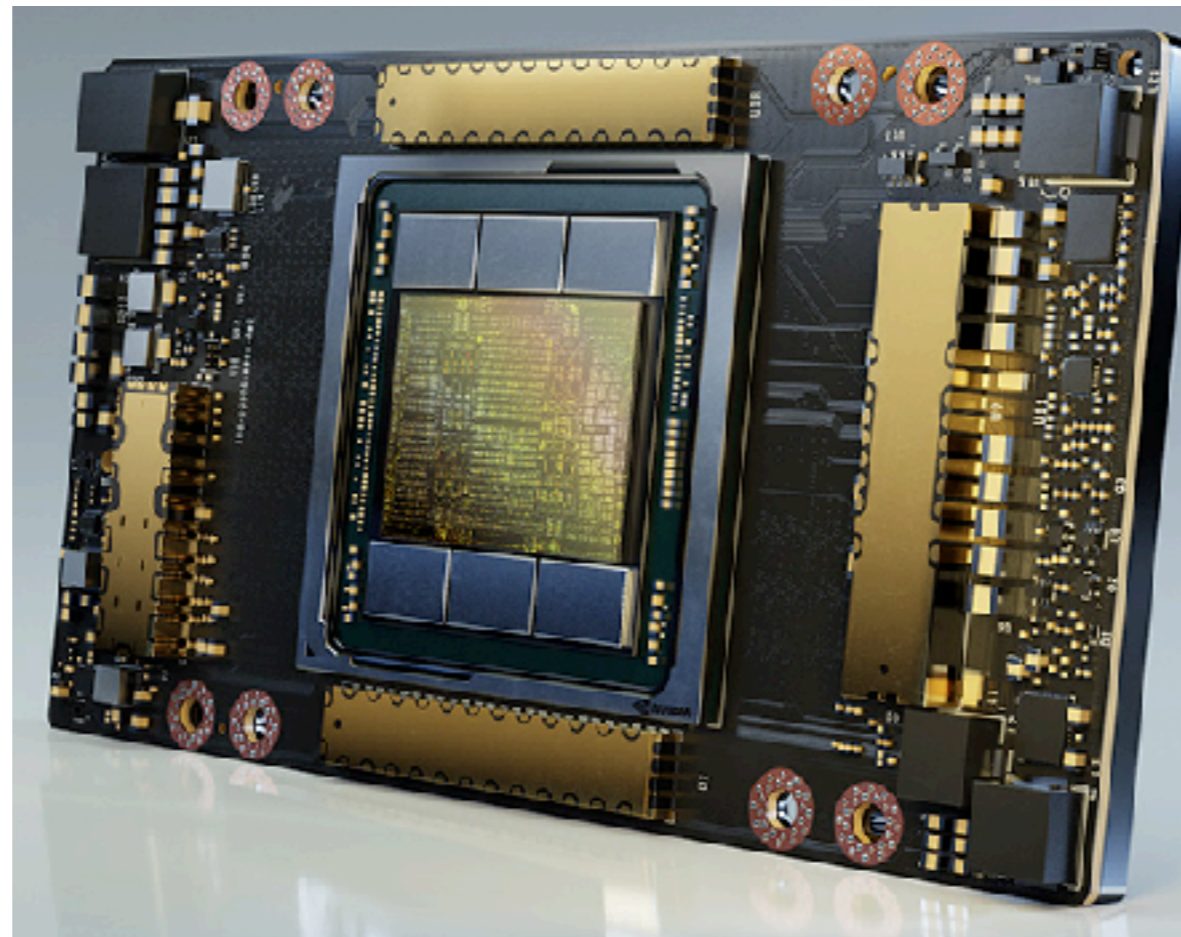
Why GPUs?



Main reason: Higher computing speed

Introduction to GPUs

- GPUs are silicon microprocessors containing cores, register, memory, etc
- **Many-core processors**
- Follow the single instruction, **multiple threads (SIMT)** execution model
 - Asynchronous programming model, i.e. threads are not executed in lockstep



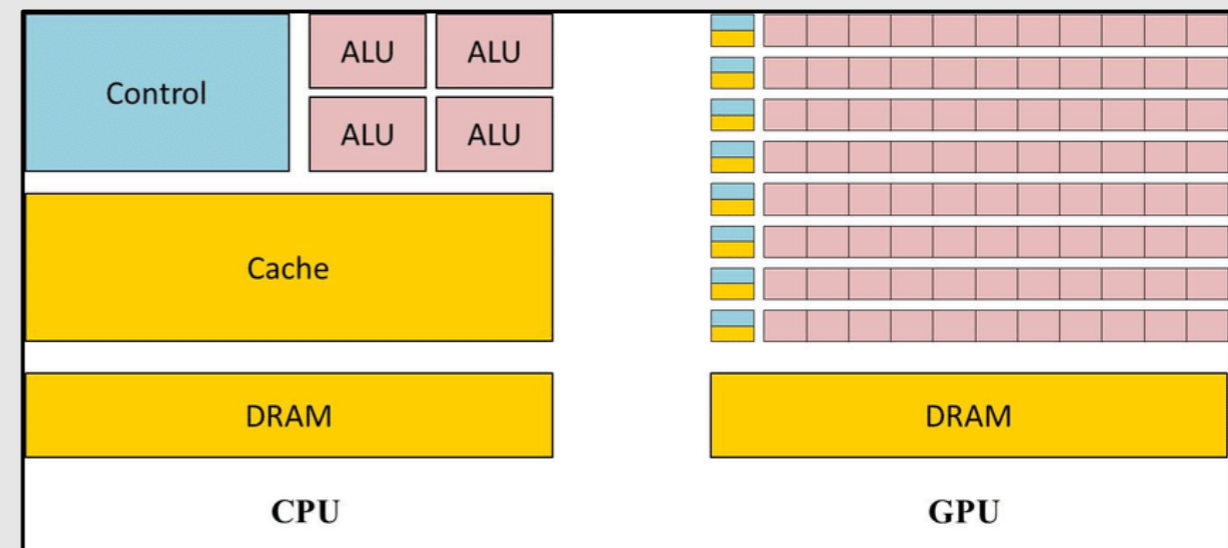
GPU vs CPU

▶ CPU:

- small number of complicated cores
 - branch prediction
 - prefetching
- multiple levels of large caches
- low latency

▶ GPU:

- very many (100k+) simple cores
 - much more hardware for low precision ops than dp
- cores in a block operate in lockstep
 - branch mis-prediction causes stalls for many cores
- small cache
- vectorized memory ops
- high throughput, high latency
- low power (per FLOP)

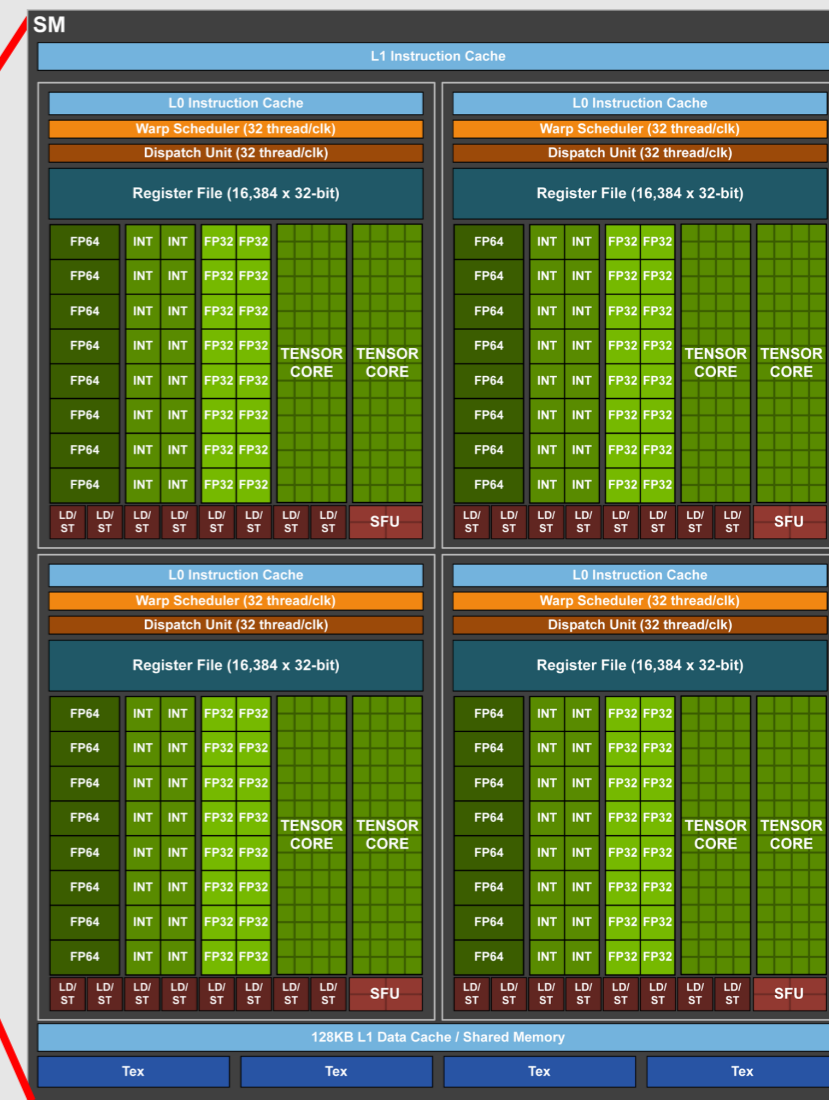
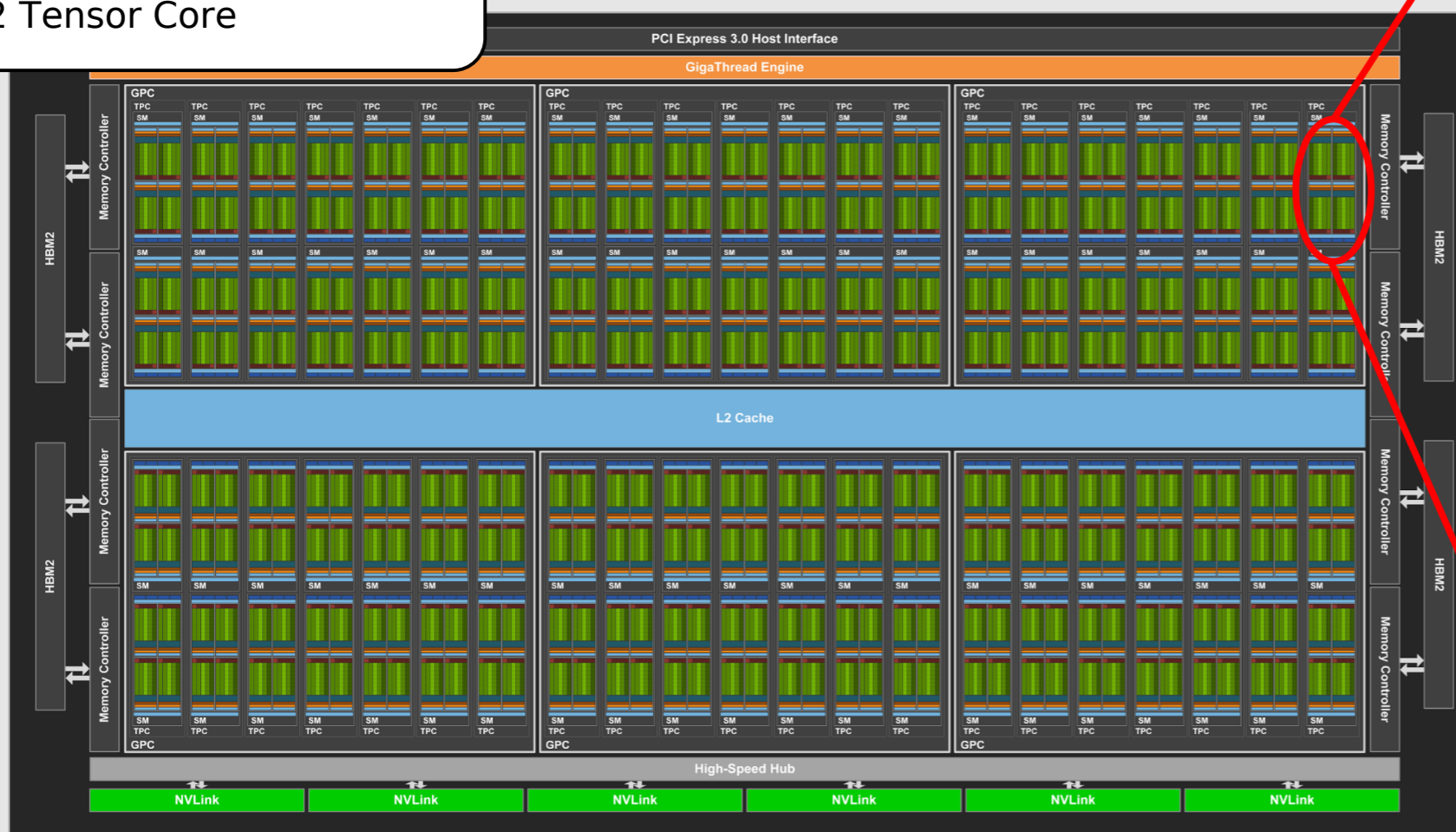


Modern GPU

NVidia V100

- 6 Graphics Processor Cluster
- 42 Texture Processor Cluster
- 84 Streaming Multiprocessor
 - 4x 8 FP64
 - 4x 16 FP32
 - 4x 16 INT32
 - 2 Tensor Core

- 7.8 TFLOP FP64
- 15.7 TFLOP FP32
- 125 TFLOP Tensor matrix mult
- 300 W



Types of GPU

	Scientific GPUs	Gaming GPUs
Precision	~3 times more single precision TFLOPS than double precision → suited for double precision	~40 times more single precision TFLOPS than double precision → not well suited for double precision
Error correction	Available	Not available
Connection	NVLink & PCIe	Only PCIe
Price	~5-6 x the price of gaming GPUs	Several hundred dollars Depending on model (and year)

Specification

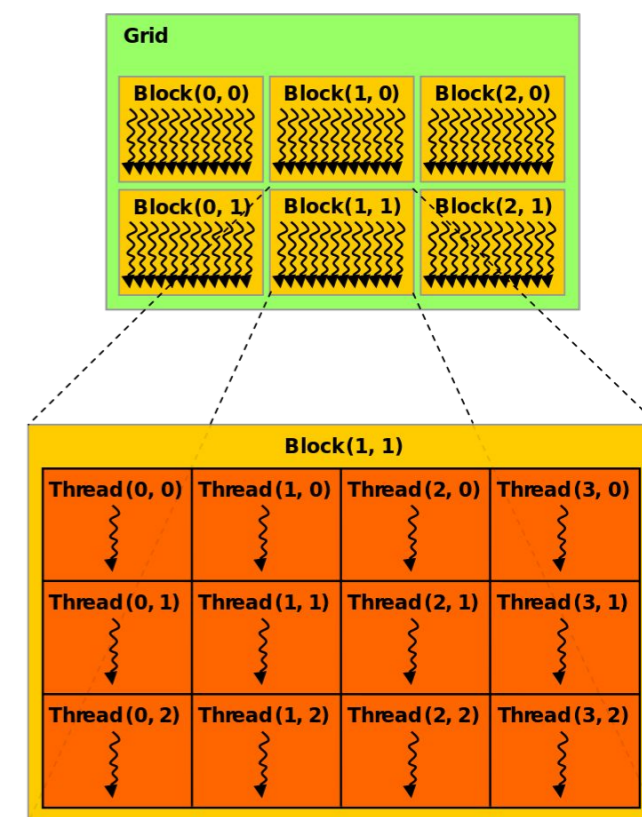
GPU vs. CPU: Specifications

	AMD Ryzen Threadripper 3990X	Nvidia A100
Core count	64 cores / 128 threads	6912 cores
Frequency	2.9 GHz	1.41 GHz
Peak Compute Performance	3.7 TFLOPs	19.5 TFLOPs (single precision)
Memory bandwidth	Max. 95 GB/s	1.6 TB/s
Memory capacity	Max O(1) TB	40/80 GB
Technology	7 nm	7 nm
Die size	717 mm ²	826 mm ²
Transistor count	3.8 billion	54.2 billion
Model	Minimize latency	Hide latency through parallelism



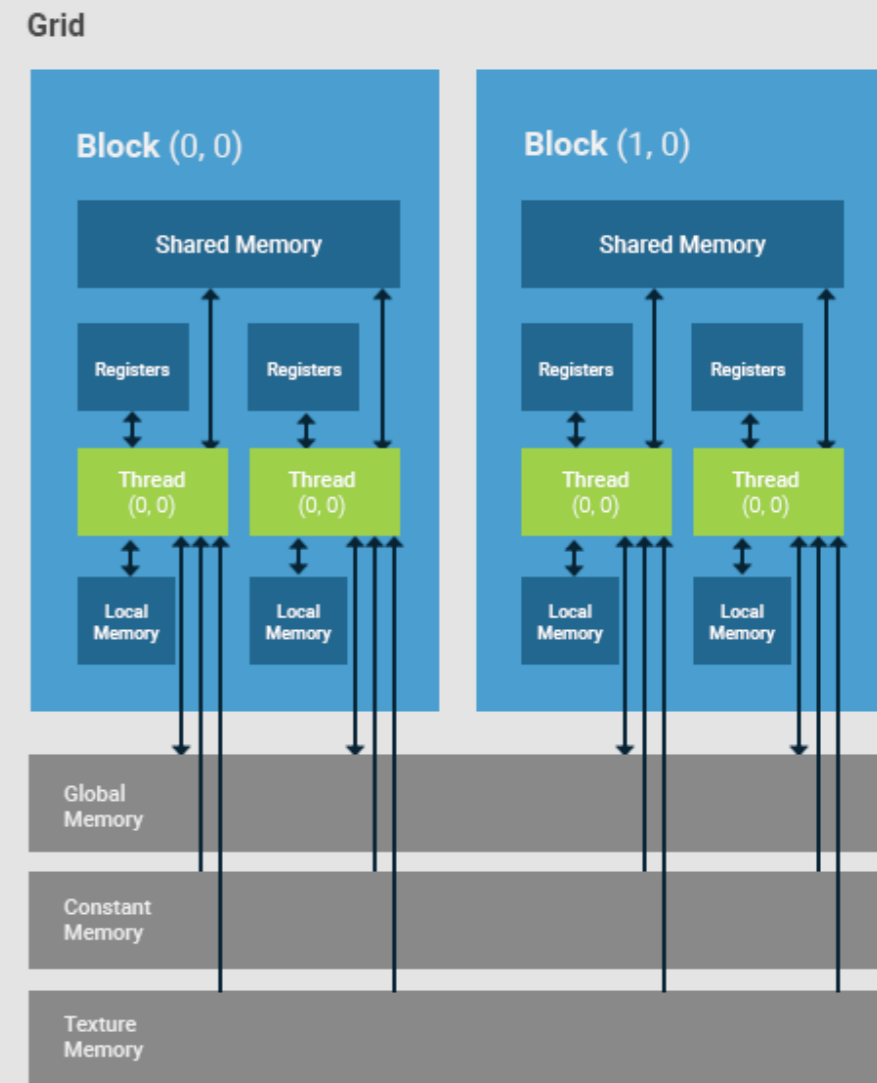
Threads, warps and blocks

- On a modern GPU, computations are executed by threads
- Threads are grouped into a warp
 - 32 threads in a warp
 - all threads in a warp execute the same instruction
 - in-warp branches or divergent memory access will cause stalls and poor performance
 - Threads are executed w/ in-order processing on a “core” or ALU
- Warps are grouped into a block, launched by warp scheduler
 - must execute on same Streaming Multiprocessor (SM)
 - (max 1024 threads/block)
 - share resources of a SM (registers, cache lines, shared mem)
 - only threads in a block can synchronize execution w/ barriers
- Blocks are grouped into grids
 - logically organized as 1D, 2D or 3D groups of blocks in a grid
- Modern GPUs need many threads for full occupancy: hide instruction latency w/ oversubscription
 - NVidia A100: $64 \text{ [warp schedulers/SM]} * 32 \text{ [threads/warp]} * 108 \text{ [SMs]} = 221184$



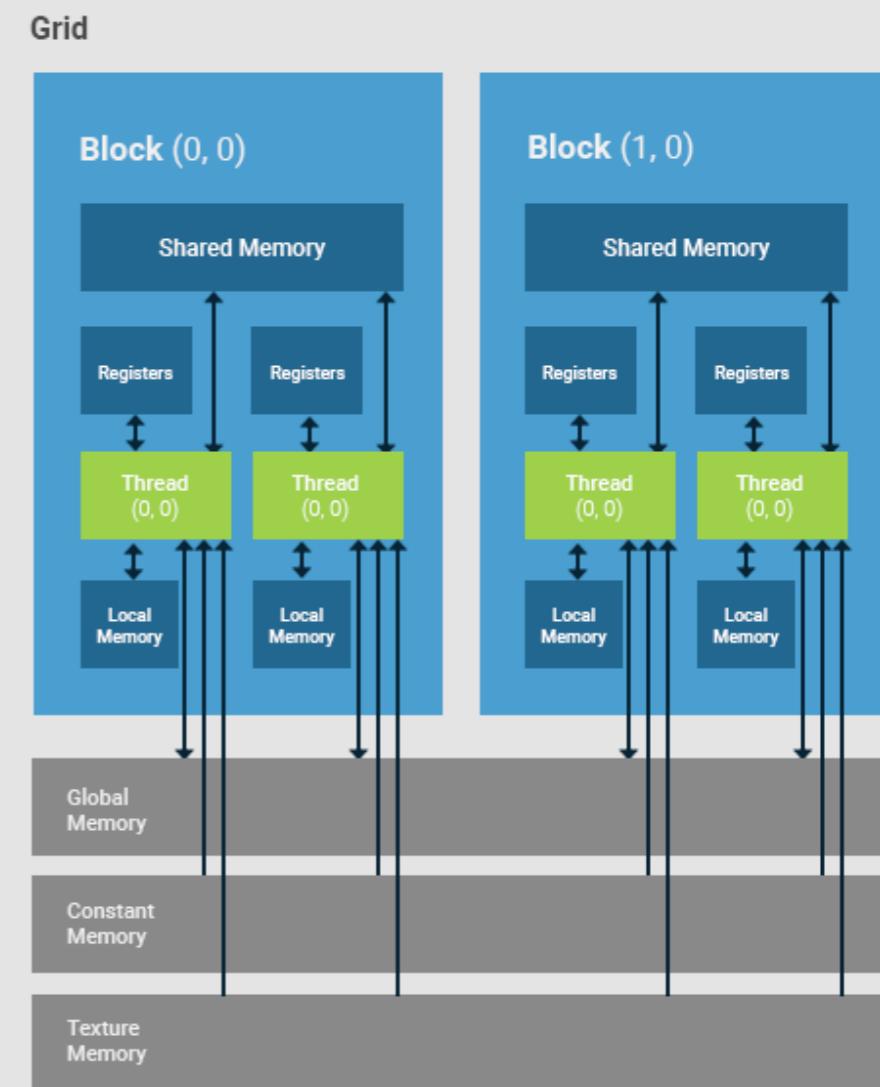
GPU Memory Hierarchy (CUDA)

- ▶ Several different levels of memory explicitly addressable, with different levels of access speed and bandwidth
- ▶ Register
 - fastest
 - used for allocating variables private to each thread
 - only small number available per SM. using too many will reduce the number of concurrent thread blocks on a SM
- ▶ Shared
 - on chip - lower latency and higher bandwidth than global
 - shared among threads in a block. using too much will reduce number of active warps
- ▶ Global
 - visible by all threads and SMs on device
 - allocated and freed by host
 - large: up to 32GB on some devices
 - slow: can take several hundred cycles to access



GPU Memory Hierarchy (CUDA)

- ▶ **Constant**
 - fast, read only device memory
 - allocated by host, visible to all kernels and SMs
 - allocated on a separate cache in each SM
 - for best performance, all threads in a warp should access same memory location
- ▶ **Texture**
 - fast, read only device memory allocated by host
 - separate cache on each SM
 - optimized for 2D memory access
- ▶ **Local**
 - virtual concept, actually located in global memory
 - used for register spills, variables that can't fit into registers, arrays whose indices can't be deduced at compile time
- ▶ **Memory transfers between device and host have large latencies (many hundreds of cycles), throughput limited by bus (PCIe 3: 32 GB/s; NVLink = 300GB/s bidirectional)**
 - memory transfers can introduce synchronization points
- ▶ **Improper memory usage by kernels can result in extremely poor performance**



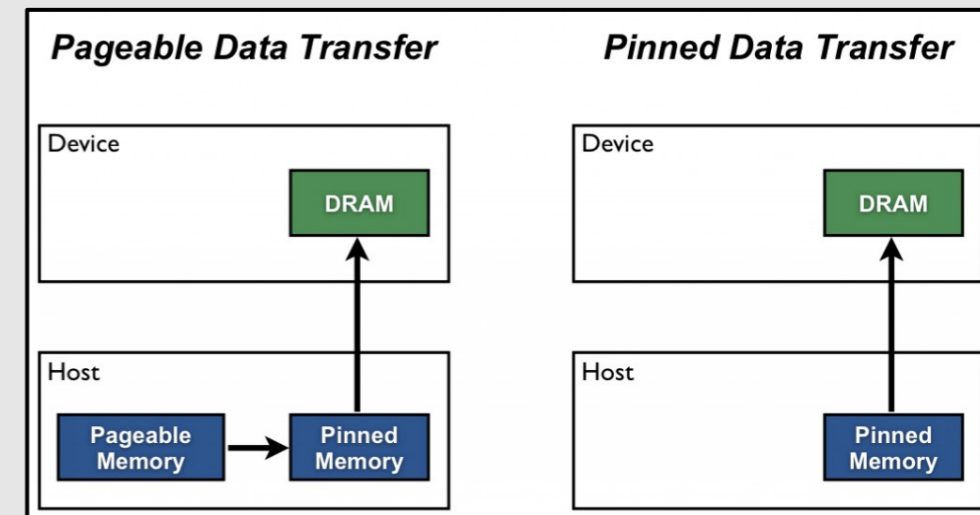
Memory Access Pattern

- ▶ Allocate global memory on device (from host)
- ▶ Copy memory from host to global device memory
- ▶ Load data from device memory to shared memory
 - Synchronize with all the other threads of the block so that each thread can safely read shared memory locations that were populated by different threads
- ▶ Process the data in shared memory
- ▶ Synchronize again if necessary to make sure that shared memory has been updated with the results
- ▶ Write the results back to device memory
- ▶ Transfer device memory back to host

- ▶ Memory structures and layouts have changed significantly between different generations and architectures of NVidia devices
 - for best performance, tuning to specific card / architecture is necessary
- ▶ Intel and AMD devices have the same multi-level hierarchy, but with their own specific peculiarities

Pinned Memory

- ▶ Memory allocated on the host RAM can be swapped out
 - they are "pageable"
- ▶ GPU cannot directly access pageable host memory
 - must first be copied to page-locked or "pinned" array
 - then transferred to device
 - can cause a significant overhead
- ▶ Can explicitly allocate pinned host memory to minimize copying
 - pinned memory is lost to the system until freed



GPU Memory Hierarchy (SyCL)

- ▶ SyCL manages memory in a more abstract manner than CUDA
 - separates the concepts of *storage* from *access*
- ▶ Memory is created/allocated in `sycl::buffers<TYPE>(SIZE)`
- ▶ Memory is accessed via `sycl::accessor<access_mode, access_target>`
 - `access_mode` can be read, write, read_write, atomic, etc
 - `access_target` can be host, global, constant, image, etc

- ▶ No explicit data movement in SyCL. Data is automatically moved between host and device as needed by the kernels
- ▶ Depending on the access type, and access target, SyCL will try to put the memory in the most optimal location

CUDA name	SyCL name
Register memory	Private memory
Shared memory	Local memory
Global memory	Global memory
Constant memory	Constant memory
Texture memory	Image memory
Local memory	N/A

Unified Memory

- ▶ Recently, concept of "unified memory address space" has been introduced
 - CUDA, hip, dpcpp
- ▶ Creates a pool of managed memory that is shared between the CPU and GPU
- ▶ Accessible to both the CPU and GPU using a single pointer
- ▶ Automatically migrates data allocated in Unified Memory between host and device
 - transferred on demand
 - looks like CPU memory to code running on the CPU, and like GPU memory to code running on the GPU
- ▶ Can override **new** allocator, so C++ objects created in unified managed memory
 - enables deep copies of complex objects
- ▶ Makes device programming much simpler
- ▶ Not as performant as explicit device memory management
 - will not overlap kernel execution in streams with asynchronous memory transfers

Performance Comparisons

- Compare performance using Floating-Point Operations per Second (FLOPS)
- GPUs deliver almost an order of magnitude for FLOPS/sec

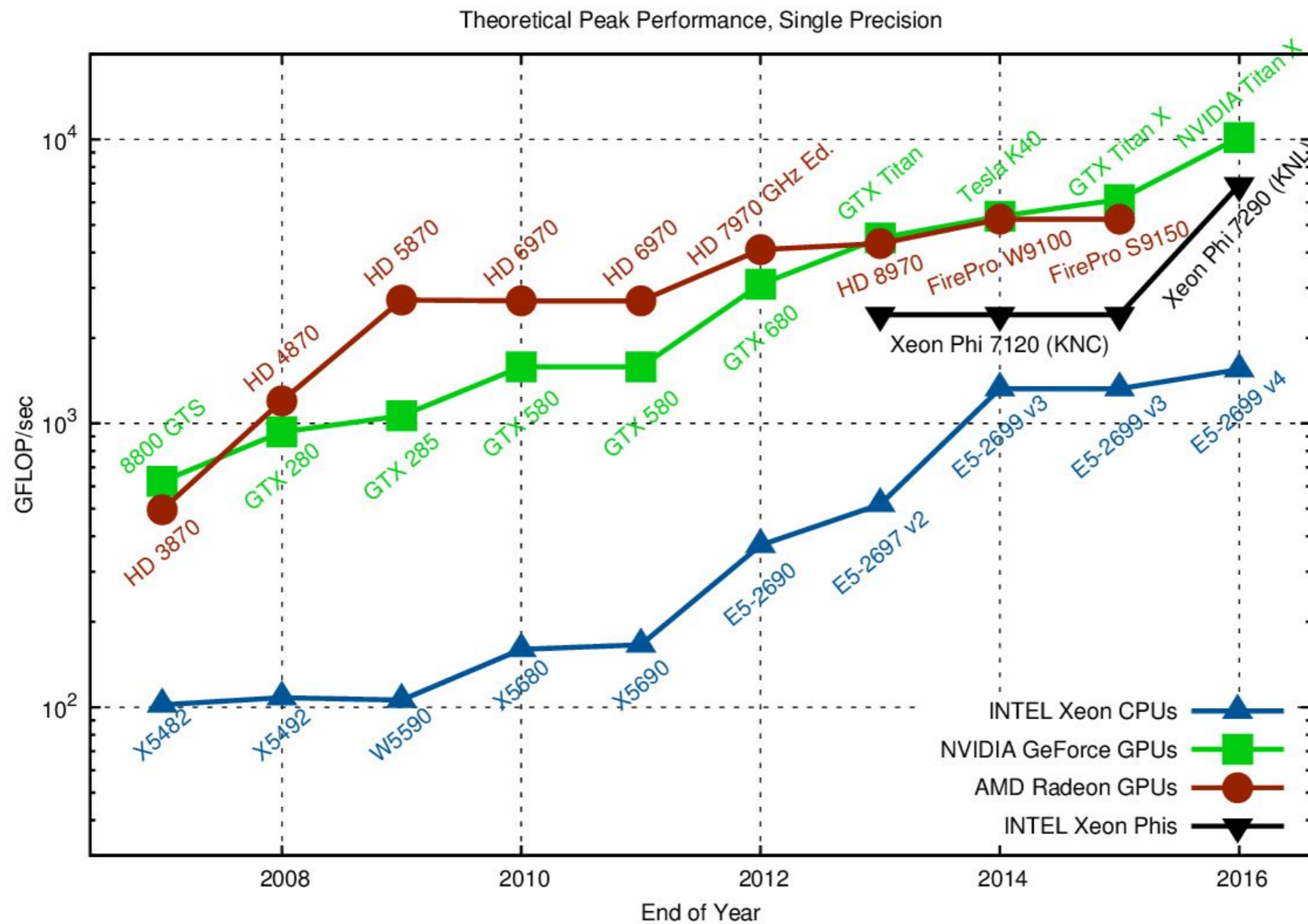
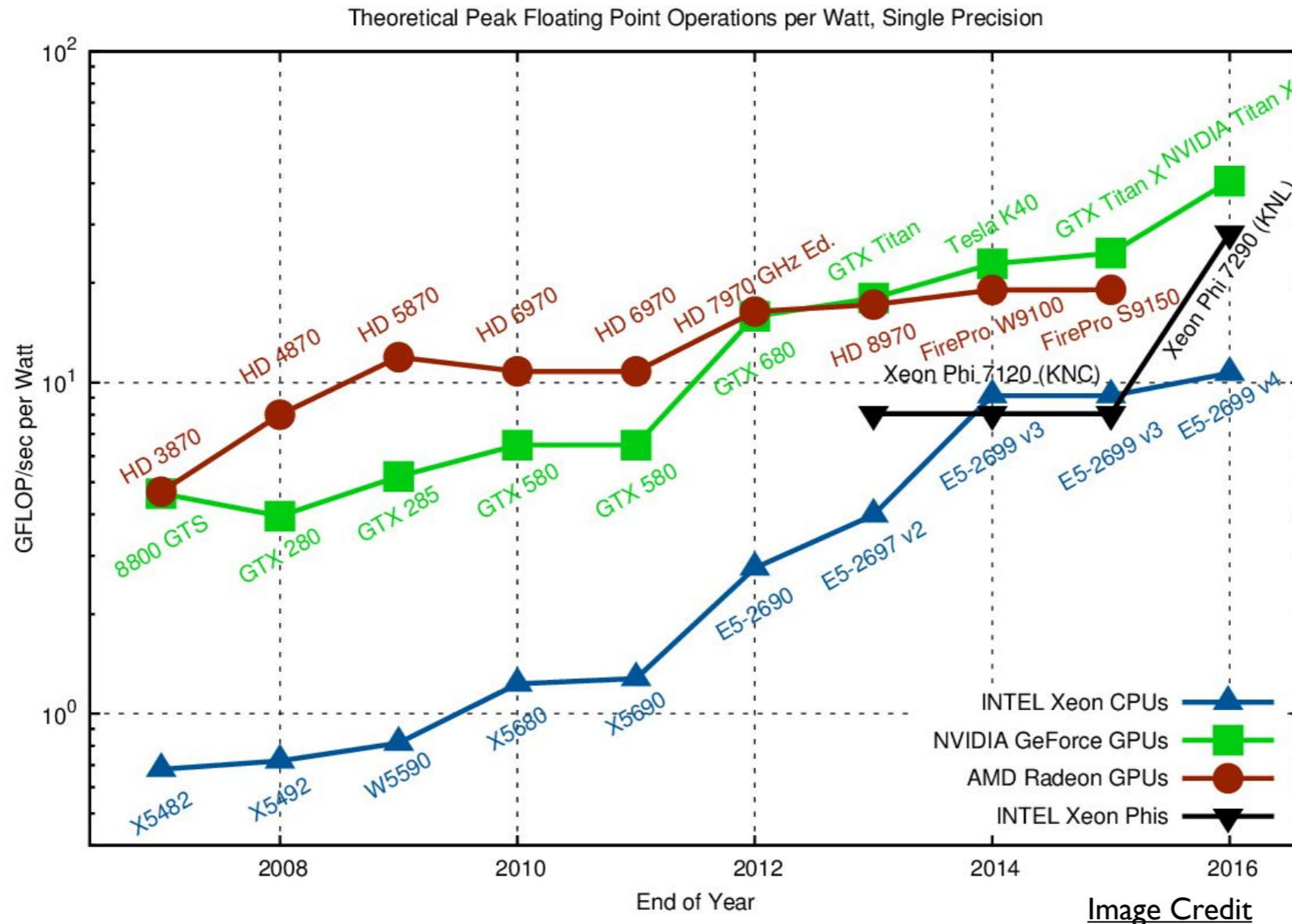


Image Credit

GPU Power Consumption

- Power consumption is often the limiting factor in hardware manufacturing/use
 - Performance constrained by the amount of power drawn and heat dissipated
- Study power consumption using FLOPs per Watt of energy consumed



FPGAs

- **Field Programmable Gate Arrays (FPGAs)** are **integrated circuits** that are available off the shelf
- High **throughput**
 - Good for workloads with many branch mispredictions and cache faults
- Low **latency**
 - $O(\mu s)$
 - constant and predictable
- More **flexible** than custom-built hardware
- Commercial market for FPGAs has been around since the 1980s



Field-programmable gate array. (2024, January 17). In Wikipedia. https://en.wikipedia.org/wiki/Field-programmable_gate_array

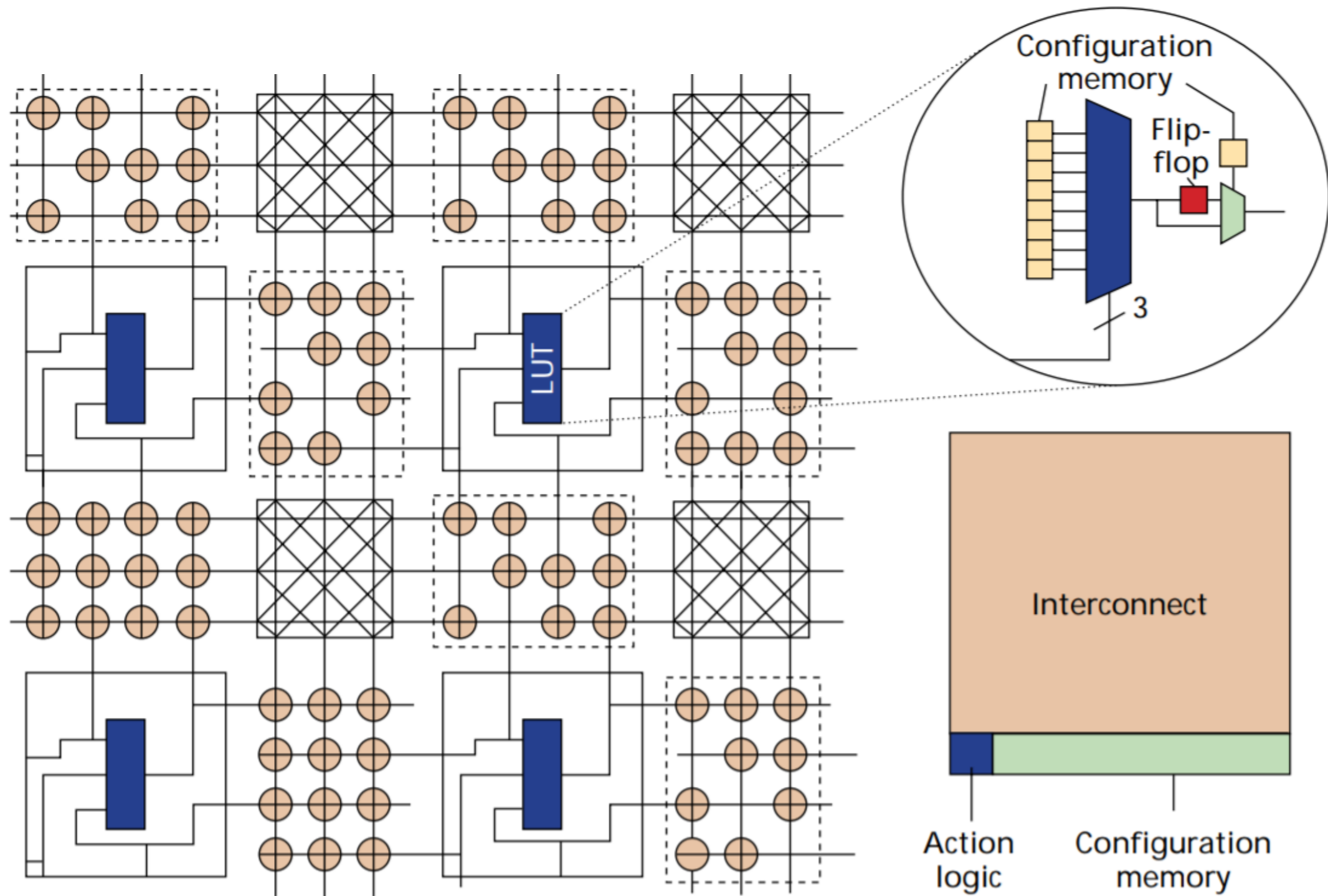


By © Raimond Spekking / CC BY-SA 4.0 (via Wikimedia Commons), CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=81288554>

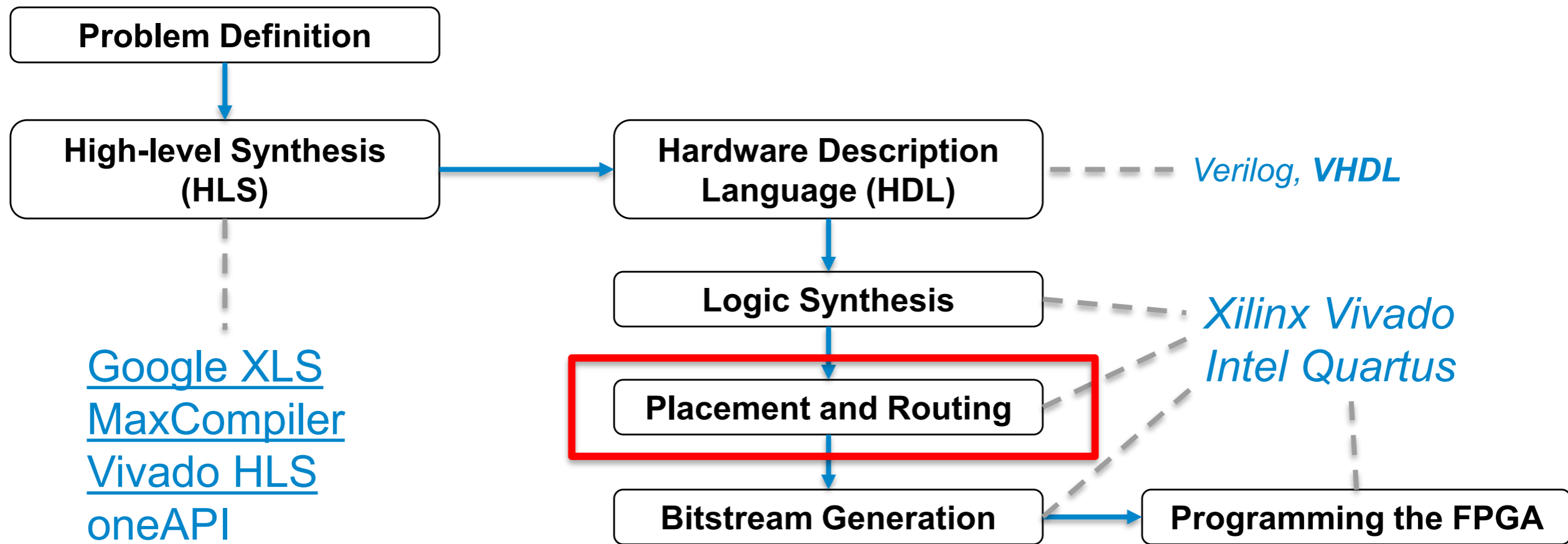
FPGAs

- FPGAs consist of thousands of **logic blocks** plus **I/O blocks**
 - Connected via **programmable interconnect**
- Program FPGAs by **configuring a circuit**
 - Hardware implementation of an algorithm
- FPGAs are very good at **integer computations**
- Do not require a computer to run because they have their own I/O
- Traditionally programmed using **hardware description languages**, e.g. Verilog, VHDL
 - Required long development times
- **High-level languages (HSL)** have become available more recently

FPGA Architecture



Modern FPGA Code Design Flow



Source

GPU vs FPGA



GPUs

- Higher latency
- Connection via PCIe (or NVLink)
- Bandwidth limited by PCIe
- Very good floating point operation performance
- Lower engineering cost
- Backward / forward compatibility



FPGAs

- Low & deterministic latency
- Connectivity to any data source
- High bandwidth
- Intermediate floating point performance
- High engineering cost
- Not so easy backward compatibility

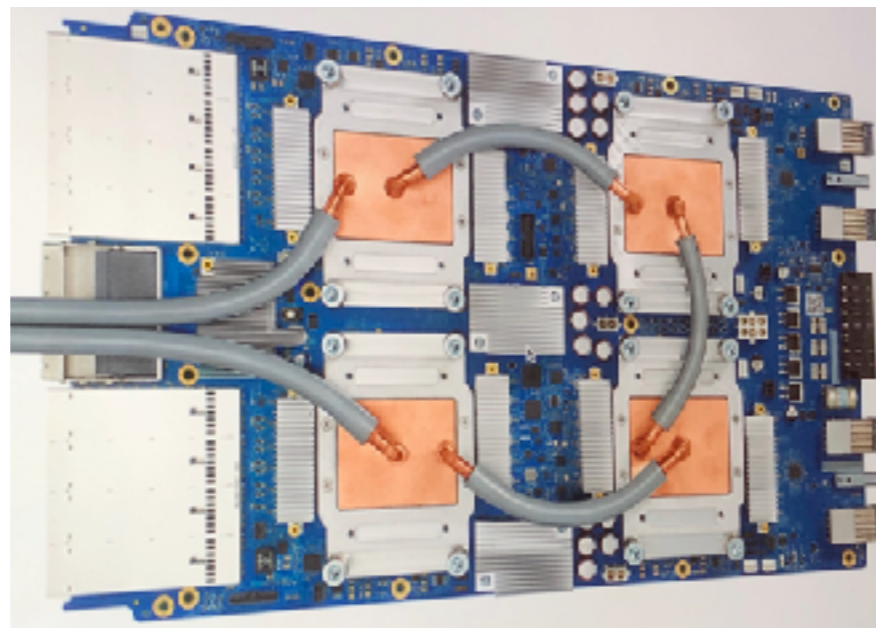


CPU - GPU - FPGA

	CPU	GPU	FPGA
Latency	$O(10) \mu\text{s}$	$O(100) \mu\text{s}$	Deterministic, $O(100) \text{ns}$
I/O with processor	Ethernet, USB, PCIe	PCIe, Nvlink	Connectivity to any data source via printed circuit board (PCB)
Engineering cost	Low entry level (programmable with c++, python, etc.)	Low entry level (programmable with CUDA, OpenCL, etc.)	Some high-level syntax available, traditionally VHDL, Verilog (specialized engineer)
Single precision floating point performance	$O(10)$ TFLOPs	$O(10)$ TFLOPs	Optimized for fixed point performance
Serial / parallel	Optimized for serial performance, increasingly using vector processing	Optimized for parallel performance	Optimized for parallel performance
Memory	$O(100)$ GB RAM	$O(10)$ GB	$O(10)$ MB (on the FPGA itself, not the PCB)
Backward compatibility	Compatible, except for vector instruction sets	Compatible, except for specific features only available on modern GPUs	Not easily backward compatible

For ML: TPUs

- Tensor Processing Units (TPUs) are a type of ASIC being developed by google specifically targeting machine learning applications
- Designed for a high volume of low precision computation (matrix multiplication)
- Available in google cloud
- Could be used for training/inference in HEP
 - Well suited to convolutional neural networks
- Current version is the Edge TPU



Source

Other examples include Intelligence Processing Units (IPUs) designed for irregular and sparse data access

Workload by Accelerator Type

GPUs:

- Relaxed latency requirements
- High FLOPs need
- I/O via PCIe no bottleneck
- Highly parallelizable problem
- Fits within GPU memory



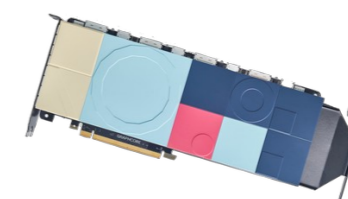
FPGAs:

- Strict latency requirements
- High I/O needs
- Highly parallelizable problem
- Fits within FPGA resources (logic elements and memory blocks)



TPUs / IPUs etc.:

- Machine learning training or inference
- TPUs: Use as a service in the cloud
- IPUs: MIMD compatible problem
- Fit within memory



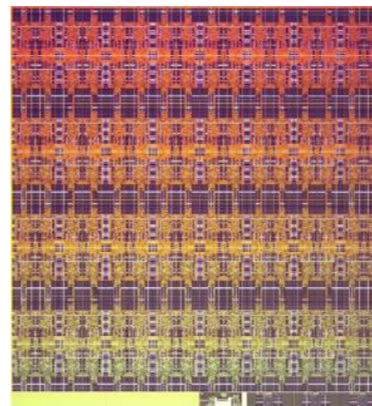
A Comment on Machine Learning

- Machine learning continues to be used ever more extensively in HEP (see the lectures from D. Rousseau)
- There is interplay between the use of ML and novel hardware architectures because such hardware can be used to very efficiently train ML algorithms
 - It can also be used for inference, but inference has less significant computational demands
- Thus in many case, the drive to use increasingly complex ML algorithms leads to the greater adoption of novel hardware
- AND, if there is a drive to use such hardware it leads naturally to the adoption of additional ML algorithms

New Computing Paradigms

Neuromorphic Computing

- **Neural networks**, a mainstay of current machine learning approaches, are inspired by how the human brain functions
- **Spiking neural networks** mimic brains more closely by incorporating the concept of time (idea from the 1990s)



Biological realism

Many-core (ARM) architecture
Optimized spike communication network
Programmable local learning
x0.01 real-time to real-time

Full-custom-digital neural circuits
No local learning (TrueNorth)
Programmable local learning (Loihi)
Exploit economy of scale
x0.01 real-time to x100 real-time

Analog neural cores
Digital spike communication
Biological local learning
Programmable local learning
x1.000 real-time

Neuromorphic computers are available Akida (from Brainchip)

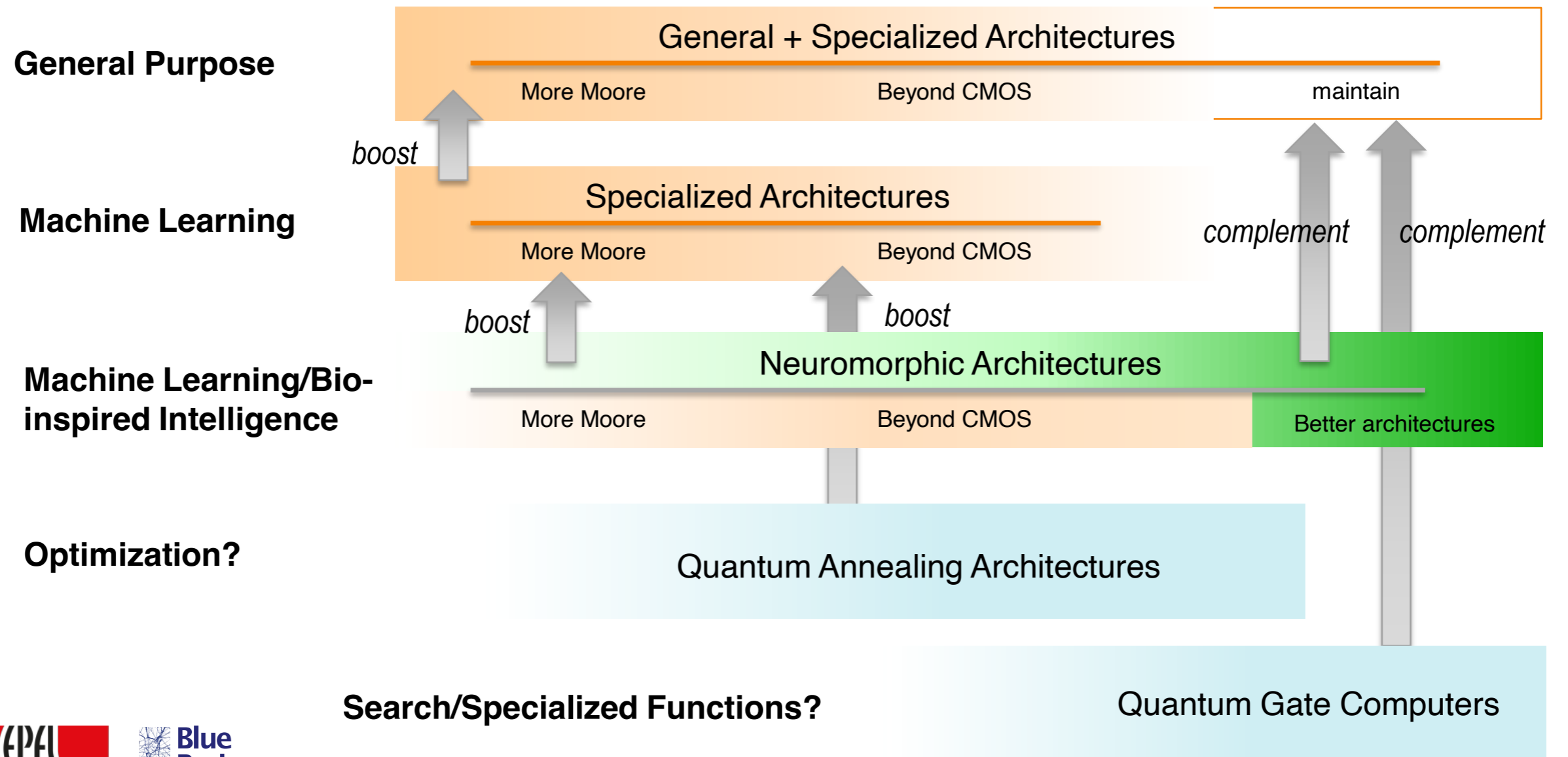
Quantum Computing?

- Initial ideas for quantum computing date back **40 years** (Benioff, Feynman, Manin, etc.)
 - Use quantum mechanical processes to **simulate quantum mechanical systems**
- Further interest was stimulated by the invention of **quantum algorithms** in the early 1980's with the promise of solutions to intractable problems on quantum computers (Shor, Grover, etc)
 - Exponential information storage
 - Revolutionize cryptography
 - Solutions to unsolved (classical) problems
- Most recently quantum computing has been in the news in regards to **quantum advantage** (supremacy)
 - Google, IBM, Jiuzhang
- Quantum computing is likely at the peak of its hype cycle

See lectures from Prof Petruccione

Potential View of the Hardware Future

Neuroscience and the Future of Computing



GPU Programming Taster

Modelled after [Charis Koraka](#)

Why can't I use python or C++?

- GPU architecture is very different from CPU
 - Many more **computation units**
 - Many **identical threads** executed in parallel
 - Identical operation on different data
 - Different operations require multiple passes
 - **Memory access** needs to be careful managed
 - Avoid register thrashing
 - Cannot handle **branching code**
 - **Latency hiding** techniques are required
- Typically need to control many low level instructions, e.g. memory organization/accesses, thread synchronization, data transfer

Current GPU Languages

CUDA

- Only for NVIDIA GPUs
- Well established, large suite of tools for analysis, profiling and debugging

HIP

- NVIDIA and AMD backends
- Almost identical to CUDA
 - `sed s/cu/hip/`

SYCL / dpc++

- Intel, NVIDIA and kinda AMD backends
- Explicit memory movement not required

Kokkos, Raja, Alpaka

- Interoperability APIs with backends for Intel, NVIDIA, AMD, and host parallel

Std::execution::parallel

- Introduced in C++17 standard
- Similar usage as `tbb::parallel_for`
- `nvc++` provides NVIDIA backend for GPUs

Language Evolution

- GPU languages have been **evolving rapidly** with the hardware
 - e.g. exploit new hardware functionality
 - new ways of sharing resources
 - proximity of CPU/GPU memory
- Standards also need to evolve as **C++ evolves**
- Actually you can program GPUs using C++ (17):
 - `std::execution::parallel`
 - Execution is currently synchronous + no low-level device control
- P2300 is a proposal for standard asynchronous programming: schedule, sender, receiver
 - Likely to be adopted in C++26
 - Additional changes to implementation details and API can be expected

Ways to use a GPU

Completely external package

- Tensorflow, PyTorch, Onnx, other ML network training
- Mathematica, Numba
- Bitcoin miner, Cyberpunk 2077, RDR2...

Use external function library

- cuBLAS, cuFFT, OptiX

Instrument CPU code with offloading directives

- openMP, openACC, `std::execution::parallel`

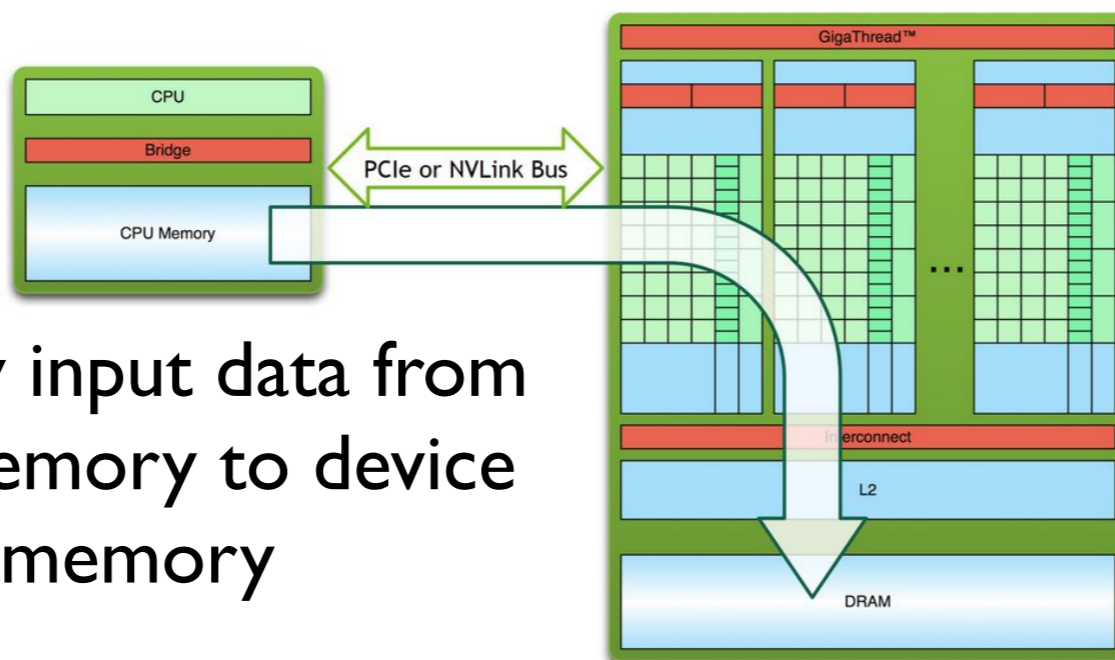
Write GPU kernels directly

- CUDA, SYCL, HIP
- Kokkos, Raja, Alopaka

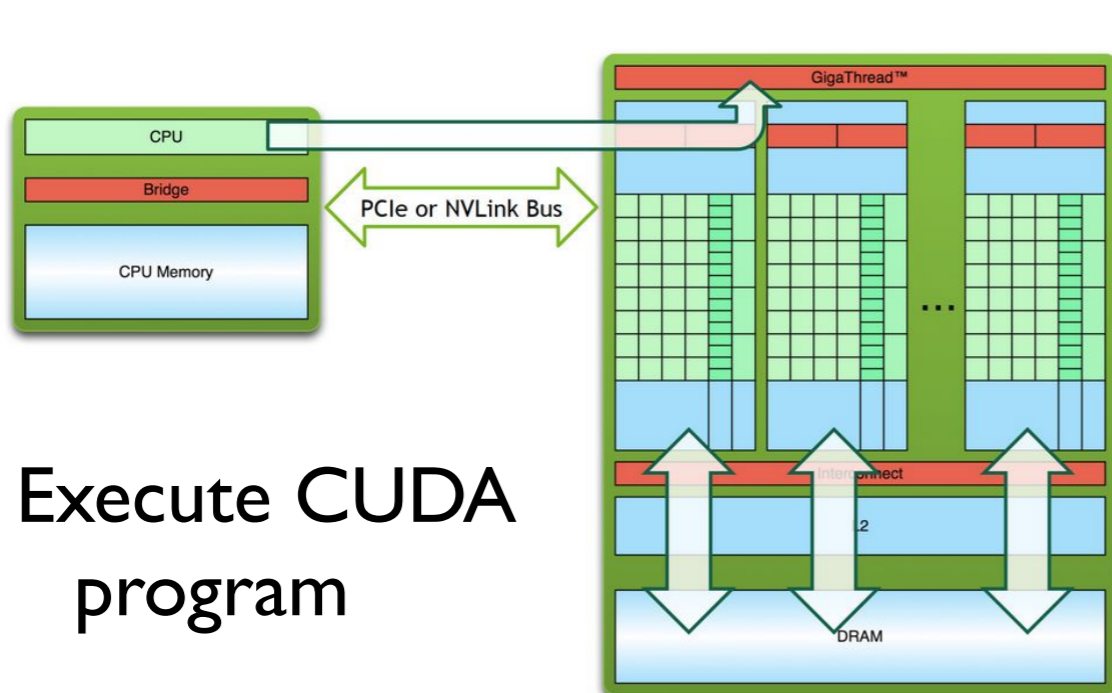
CUDA Programming Model

- Compute Unified Device Architecture (**CUDA**) is a program language developed by NVidia and used to develop applications on NVidia GPUs
- **Three main steps** to execute CUDA code

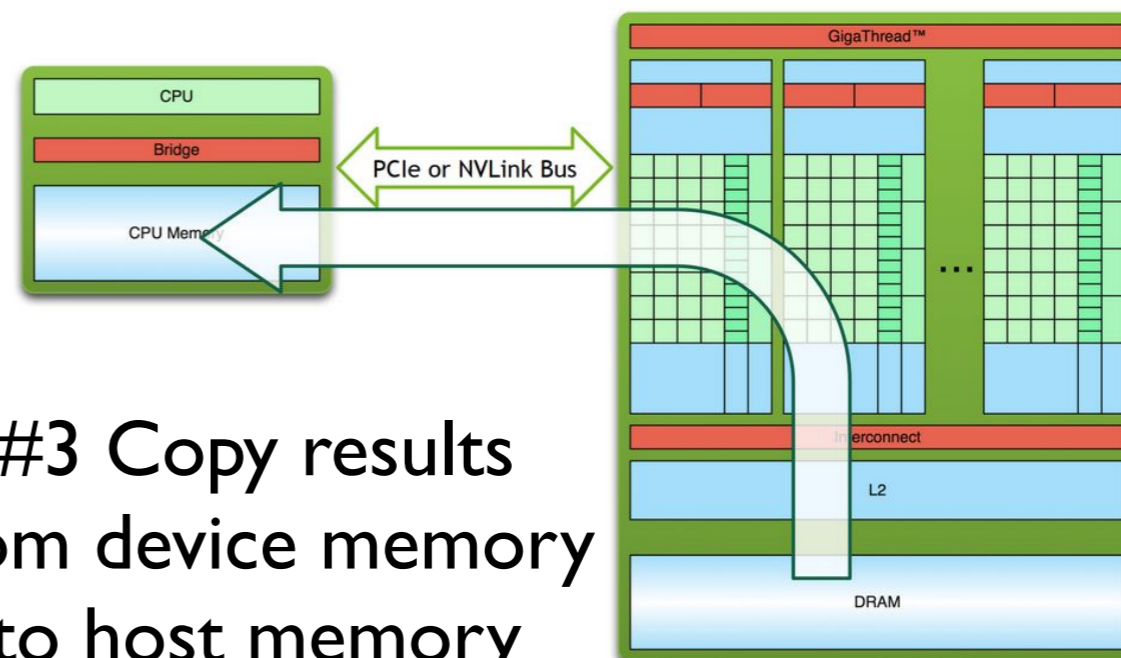
#1 Copy input data from host memory to device memory



#2 Execute CUDA program

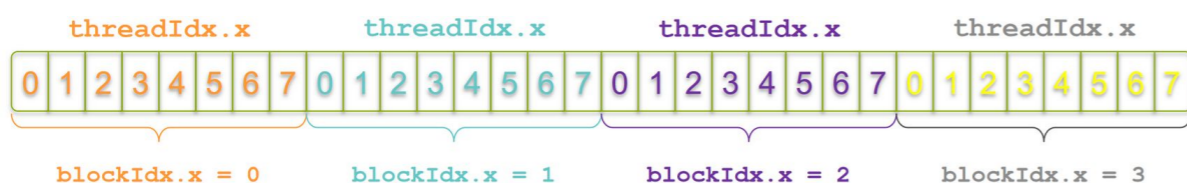
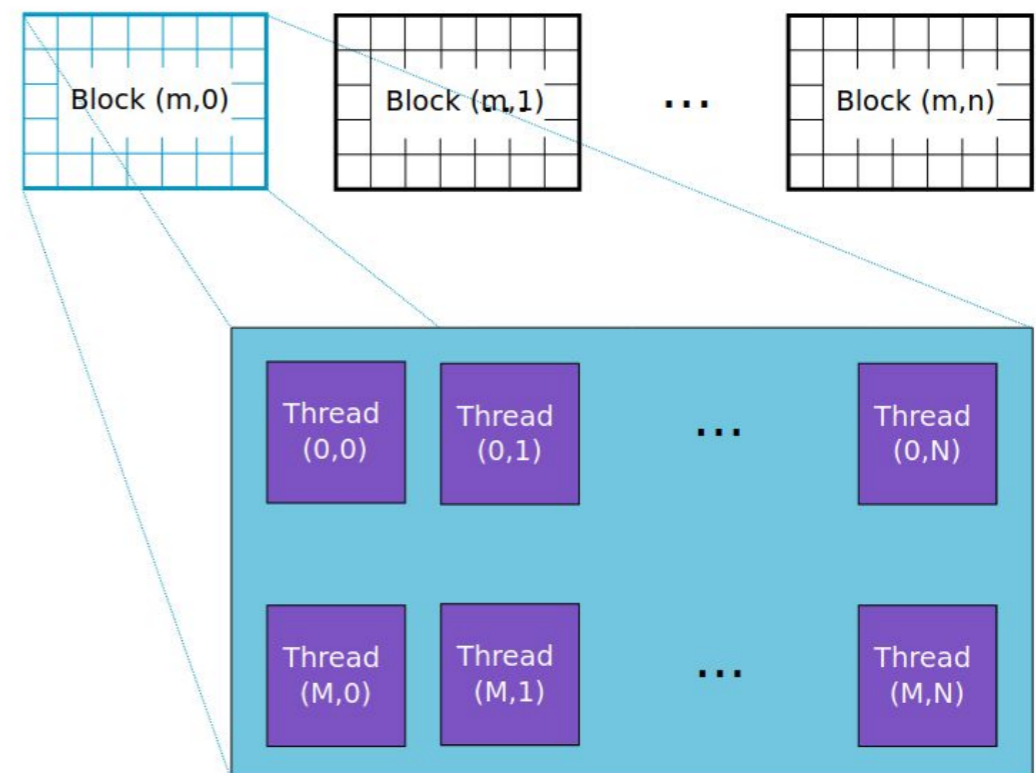
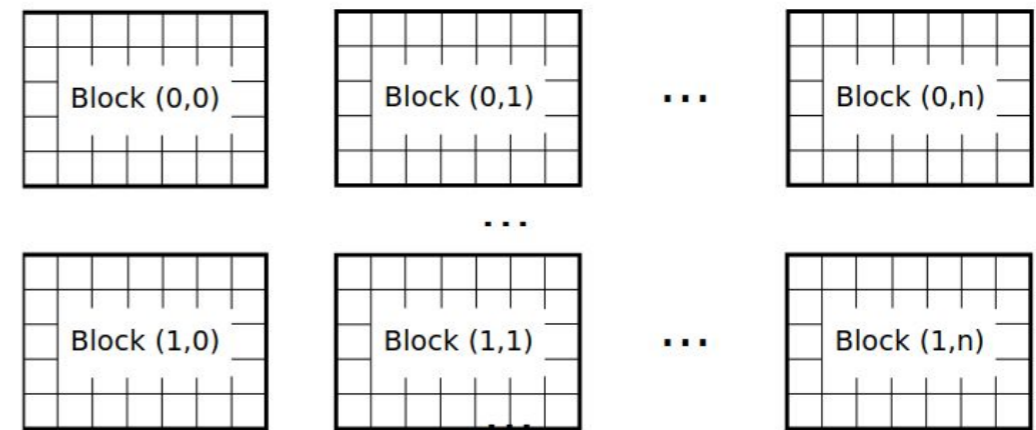


#3 Copy results from device memory to host memory



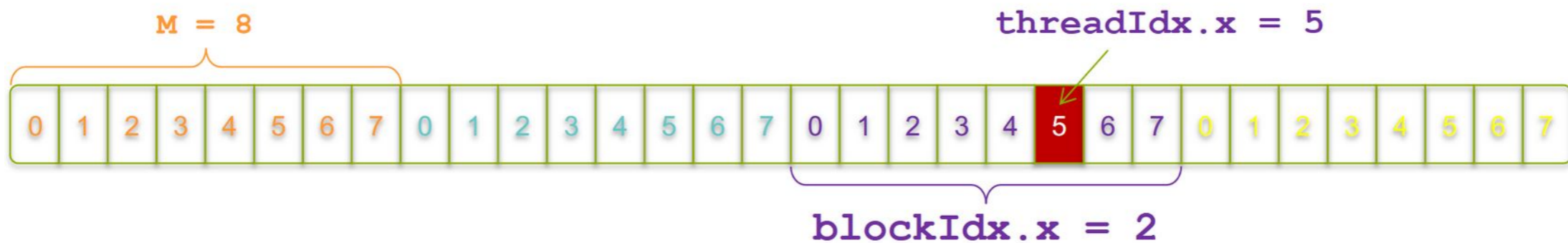
Threads and Blocks

- Built-in variables are available for **threads** and **blocks**
 - `threadIdx` & `blockIdx`
- 3-dimensional indexing can be used to express vectors and matrices
 - `threadIdx.x`,
`threadIdx.y`, `threadIdx.z`
- CUDA architecture imposes a limit of **1024 threads per block**
- Thread block dimension is accessible within the kernel using `blockDim`



Indexing Example

- **Unique element index** can be expressed using the `threadIdx` & `blockIdx` variables
 - e.g. `index = threadIdx.x + M * blockIdx.x`
 - (if each block consists of M threads)



```
int index = threadIdx.x + blockIdx.x * M;
          =          5   +          2   * 8;
          = 21;
```

Kernels & Functions

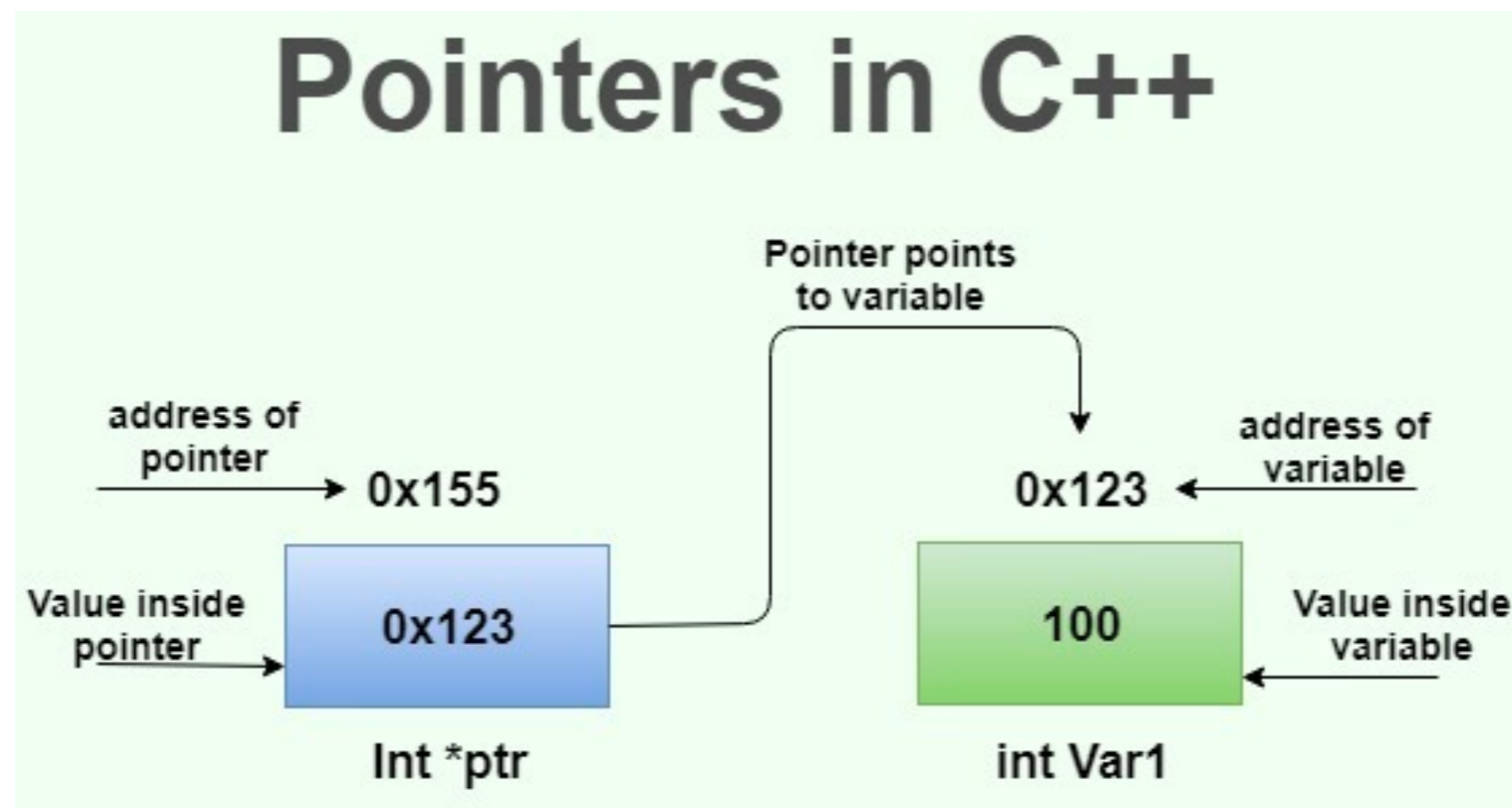
- A CUDA **kernel** is a function that gets executed on the GPU
- Contains the part of the application that is parallelizable
- Will be executed many times in parallel by different CUDA threads

Declaration	Callable from:	Executed on:
<code>__global__</code>	host	device
<code>__device__</code>	device	device
<code>__host__</code>	host	host

- `__global__` keyword defines a kernel function
 - Launched by host and executed on device
- `__device__` and `__host__` can be used together
- `__host__` declaration can be omitted if used alone

Reminder: Pointers

- Pointers were covered in the C++ refresher slides from C. Doglioni
- GPU programming requires careful memory management



Source

Example CUDA Program

- Main components of a CUDA program
 - **Function** declarations
 - `__host__` / `__global__` / `__device__`
 - Copying **data** to/from host
 - `cudaMalloc`/`cudaMemcpy`/`cudaFree`
 - Kernel **launch** `<<<nBlocks, nThreads>>>`(`<arguments>`)
 - **Concurrency** management
 - `__syncthreads()` / `CudaDeviceSynchronize()`

```

__global__ void do_something (int* a) {
    *a = 2;
}

int main() {
    int* a;
    int* d_a;
    // Host copy of variable a
    a = (int*) malloc(sizeof(int));
    // Device copy of variable a
    cudaMalloc(&d_a, sizeof(int));
    // Set the host value of a
    *a = 1;
    // Copy the value of a to the device
    cudaMemcpy(d_a, a, sizeof(int), cudaMemcpyHostToDevice);
    // Launch the kernel to set the value
    do_something<<<1,1>>>(d_a);
    cudaDeviceSynchronize();
    // Copy the value of a back to the host
    cudaMemcpy(a, d_a, sizeof(int), cudaMemcpyDeviceToHost);
    // Free the allocated memory
    free(a);
    cudaFree(d_a);
}

```

Simple Example: saxpy in C++

- ▶ Traditional single precision matrix addition example:

$$\mathbf{Z} = \mathbf{a} * \mathbf{X} + \mathbf{Y}$$

```
void saxpy(int n, float a, float *x, float *y) {
    for (int i=0; i<n; ++i) {
        y[i] = a*x[i] + y[i];
    }
}

main() {
    int N = 1 << 20;
    float *x, float *y;
    x = new float[N]; y = new float[N];
    for (int i=0; i<N; ++i { x[i] = 1.0f; y[i] = 2.0f; }

    saxpy(N, 3.0f, x, y);

    float maxErr = 0.0f;
    for (int i=0; i<N; ++i) {maxErr=std::max(maxErr, std::fabs(maxErr-5.0f));}
    std::cout << "max Error:" << maxErr << std::endl;

    delete [] x; delete [] y;
}
```


Simple Example: saxpy in OpenMP

- ▶ OpenMP

kernel pragma

- ▶ this pragma does not offload to GPU but rather across cores/threads

- ▶ can offload with openMP, but much more complicated than with openACC

```
void saxpy(int n, float a, float *x, float *y) {  
    #pragma omp parallel for  
    for (int i=0; i<n; ++i) {  
        y[i] = a*x[i] + y[i];  
    }  
}  
main() {  
    int N = 1 << 20;  
    float *x, float *y;  
    x = new float[N]; y = new float[N];  
    for (int i=0; i<N; ++i) { x[i] = 1.0f; y[i] = 2.0f; }  
  
    saxpy(N, 3.0f, x, y);  
  
    float maxErr = 0.0f;  
    for (int i=0; i<N; ++i) {maxErr=std::max(maxErr, std::fabs(maxErr-5.0f));}  
    std::cout << "max Error:" << maxErr << std::endl;  
  
    delete [] x; delete [] y;  
}
```

Simple Example: saxpy in OpenACC

- ▶ OpenACC
- ▶ requires special compilers that recognize openACC pragmas, eg PGI

```
void saxpy(int n, float a, float *x, float *y) {  
    #pragma acc parallel loop  
    for (int i=0; i<n; ++i) {  
        y[i] = a*x[i] + y[i];  
    }  
}  
main() {  
    int N = 1 << 20;  
    float *x, float *y;  
    x = new float[N]; y = new float[N];  
    for (int i=0; i<N; ++i { x[i] = 1.0f; y[i] = 2.0f; }  
  
    saxpy(N, 3.0f, x, y);  
  
    float maxErr = 0.0f;  
    for (int i=0; i<N; ++i) {maxErr=std::max(maxErr,std::fabs(maxErr-5.0f));}  
    std::cout << "max Error:" << maxErr << std::endl;  
  
    delete [] x; delete [] y;  
}
```

Simple Example: saxpy in CUDA

▶ CUDA

explicit memory management

▶ compile with nvcc

number of blocks

number of threads per block

```

__global__
void saxpy(int n, float a, float *x, float *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

main() {
    int N = 1 << 20;
    float *x, float *y, float *d_x, float *d_y;
    x = new float[N]; y = new float[N];
    for (int i=0; i<N; ++i) { x[i] = 1.0f; y[i] = 2.0f; }
    cudaMalloc(&d_x, N*sizeof(float)); cudaMalloc(&d_y, N*sizeof(float));
    cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
    saxpy<<<(N+255)/256, 256>>>(N, 3.0f, d_x, d_y);
    cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

    float maxErr = 0.0f;
    for (int i=0; i<N; ++i) {maxErr=std::max(maxErr, std::fabs(maxErr-5.0f));}
    std::cout << "max Error:" << maxErr << std::endl;

    cudaFree(d_x); cudaFree(d_y)
    delete [] x; delete [] y;
}

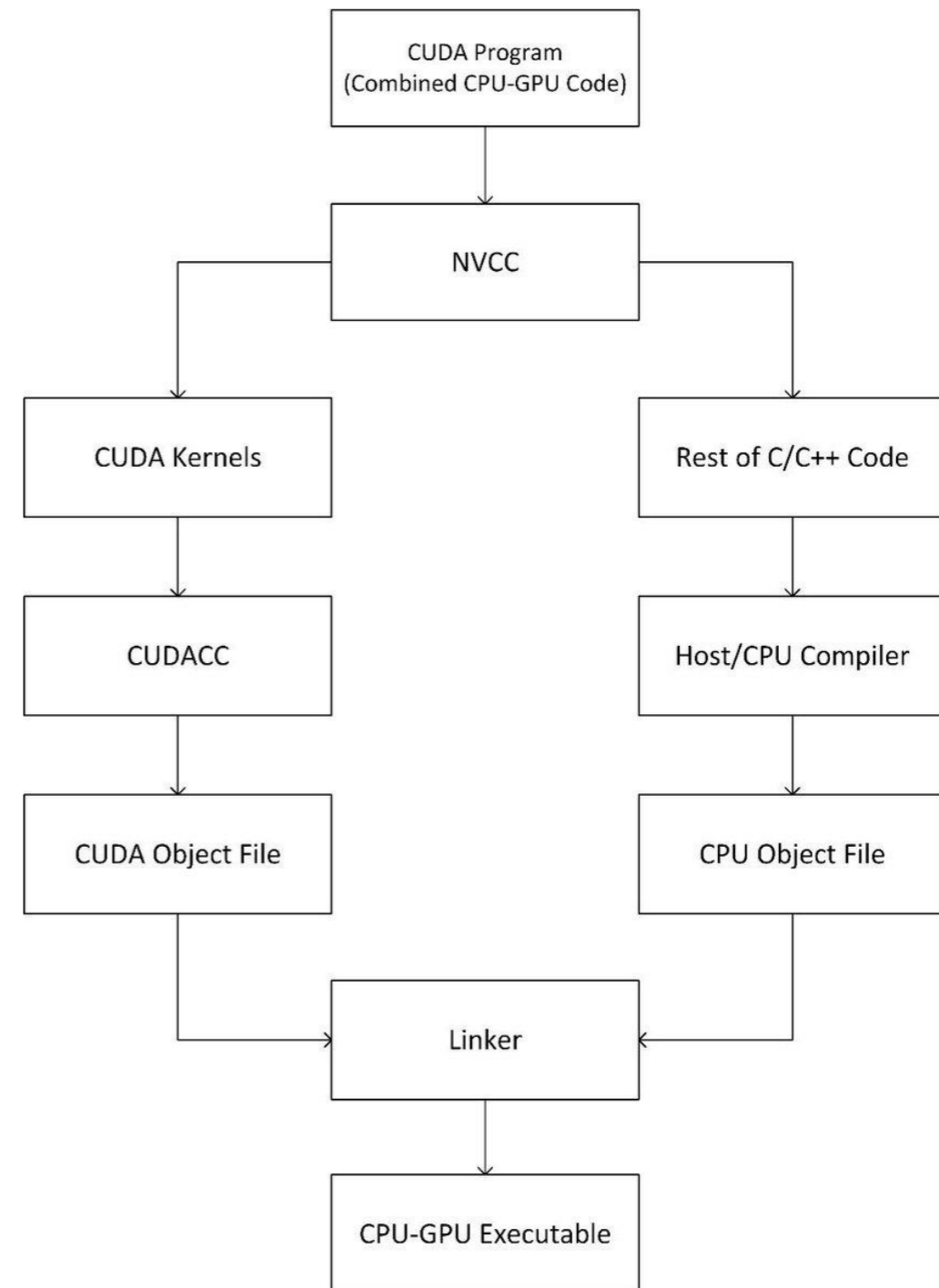
```

Memory Management

- Host and device have **separate memory**
 - Device = GPU memory
 - Host = CPU memory
- CUDA **kernels** operate out of **device** memory
- CUDA provides functions to
 - **allocate** device memory
 - `cudaMalloc (&ptr, size_in_bytes_to_allocate)`
 - **release** device memory
 - `cudaFree (ptr)`
 - **transfer data** between host and device memory
 - `cudaMemcpy (destination_ptr, source_ptr, size_in_bytes, direction)`

Compilation

- CUDA programs are **compiled** in a similar way to C++ program
- Store CUDA code in a file with a .cu extension
- CUDA compiler is **nvcc** (provided by Nvidia)
 - nvcc for CUDA parts
 - gcc for C++ parts
 - nvcc converts .cu files into C++ for the host system and CUDA assembly of binary instructions for the device
- `nvcc myCUDAProgram.cu -o myCUDAProgram`

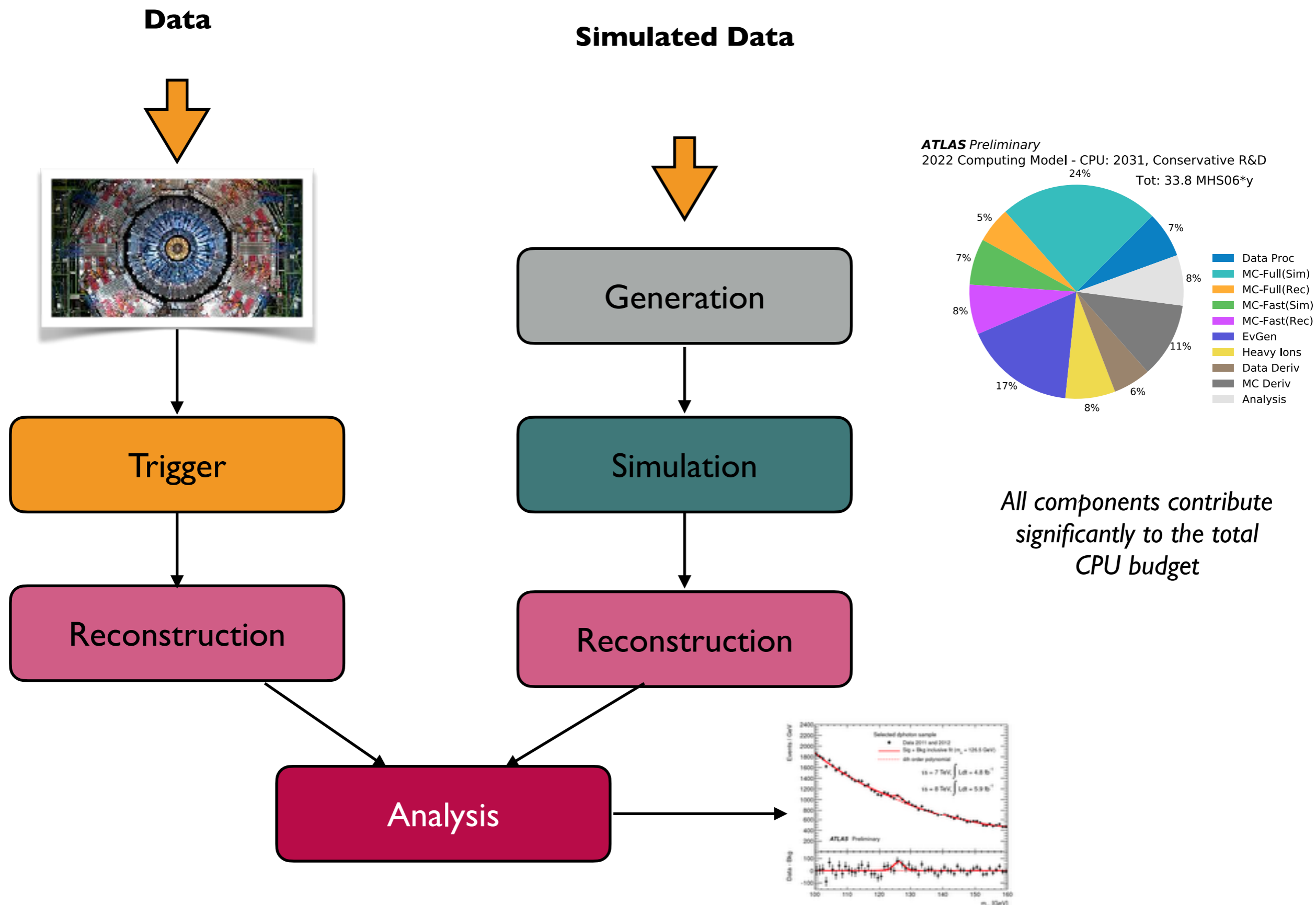


Part 2: Application of Novel Computing Hardware to HEP

Novel Hardware in HEP

- So far, we've seen how the type of computing hardware available to obtain optimal performance has been evolving towards more heterogeneous architectures
- We've had a look at the different types of hardware that are used to construct these machines
- Next, we are going to look into examples of how such architectures are being used in HEP

Typical HEP Computing Workflow



Event Generation

Event Generation

- Event generation is a natural candidate for parallelization because the generation of **different elements** in the particle tree is **largely independent***
- As particles are produced, they can be handled by different threads
- Scheduling challenge: different threads do not necessarily have the same execution time
- Madgraph is an event generator commonly used in HEP
- **Madgraph4gpu** is a project to port Madgraph to run on GPUs
- A new generator, Pepper, has been developed specifically targeting GPUs

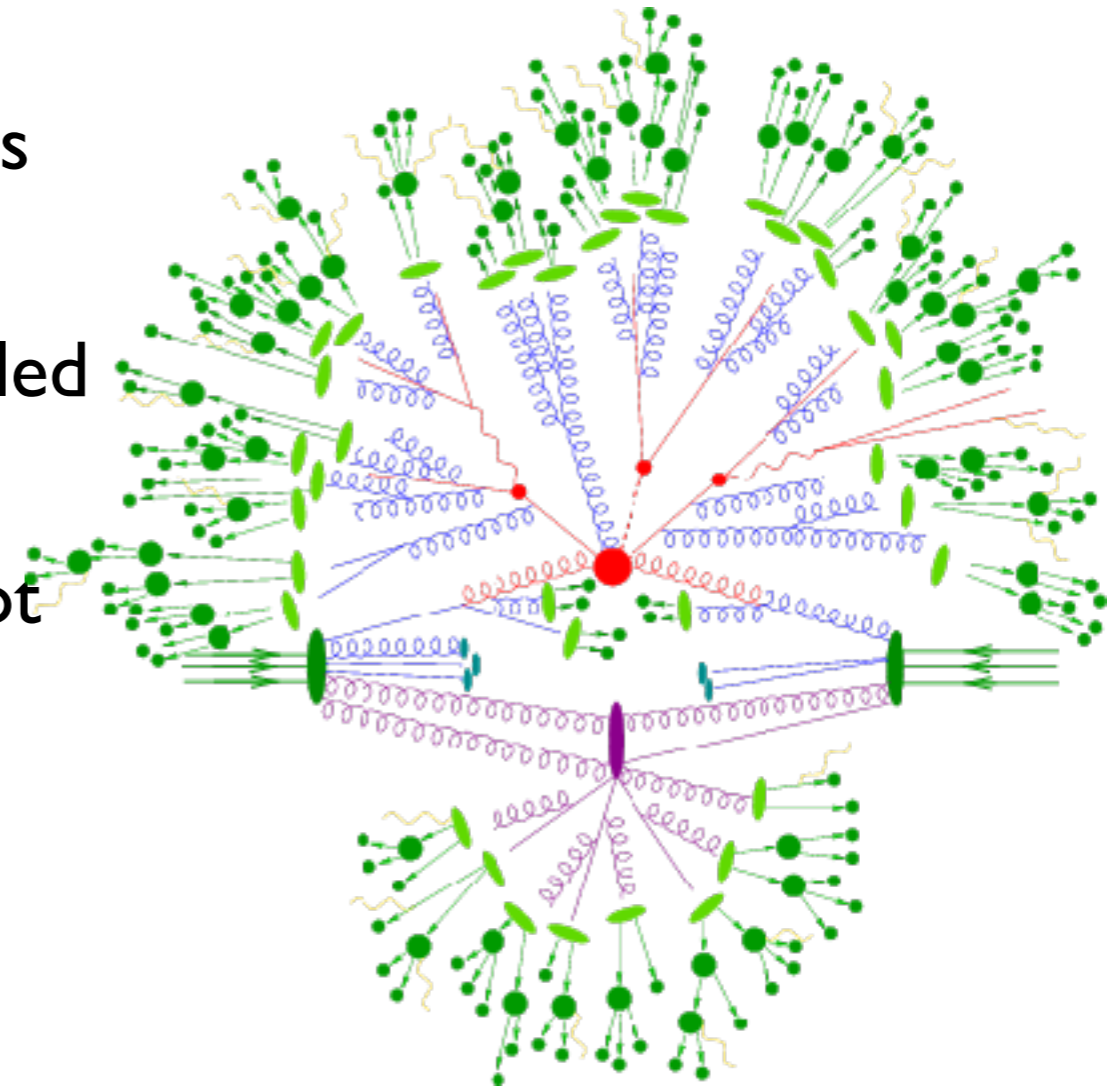

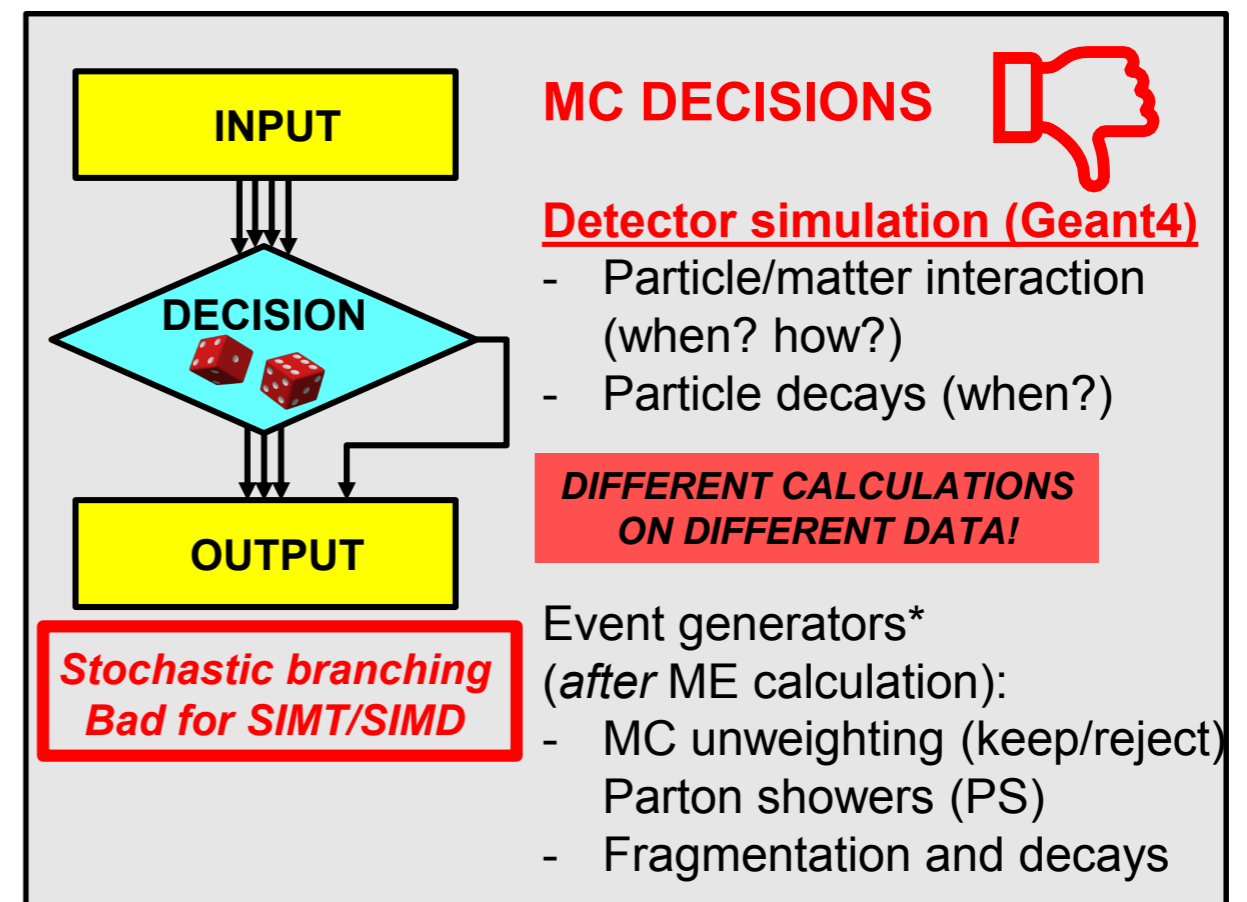
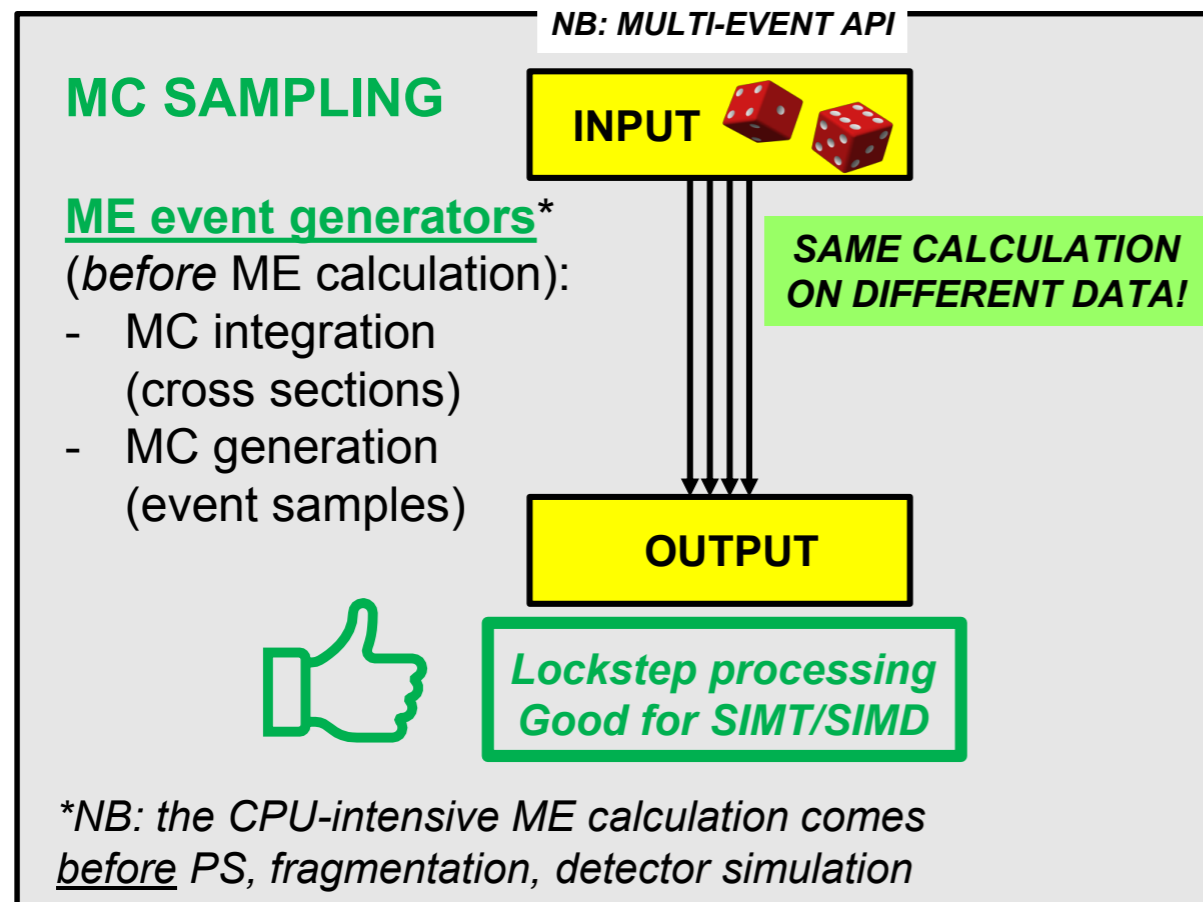


Image Credit

*Strictly speaking this is not fully correct due to effects like color reconnection

ANY MC event generator is a great fit for GPUs and vector CPUs!

- Monte Carlo methods are based on drawing (pseudo-)random numbers: a dice throw 
- From a software workflow point of view, these are used in *two rather different cases*:



Madgraph on GPUs

- Madgraph runs in Fortran
 - 95% of CPU time used for the **matrix element calculation** (for a complex process)
- Madgraph4gpu project targets the matrix element calculation by porting it to run on GPUs and vector CPUs

Process	Madevent 262 144 events			Standalone CUDA
	Total	Momenta+unweight	Matrix elm	ME Throughput
$e^+e^- \rightarrow \mu^+\mu^-$	17.9 s	10.2 s	7.8 s	$1.9 \times 10^6 \text{s}^{-1}$
+CUDA Tesla A100	10.0 s	10.0 s	0.02s	$633.8 \times 10^6 \text{s}^{-1}$
	1.8 x	1.0 x	390 x	334 x
$gg \rightarrow t\bar{t}gg$	209.3 s	7.8 s	201.5 s	$2.8 \times 10^3 \text{s}^{-1}$
+CUDA Tesla A100	8.4 s	7.8 s	0.6 s	$758.9 \times 10^3 \text{s}^{-1}$
	24.9 x	1.0 x	336 x	271 x
$gg \rightarrow t\bar{t}ggg$	2507.6 s	12.2 s	2495.3 s	$1.1 \times 10^2 \text{s}^{-1}$
+CUDA Tesla A100	30.6 s	14.1 s	16.5 s	$170.7 \times 10^2 \text{s}^{-1}$
	82.0 x	0.9 x	151 x	155 x



- Using CUDA, achieve a speed up of 150-300x for the ME for complex events
- Also, significant speed up on vectorized CPUs

Simulation

Simulation

- Detector simulation tracks the passage of thousands of particles through the detector
 - Simulate interactions with **detector material**
 - Interactions can produce **new particles**, which subsequently need to be tracked

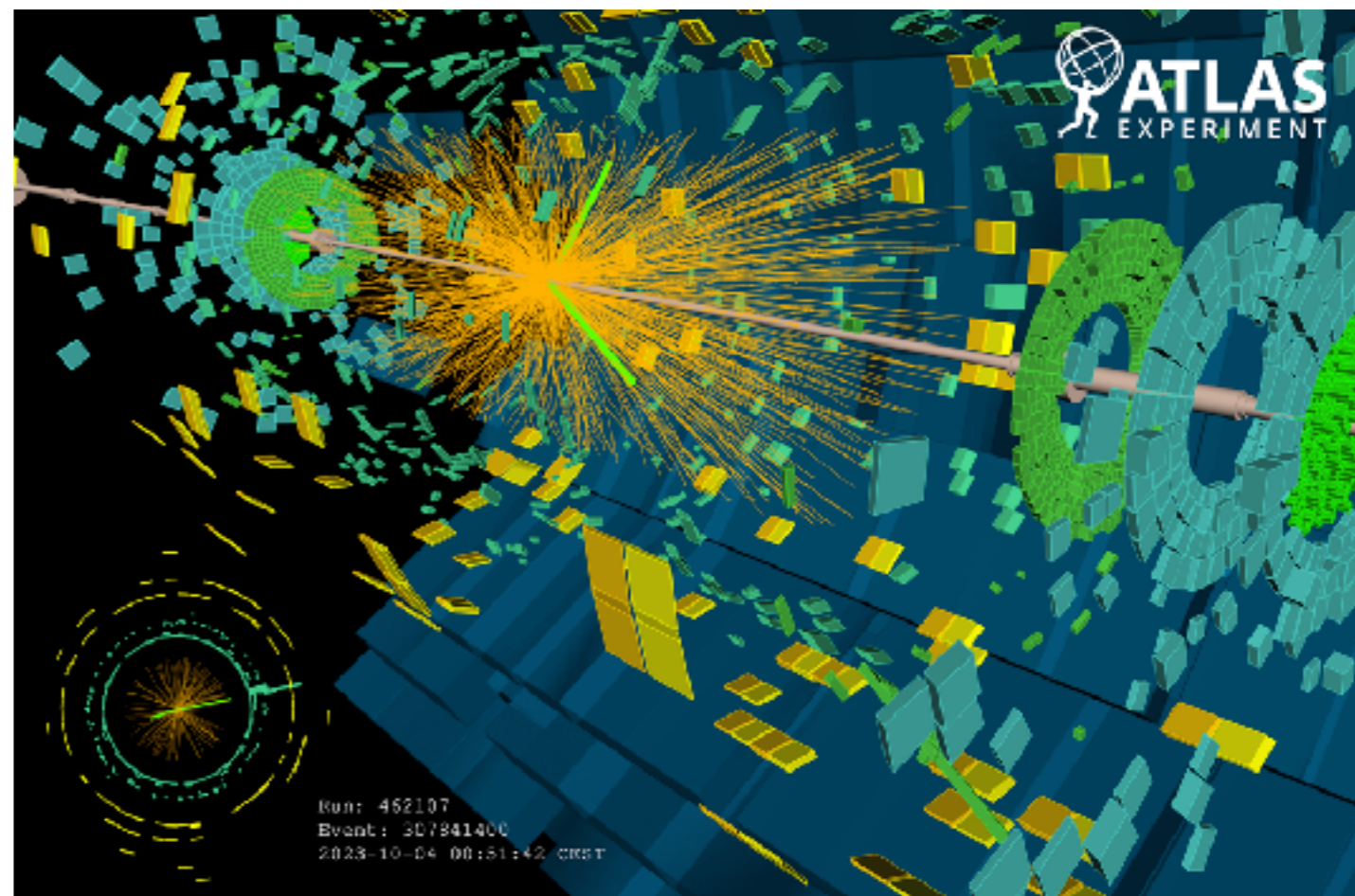


Image Credit

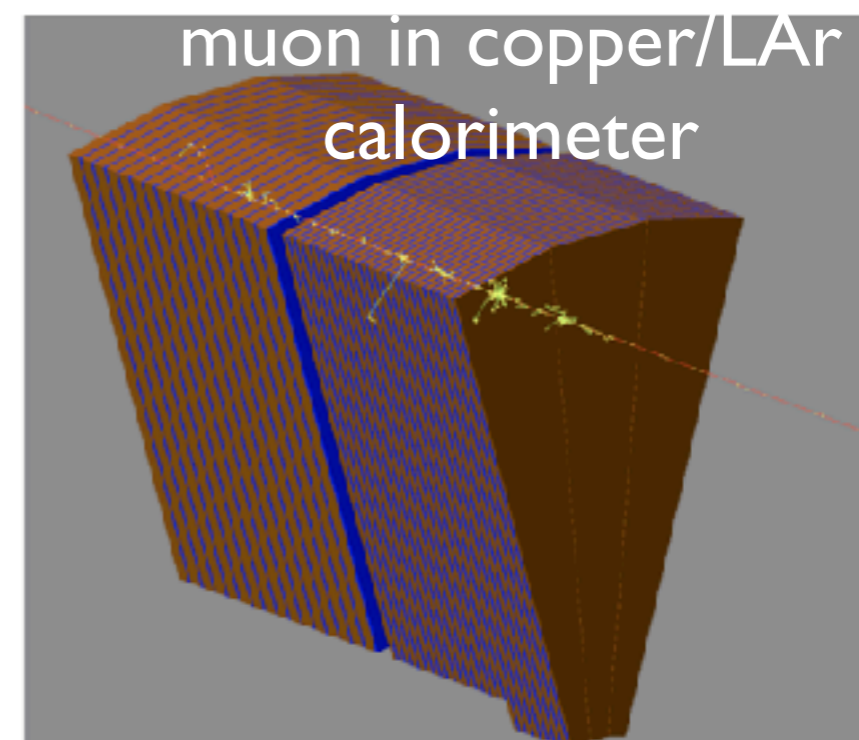
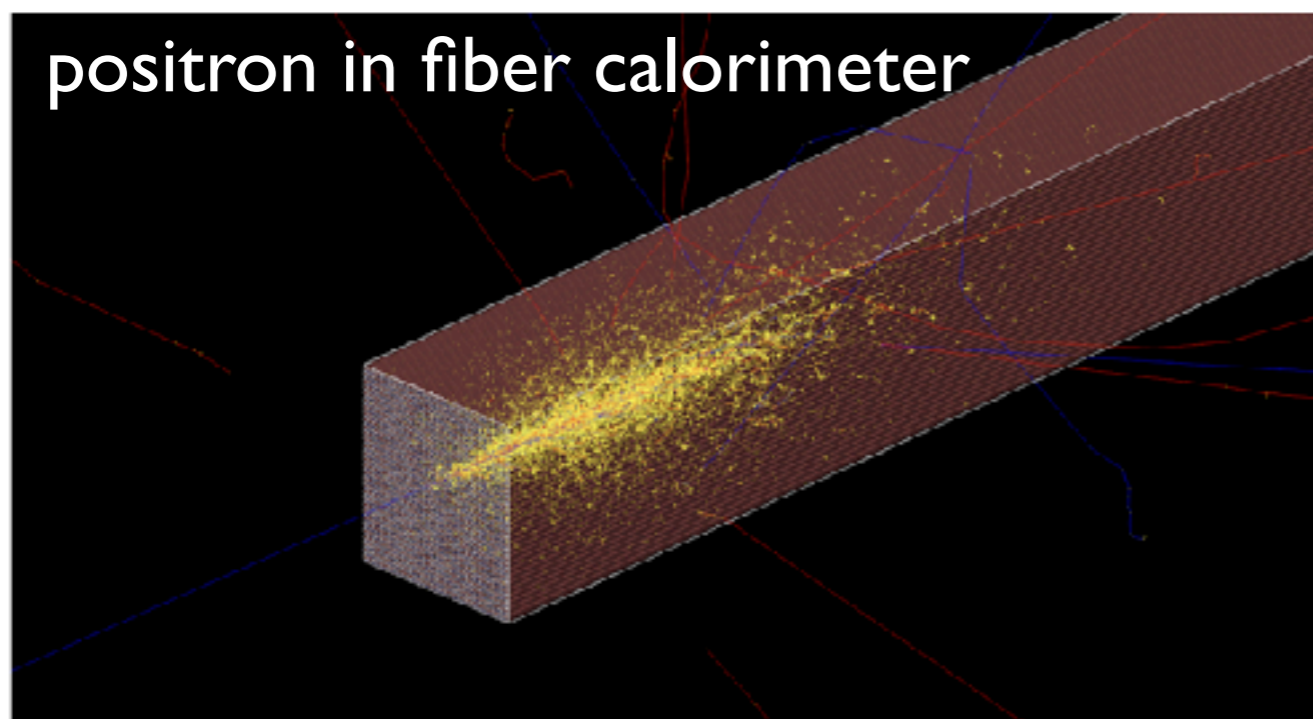
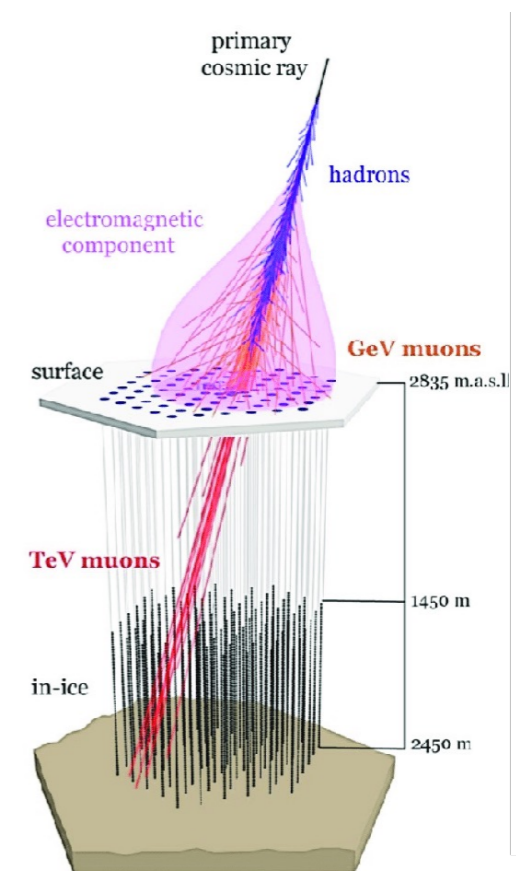
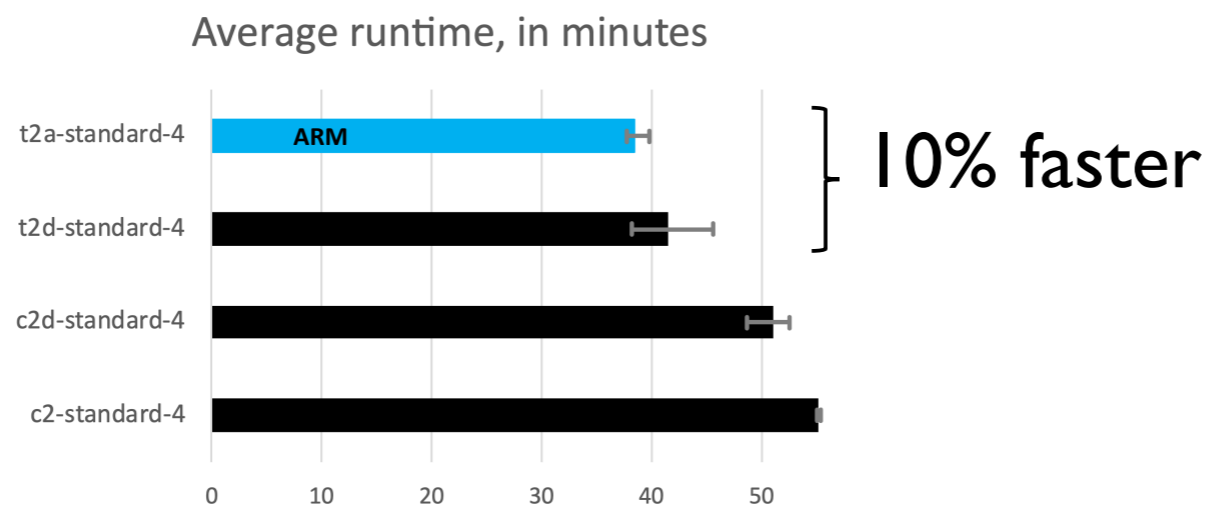


Image Credit

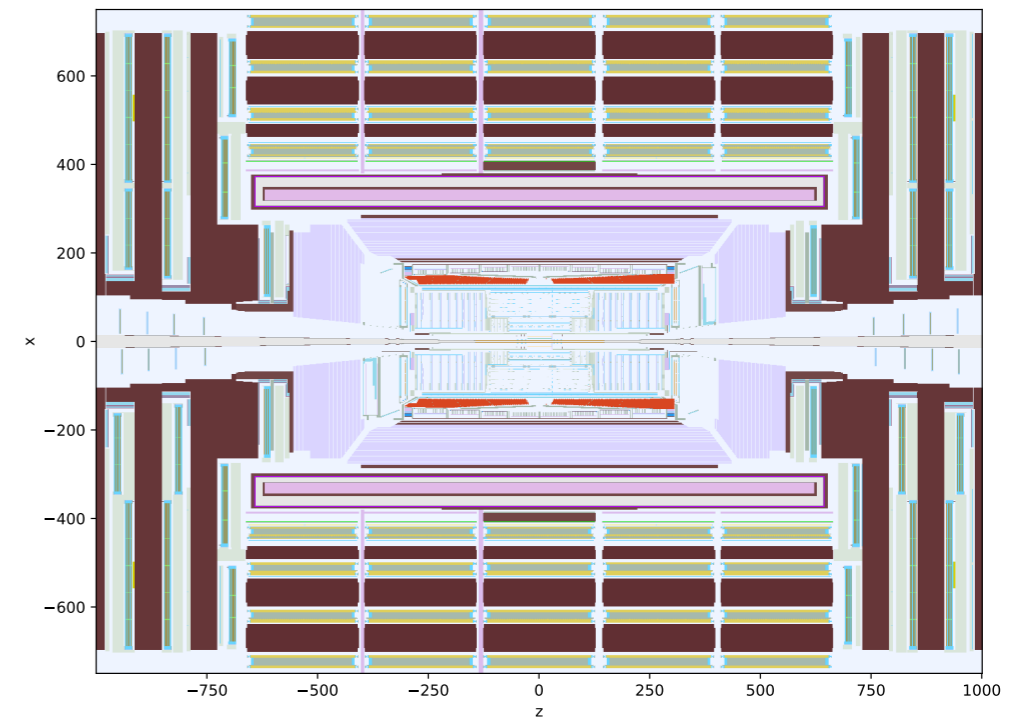
Simulation with Accelerators

- Most accurate tool for detector simulation is the Geant4 toolkit
 - Multithreading and task-based parallelization are available
 - Projects to port computationally intensive components of G4 to GPU are ongoing
- Many fast simulation tools with varying degrees of accuracy are available
 - Parametrization (DELPHES), machine learning (FastCaloGAN)
- Relevant examples include
 - IceCube simulation on ARM
 - ATLAS Fast Calorimeter simulation on GPUs



Celeritas: EM Physics on GPUs

- Equivalent to `G4EmStandardPhysics`
...using Urban MSC for high-E MSC; only γ , e^\pm
- Full-featured Geant4 detector geometries using VecGeom
- Runtime selectable processes, physics options, field definition
- Execution on CUDA (Nvidia), HIP* (AMD), and CPU devices

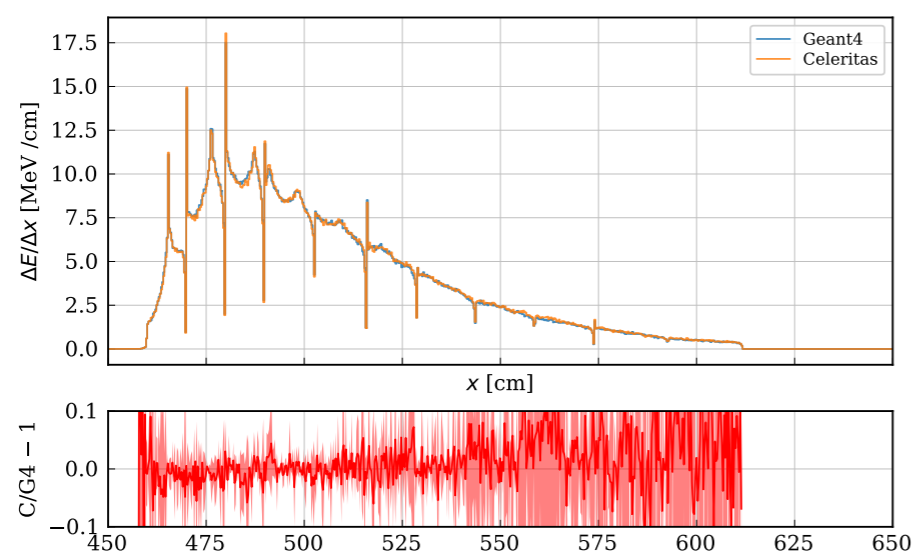
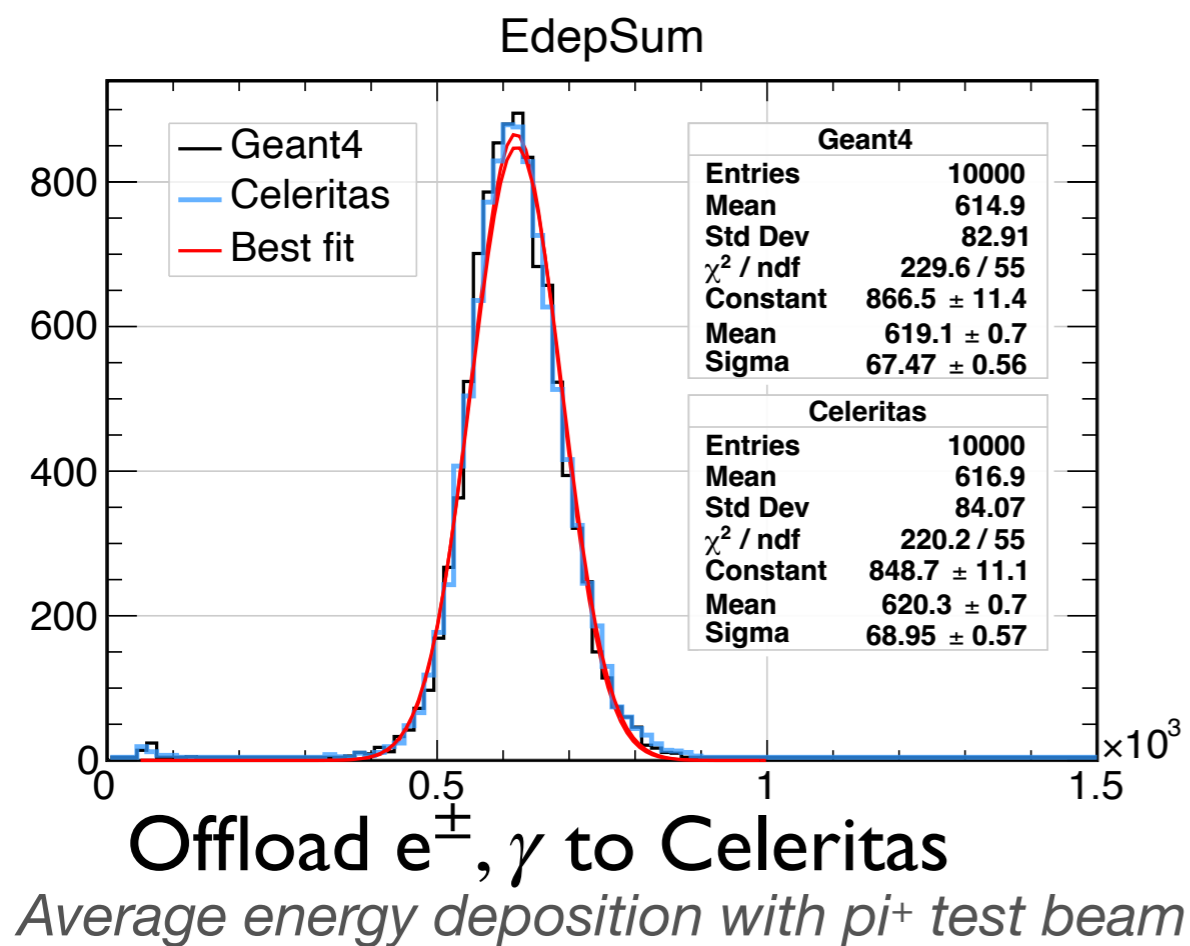


GPU-traced rasterization of CMS 2018

**VecGeom is incompatible with HIP:
ORANGE GPU prototype used instead*

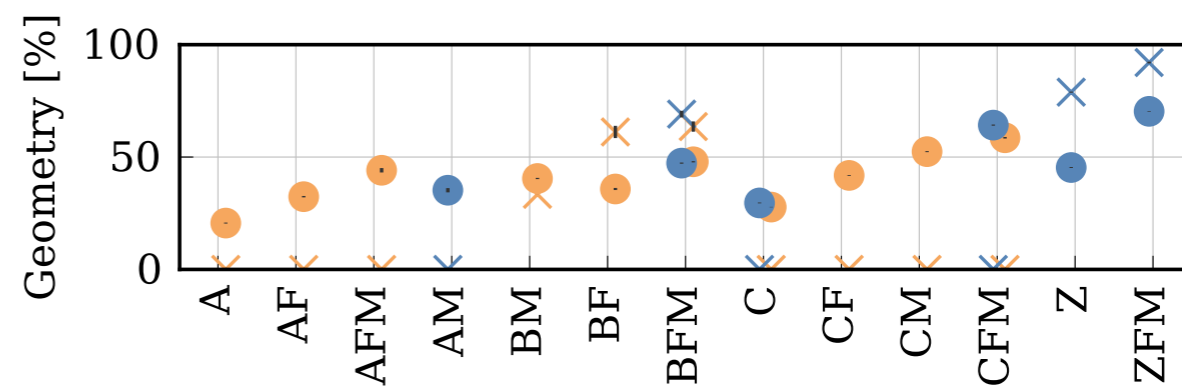
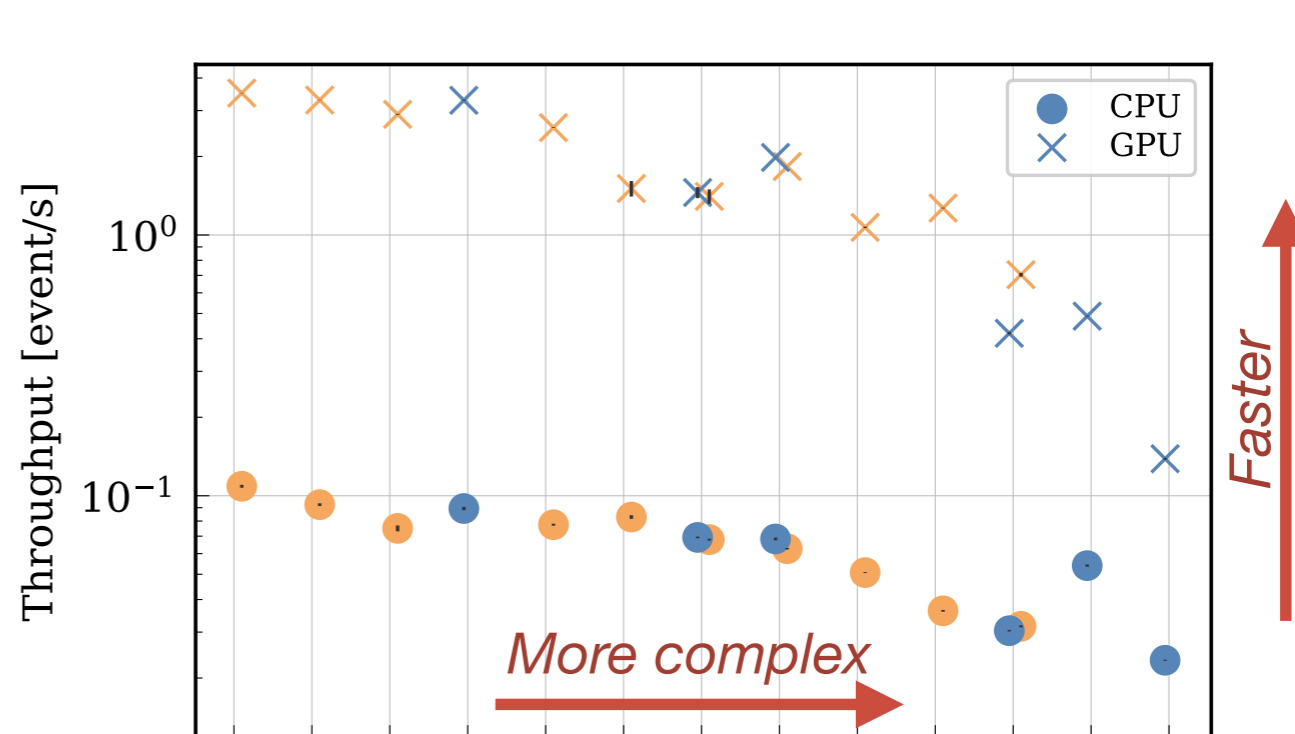
Celeritas Results

ATLAS Tile Calorimeter Test Beam



Slab-integrated energy deposition

Performance on SUMMIT



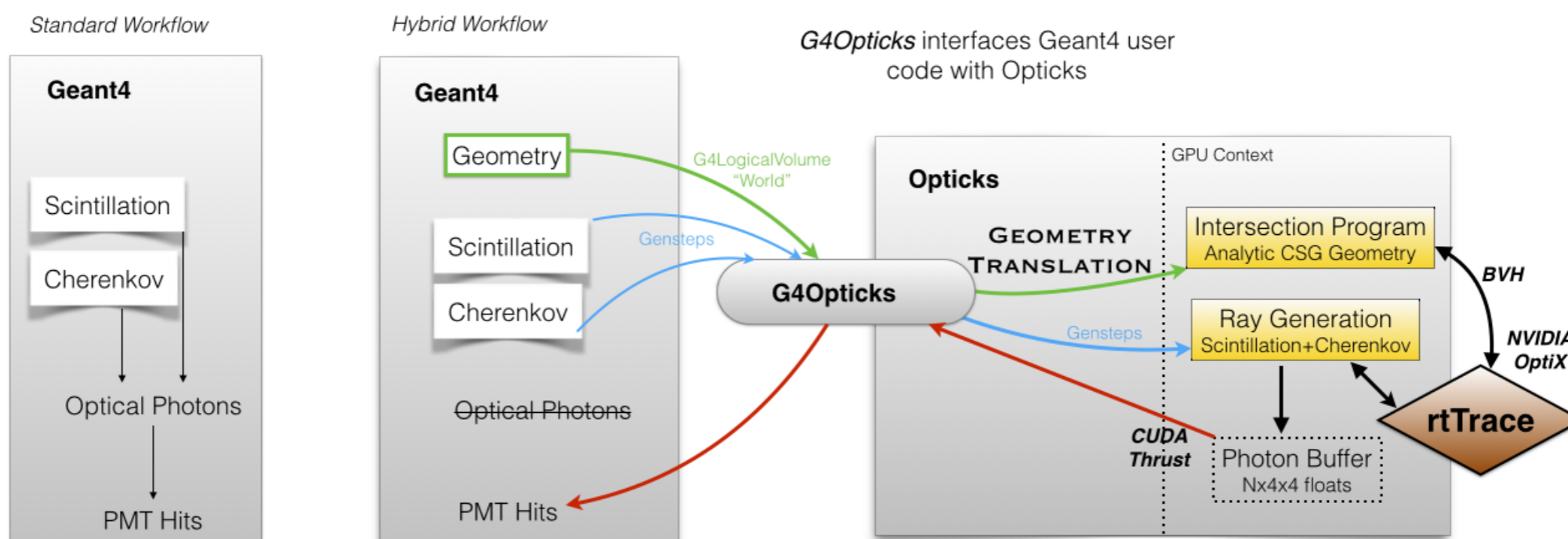
Source

NVIDIA OptiX

- Ray tracing is a technique commonly used in computer graphics
- NVIDIA OptiX is an API for ray tracing, which offloads ray tracing computations to GPUs
- Ray tracing algorithms can also be used for photon simulation
 - Opticks (and G4Opticks Package)



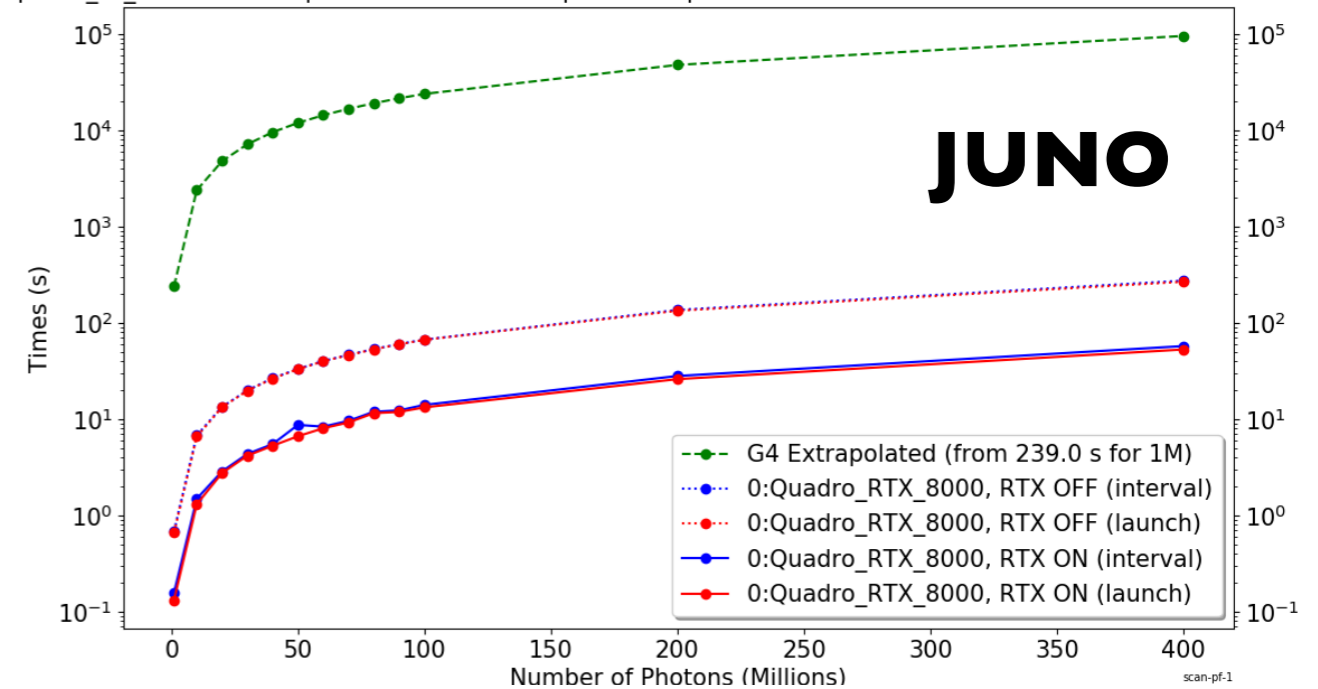
Source



Opticks for Photon Simulation

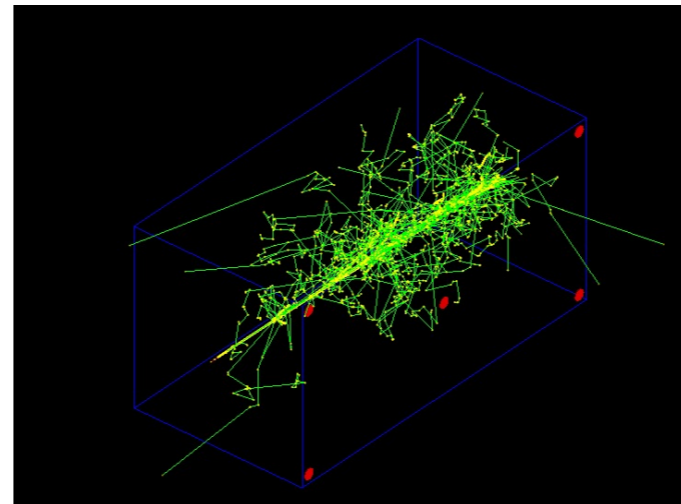
- Using Opticks, JUNO obtains 3 orders of magnitude improvement (wrt single threaded CPU)
- For DUNE, the speed up is 189x for the LArTPC

Opticks_vs_Geant4 : Extrapolated G4 times compared to Opticks launch+interval times with RTX mode ON and OFF



DUNE Performance:

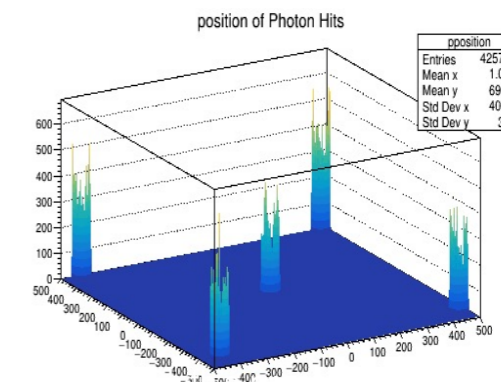
Hardware:	
CPU	Intel® Core i9-10900k@ 3.7 GHz, 10 CPU cores
GPU	NVIDIA GeForce RTX 3090 @ 1.7 GHz, 10496 cores
Software:	
Geant4: 11.0, Opticks based on OptiX® 6	



[Source](#)

Number of CPU threads	Geant4 [sec/evt]	Opticks [sec/evt]	Gain/speed up
1	330	1.8	189x

→ It becomes feasible to run full optical simulation event by event!

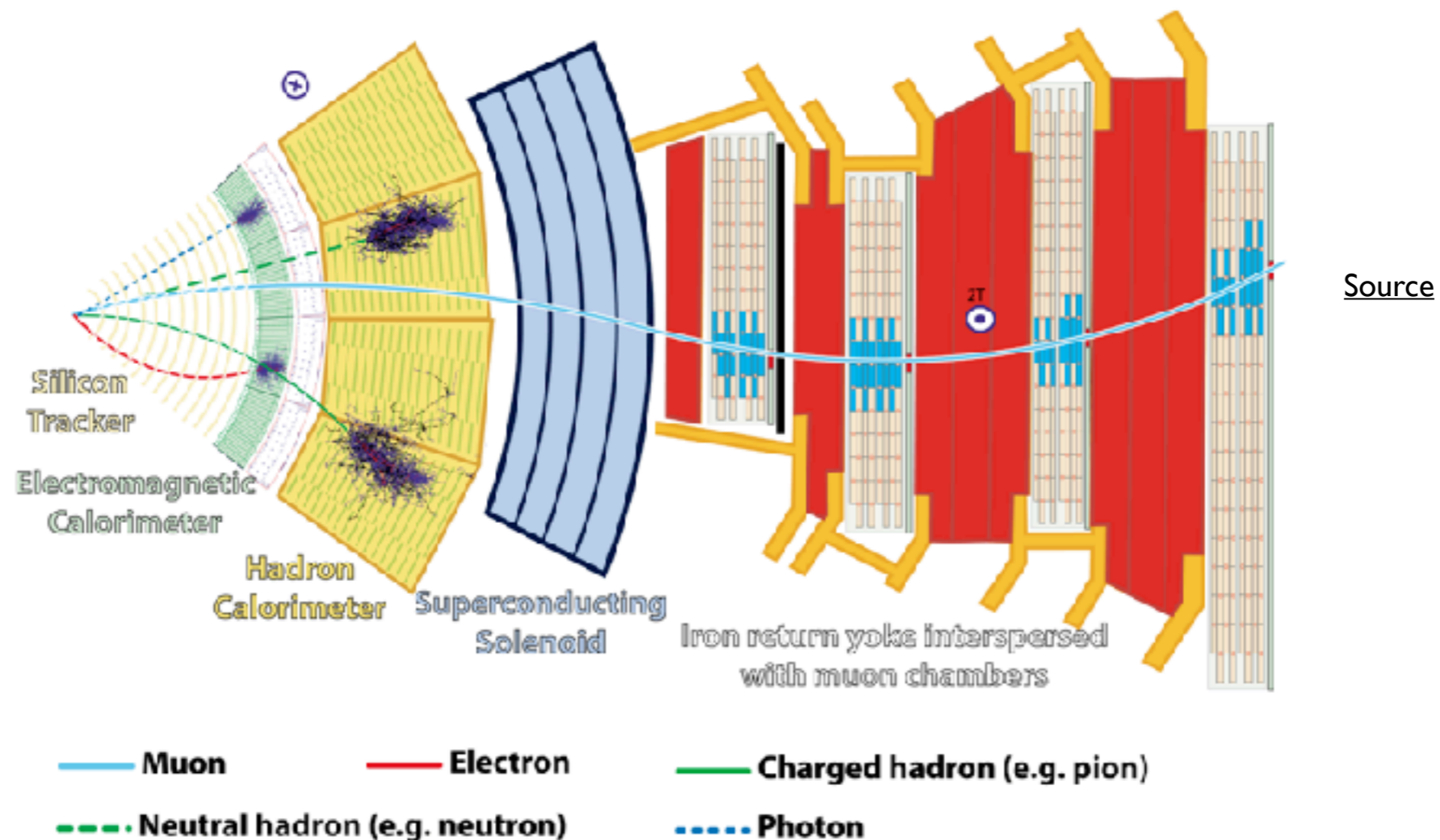


[Source](#)

Reconstruction

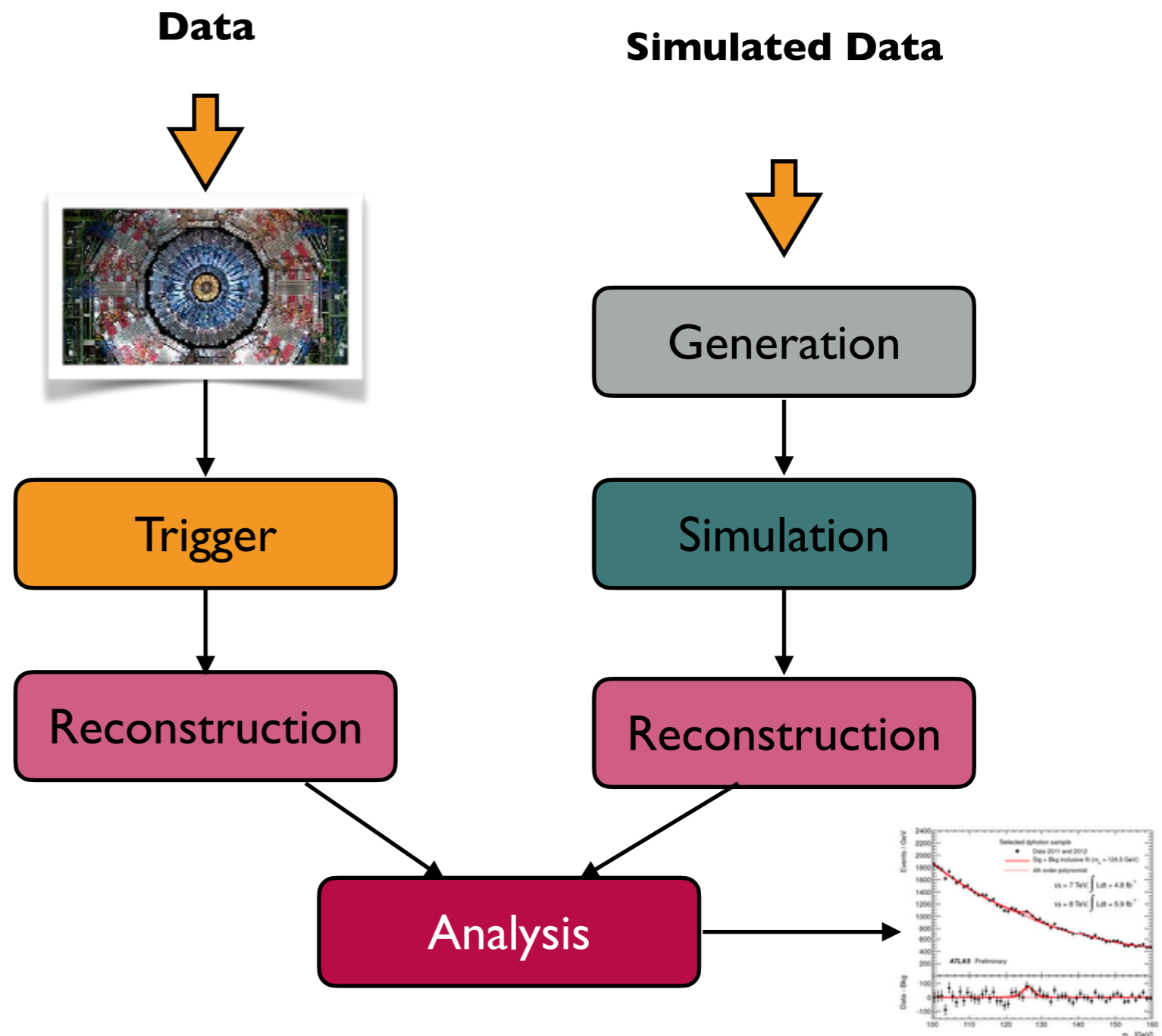
Reconstruction

- Reconstruction algorithms use the **raw input** from detectors to reconstruct the **particles** that passed through the detector
- Two main categories:
 - **2D energy**, e.g. calorimeter
 - **3D trajectories**, e.g. charged hadrons, muons
- Some particles, e.g. electrons rely on a combination of the two



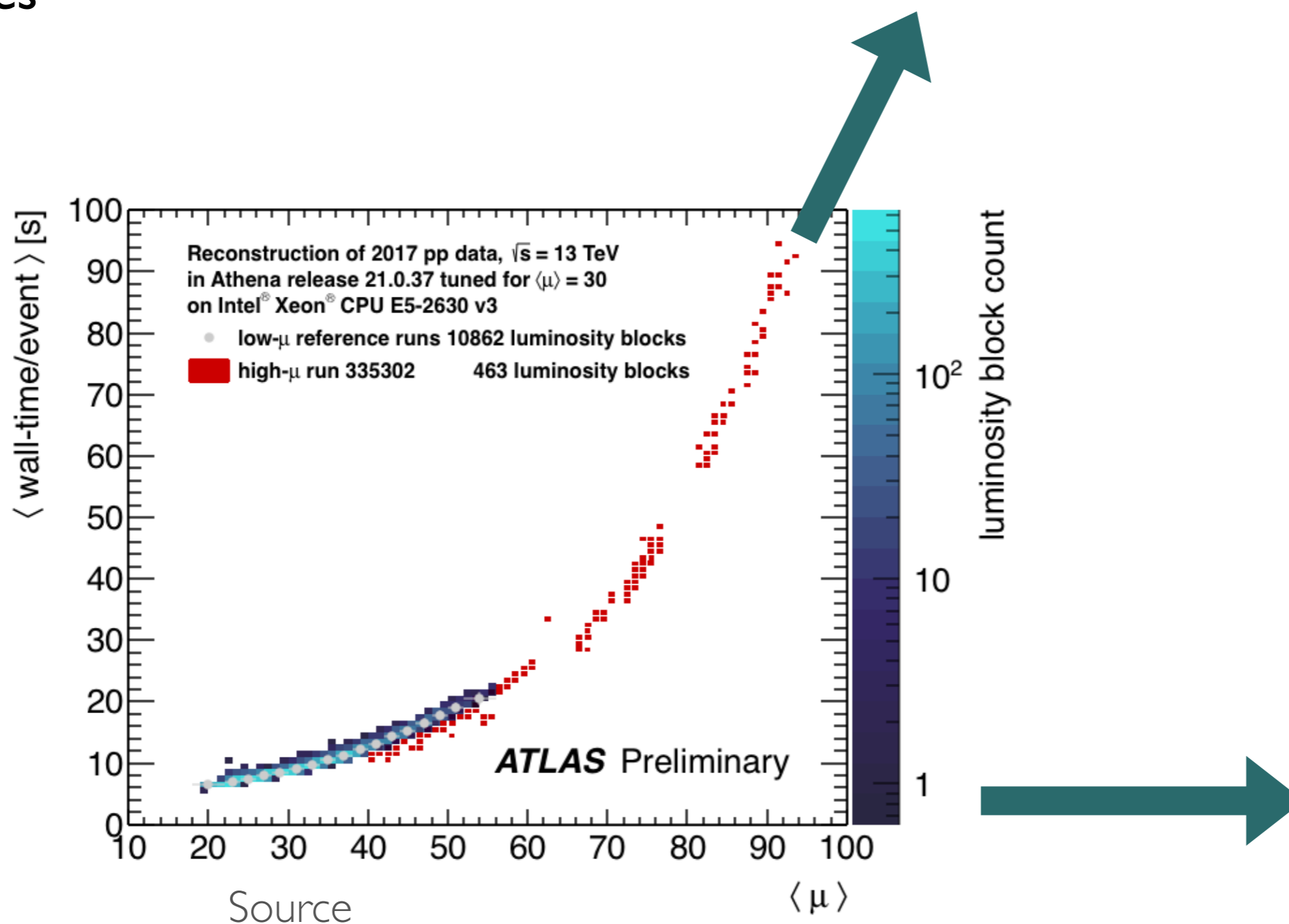
Online and Offline

- Reconstruction algorithms are run in different configurations
 - **Simulation**
 - **Data**
 - **Online**
 - **Offline**
- Algorithms are not necessarily the same
- Accelerators can be used (or not) in each of these contexts



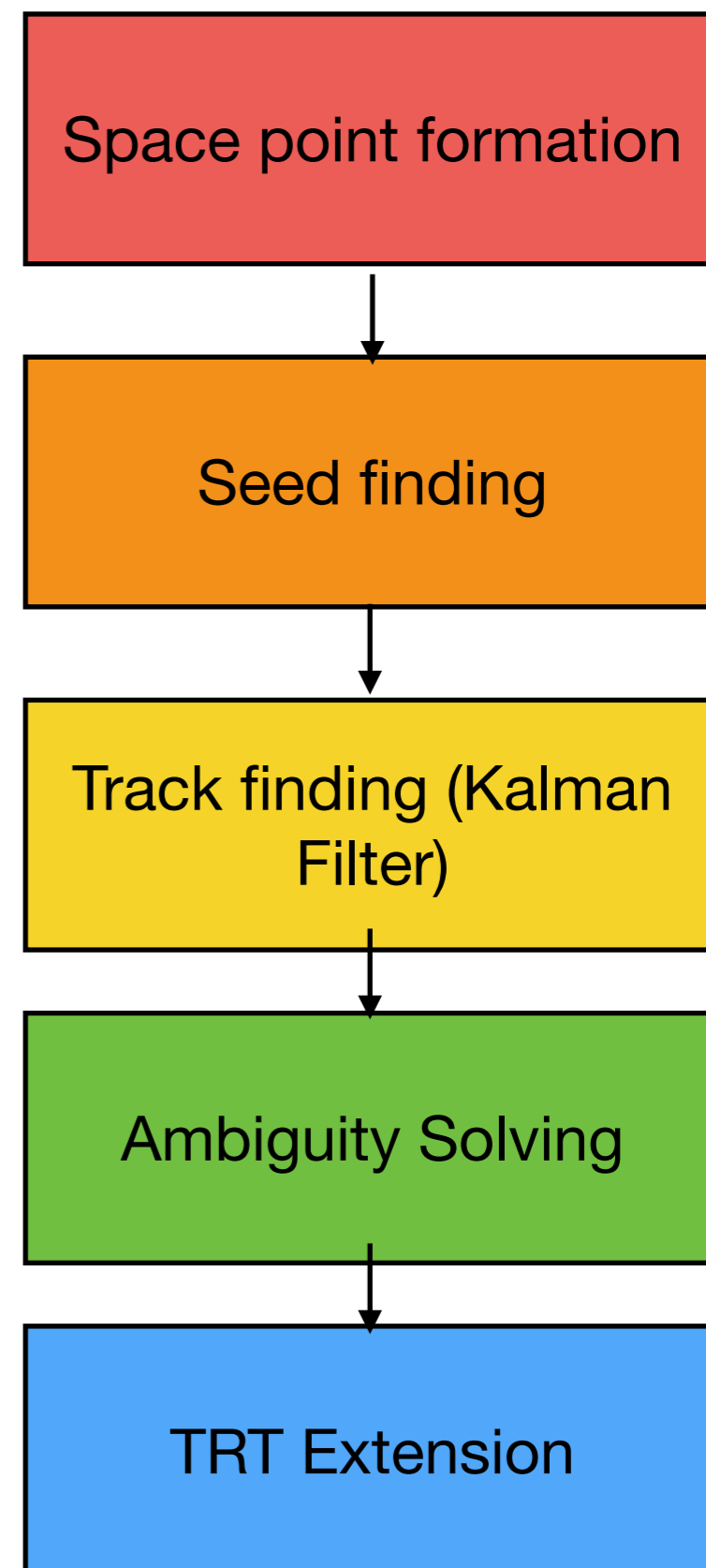
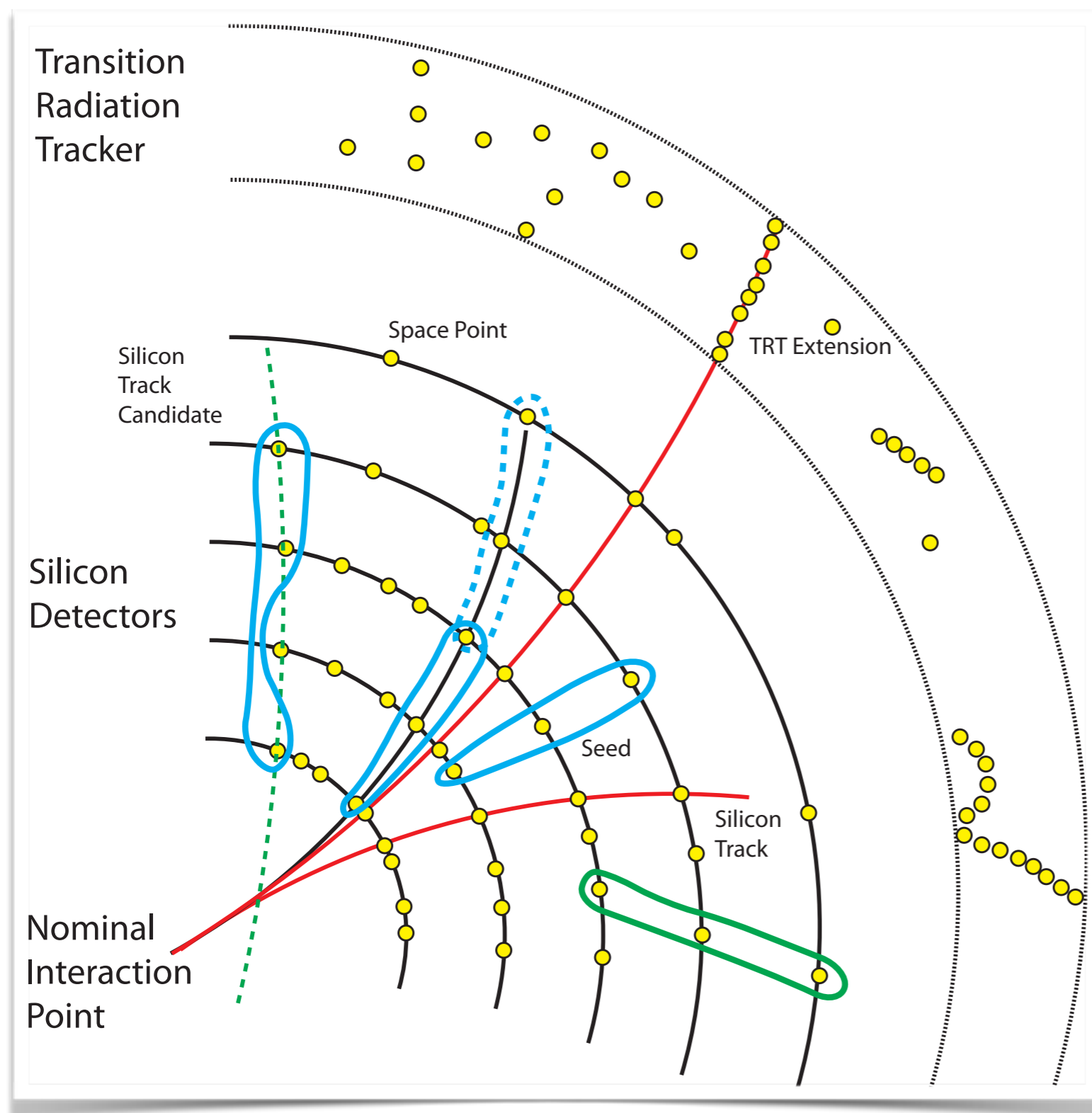
Accelerating Reconstruction

- **Reconstruction** algorithms tend to one of the favored target for acceleration
- In particular, **track reconstruction** algorithms are computationally expensive and **scale poorly** (often quadratically) with the number of particles



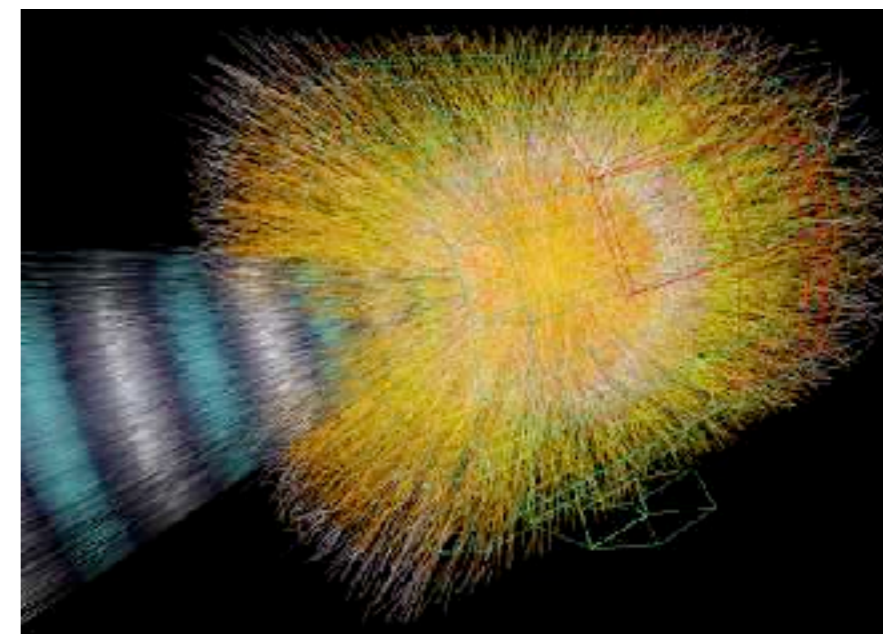
Track Reconstruction

ATLAS-CONF-2010-072

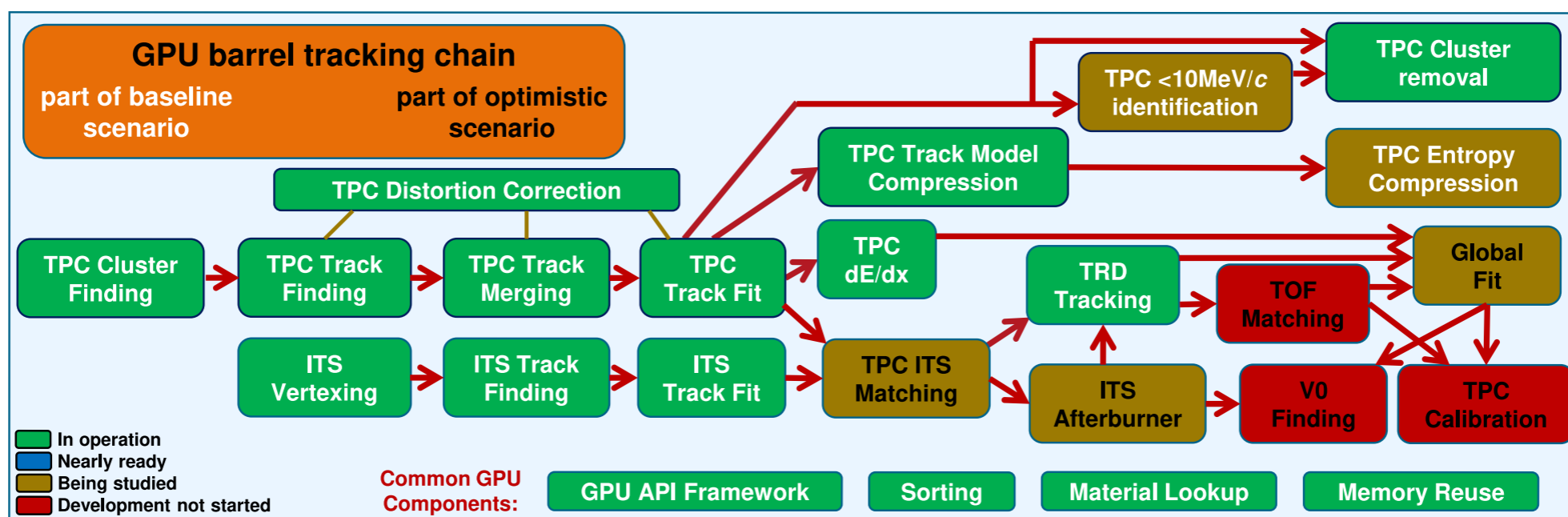


ALICE: Track reconstruction on GPU

- **ALICE** is an early adopter of GPUs amongst the LHC experiments
- Their tracking detector, the **Time Projection Chamber (TPC)**, dominates the computing needs
- Process 10 ms time frames, each $O(10 \text{ GB})$ in size
- GPUs also used for **compression** and **calibration** of TPC data since Run I
- ALICE is currently adding reconstruction of other detectors on GPUs

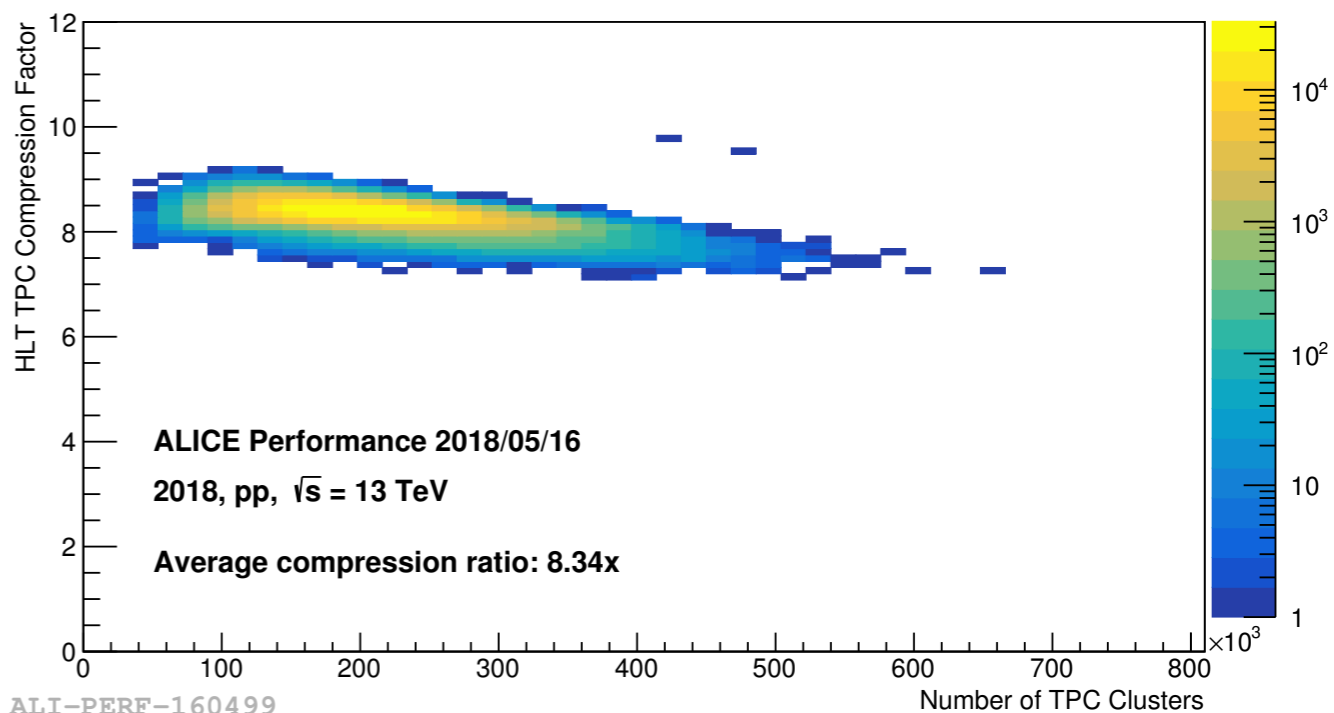
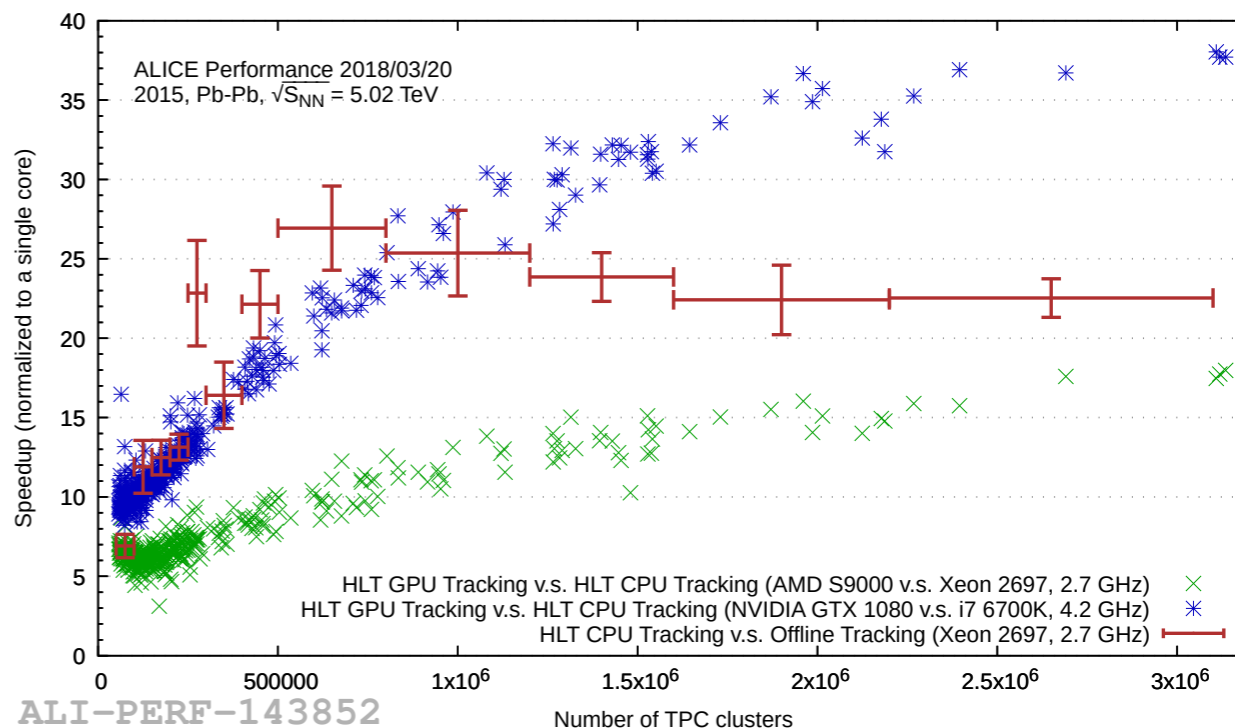


Source



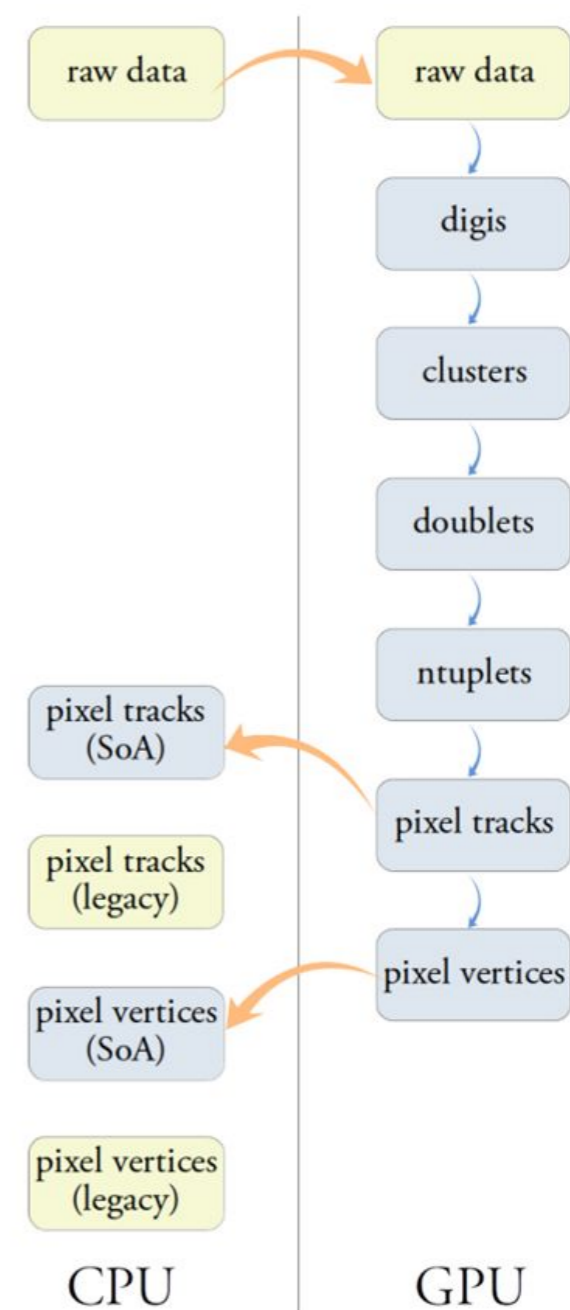
ALICE: GPU Tracking

- For Run 3, the ALICE data-taking read out has been increased to a **50 kHz continuous read out** of PbPb collisions
- Cellular Automaton (CA) algorithm
- Exclude multiple track hypotheses
- 800x total speed up (wrt offline)
- **Comparable efficiency** but **degraded resolution** wrt to offline
- 8x data compression



CMS: Patatrack

- The heterogeneous computing on CMS tracking is mainly led by [Patatrack](#) project (born in 2016)
 - Track reconstruction in Pixel detector
 - Developed for event filter
- Full tracking chain on GPU
 - No memory transfer between CPU and GPU
- Adopted Struct-of-Array (SoA) for fast memory access to Event Data Model (EDM)
- CUDA is main API, but development with HIP/Kokkos/Alpaka/openMP is on-going as well

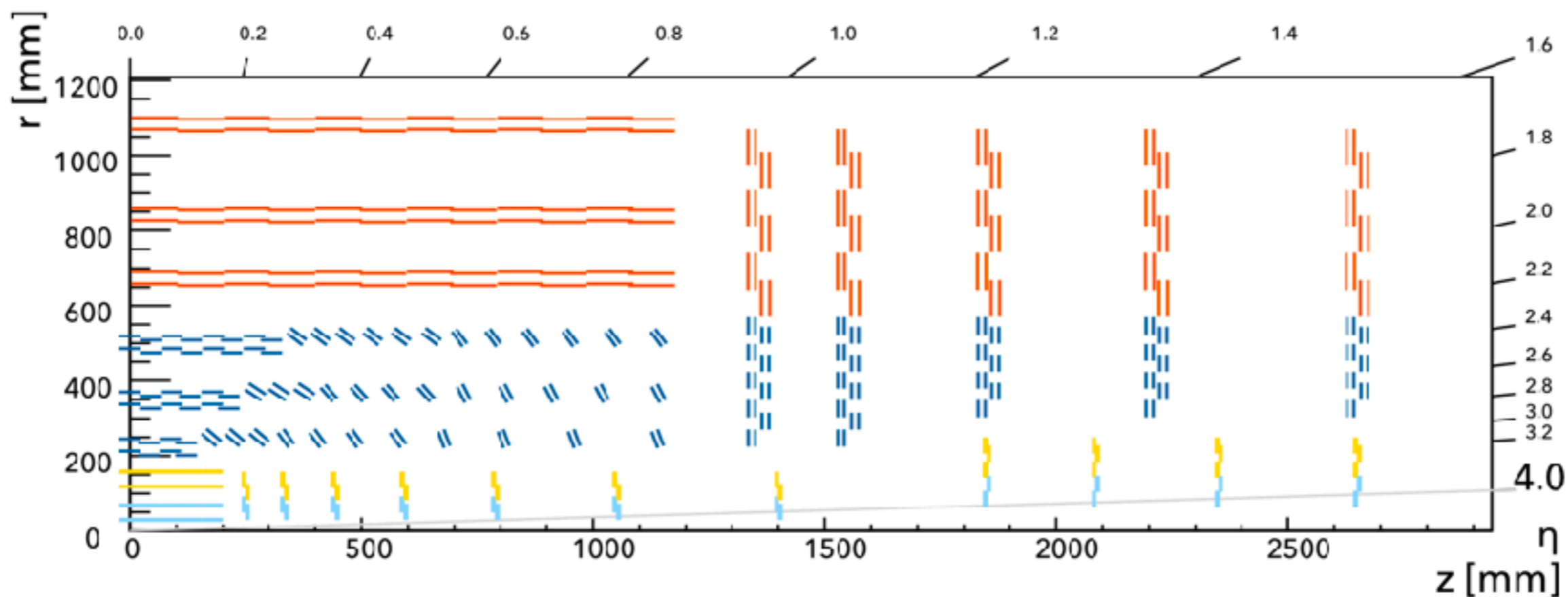
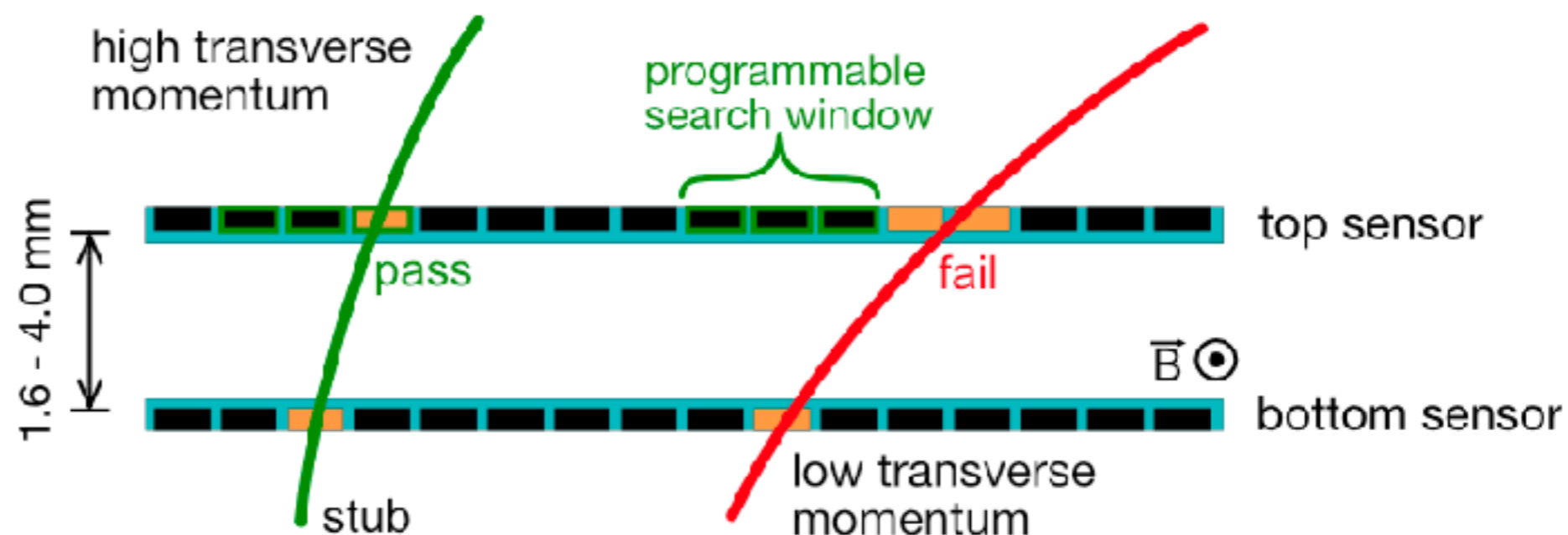


Broken Line for Track Fit

- There is no iteration on spacepoints as KF does
- Instead, **two matrix equations** are solved to minimize the least square estimator which is a function of residuals and kink angle on detector plane
 - The motion of helix is decoupled into transverse (x-y) and longitudinal (r-z) plane
- Pros and Cons with respect to KF
 - Pros: Generally faster than KF with similar resolution
 - Cons: The extension to external hits is inefficient because the matrix equation should be solved from the beginning
- In Patatrack, each thread runs Broken Line Fit for each N-tuplet
 - Elgen3 library that natively supports CUDA is used to solve matrix equation

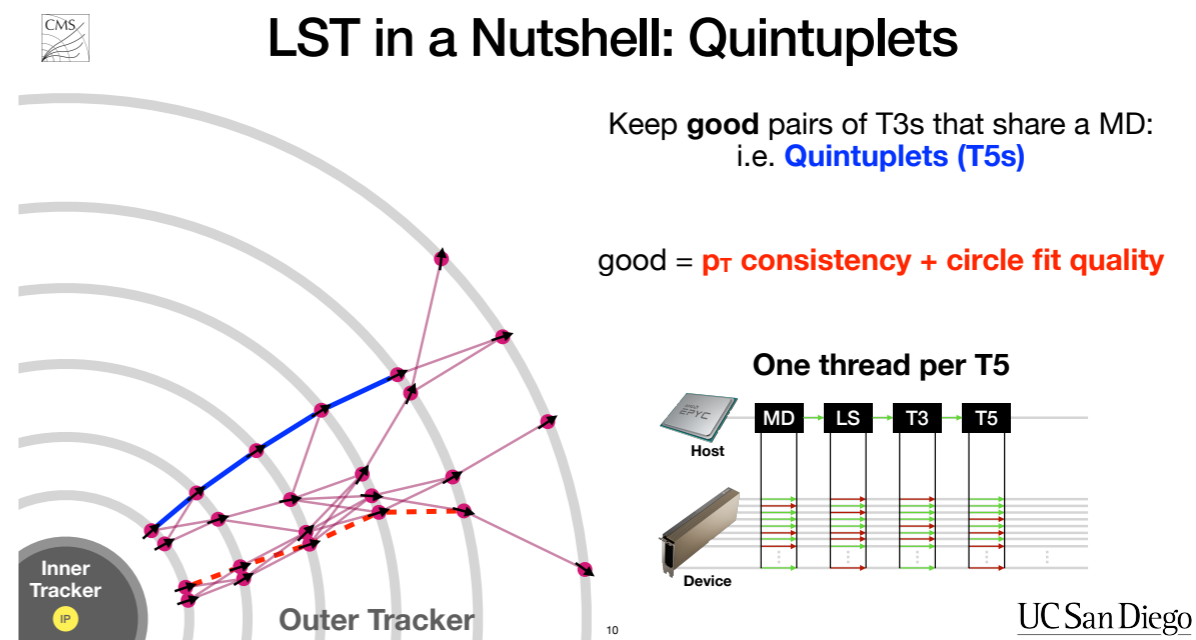
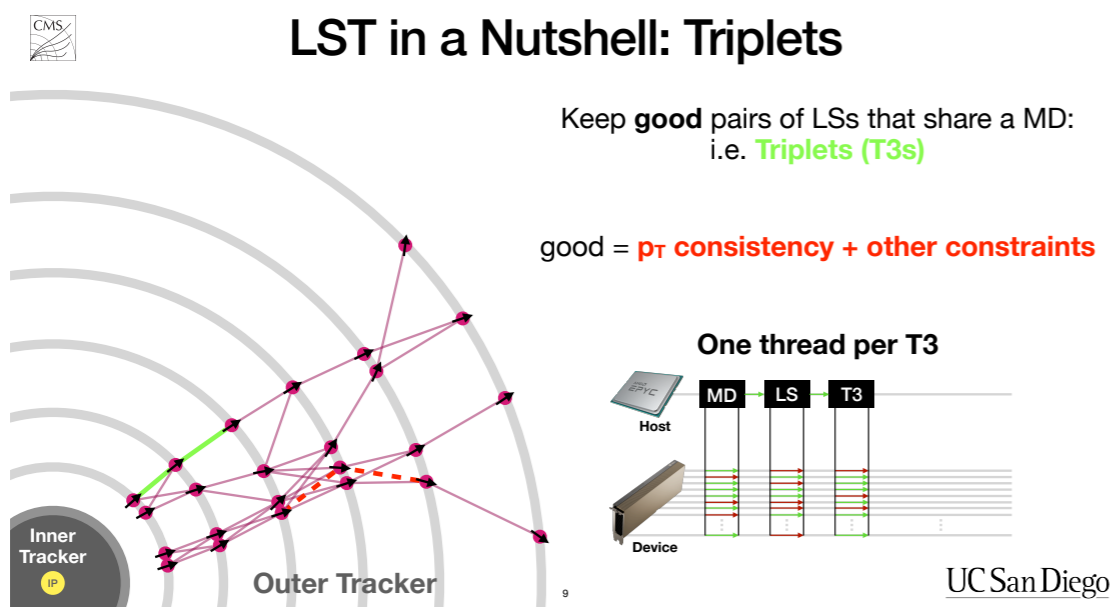
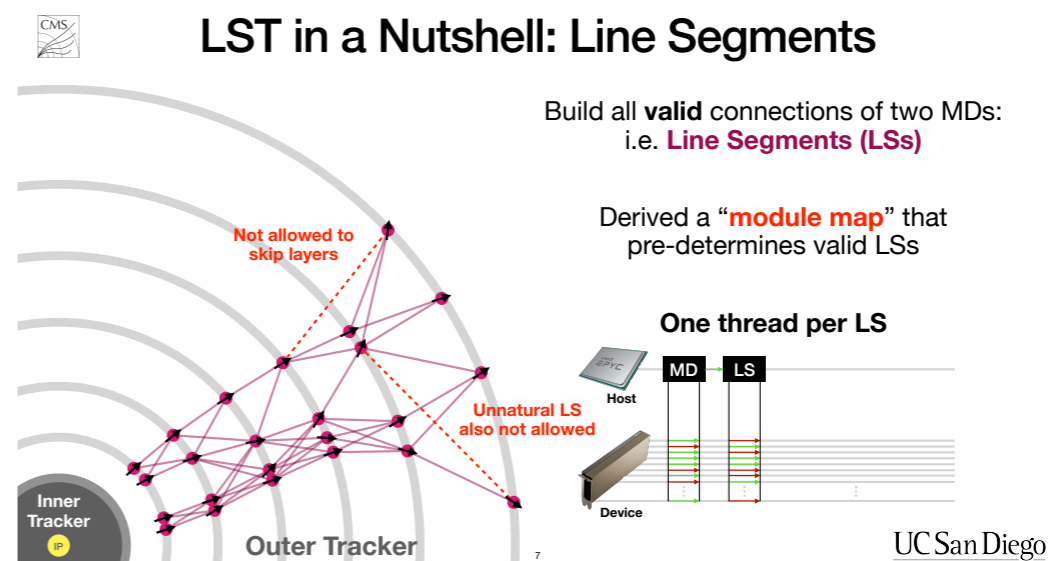
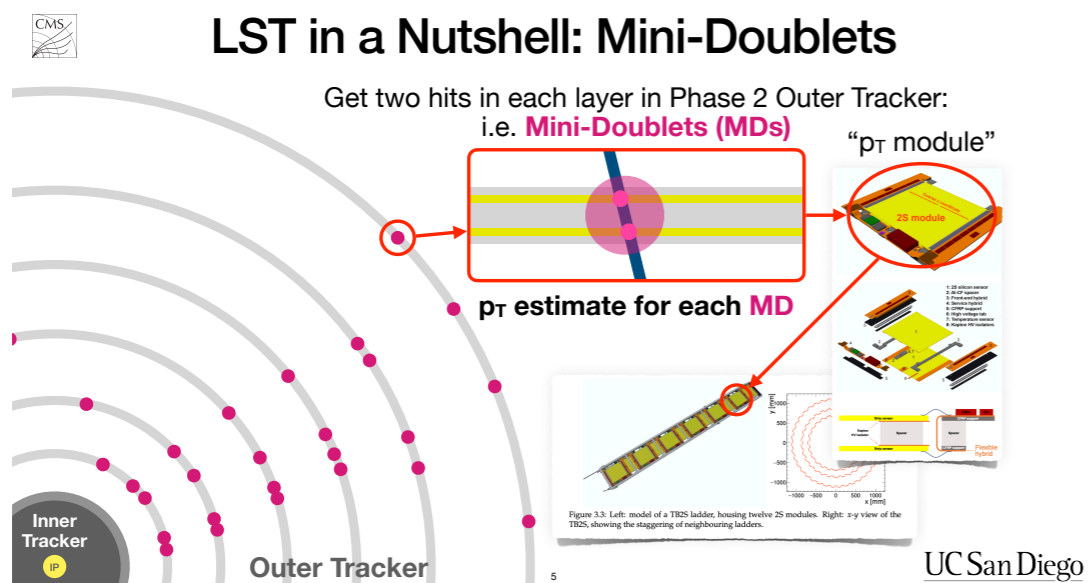
CMS Tracker for HL-LHC

Novel CMS Tracker Design for HL-LHC features pT modules enable LI high PT track trigger



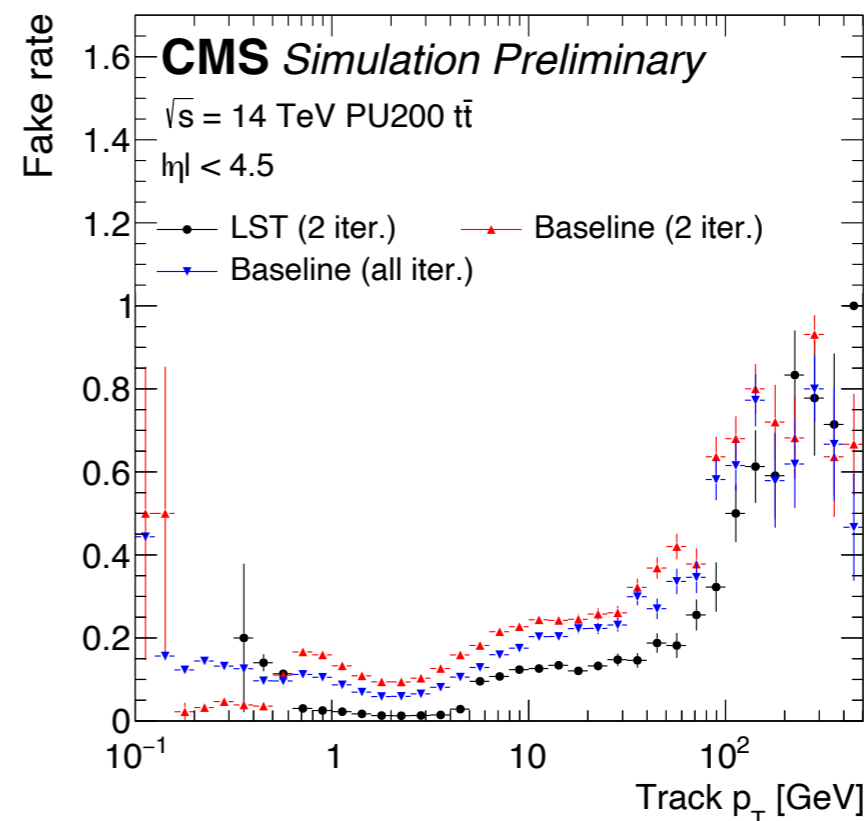
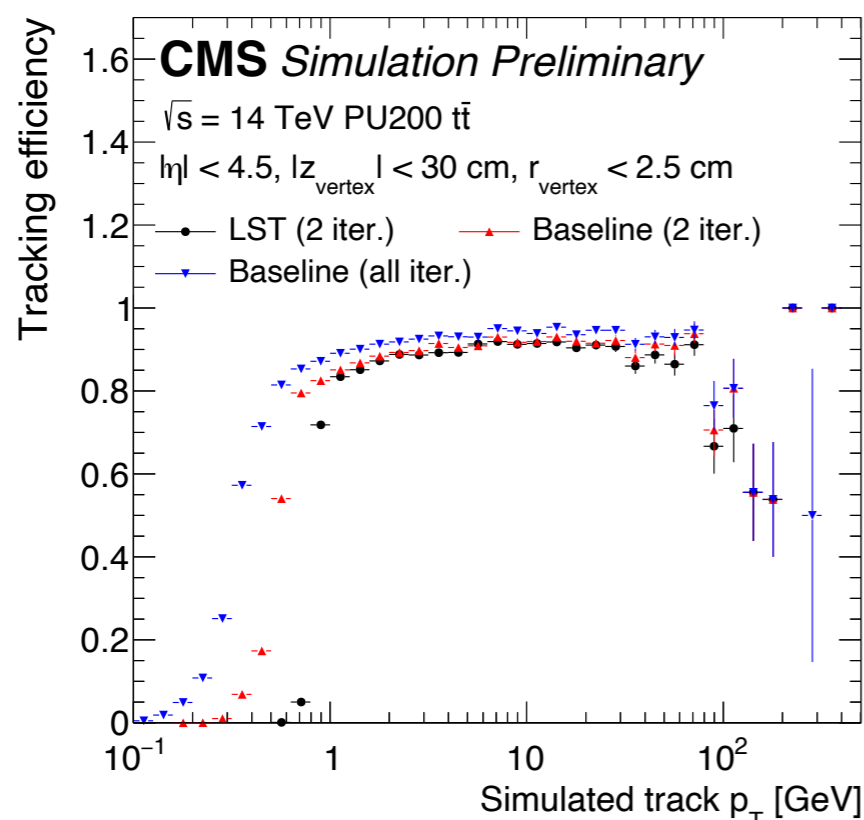
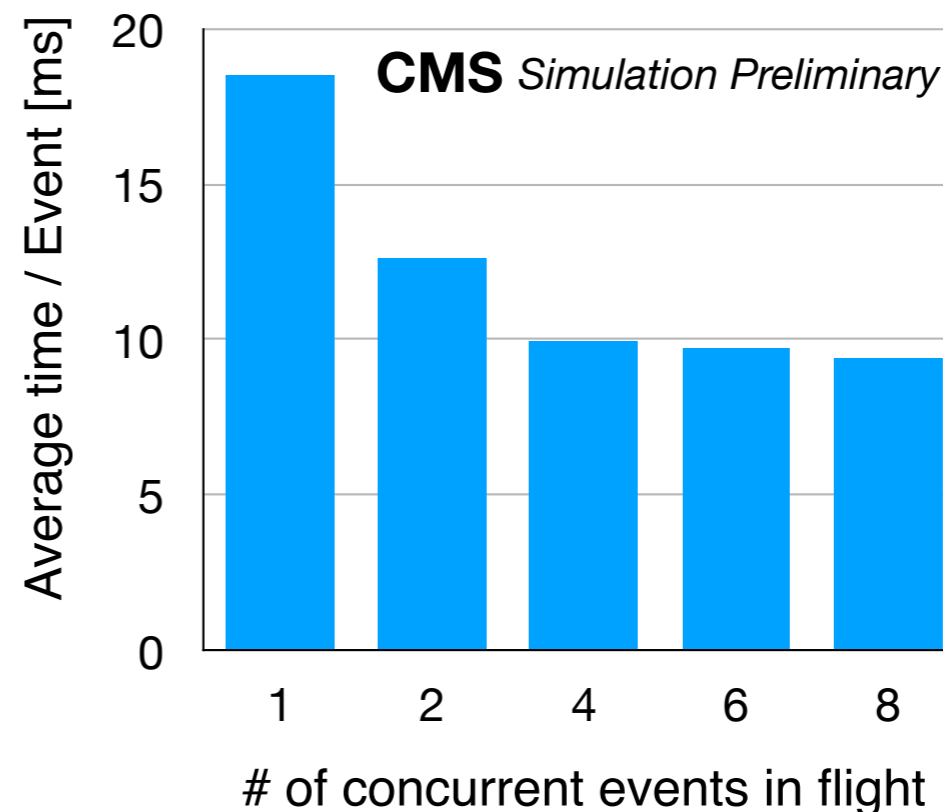
CMS: Line Segment Tracking (LST)

- CMS has designed a new algorithm specifically designed to run on GPUs and exploit this tracker design



LSST Performance

- Algorithm is still under development and optimization
- **Similar efficiency** and **lower fake rate** to current CMS algorithm (except at very low p_T)
- Approximately 10 ms per event

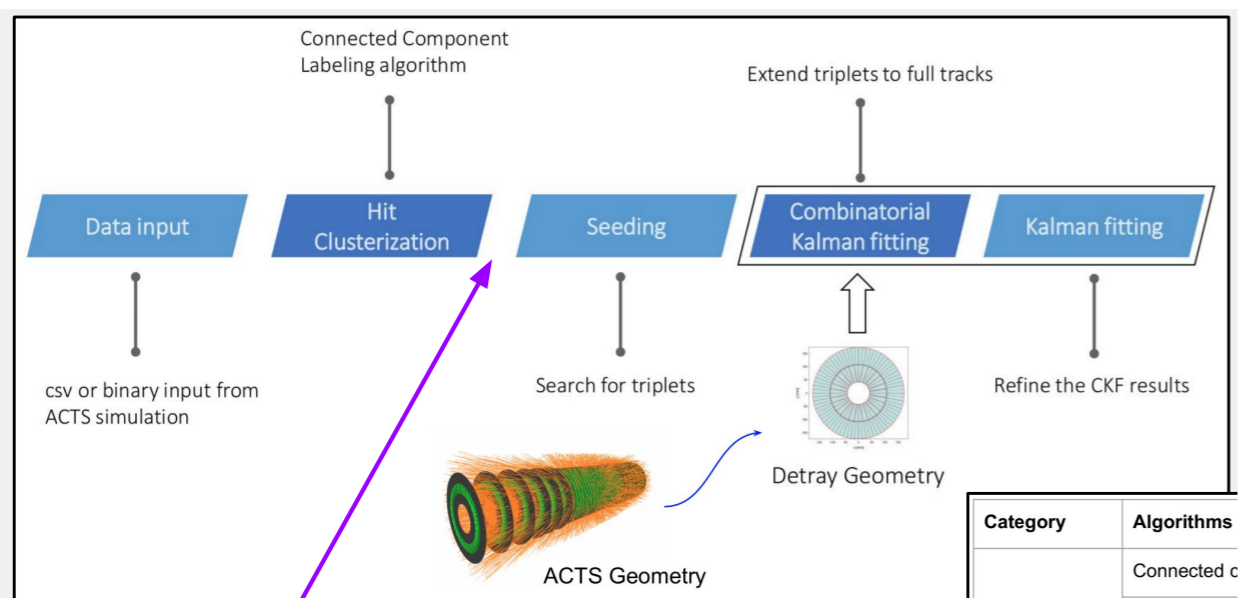


Source

Excludes transfer time between host and device

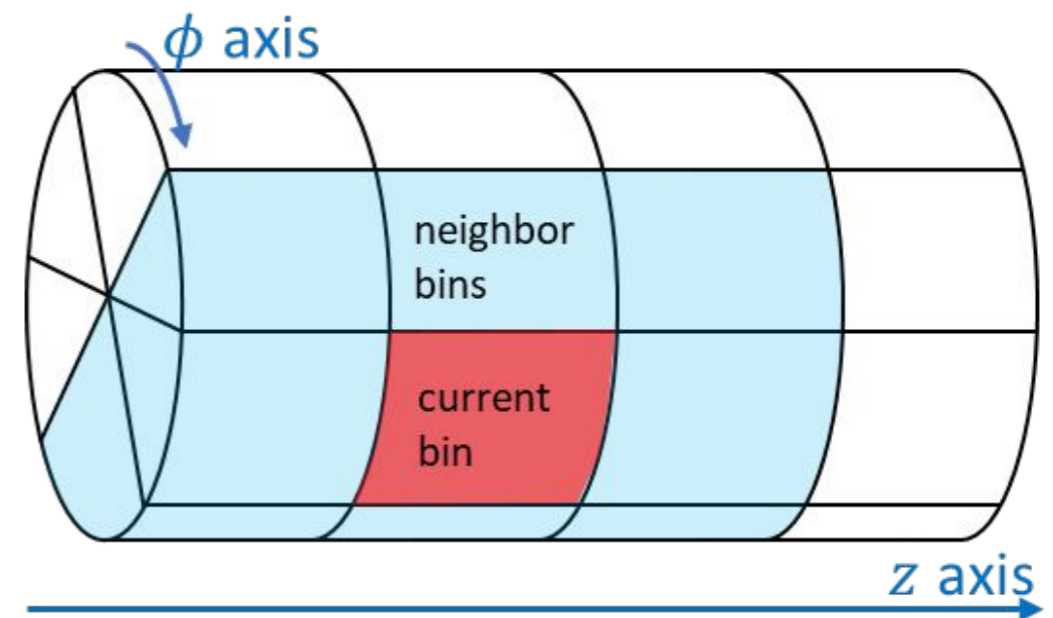
ATLAS: traccc

- The traccc project aims to develop an end-to-end or full chain track reconstruction chain on GPU
 - Avoids time lost in copying between device and host memory
 - CUDA and SYCL implementations (+ CPU)
- Separate package for detector geometry (detray) and memory management (vecmem)
- Code has been designed in an experiment-independent way

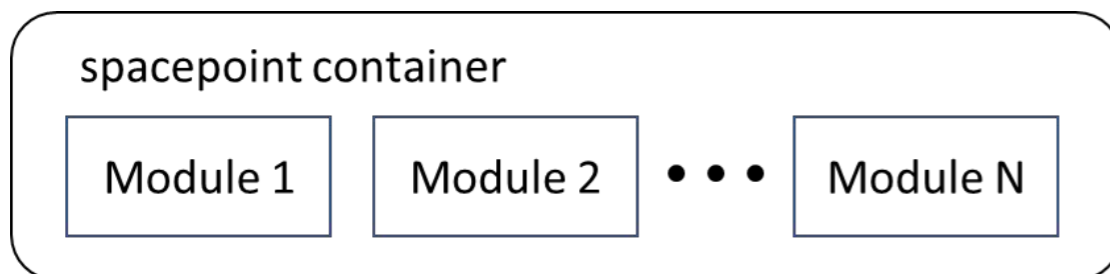


Spacepoint binning

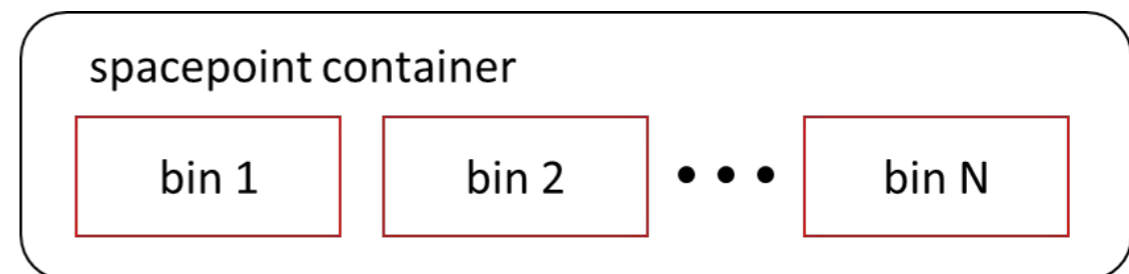
- Seeding is done bin-by-bin by grouping spacepoints w.r.t z and azimuthal angle
- There are two cuda kernels:
 - Count the number of spacepoints per bin to allocate the memory to the container
 - Fill out bins with spacepoints
- Each thread for each spacepoint



➤ Before binning

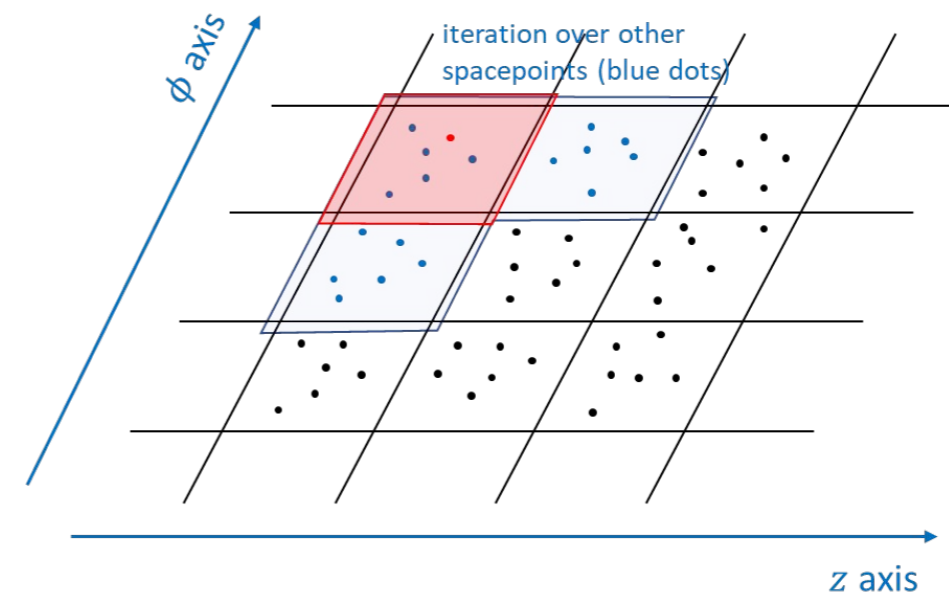
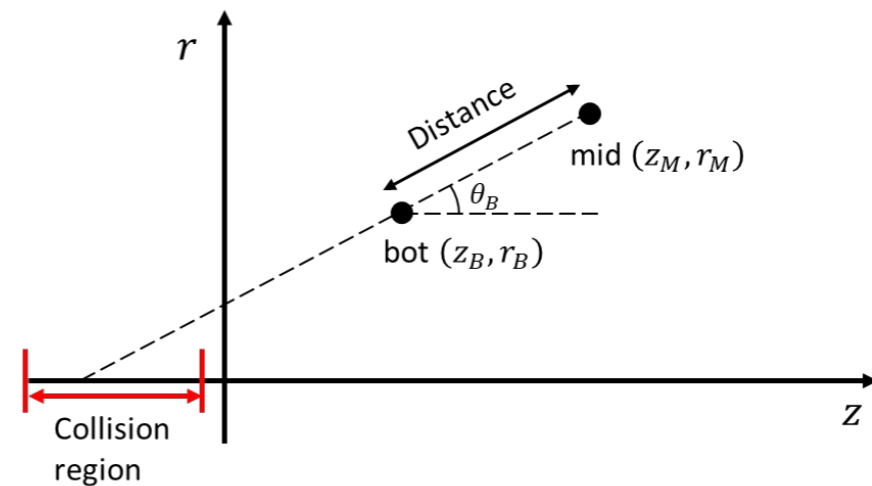


➤ After binning



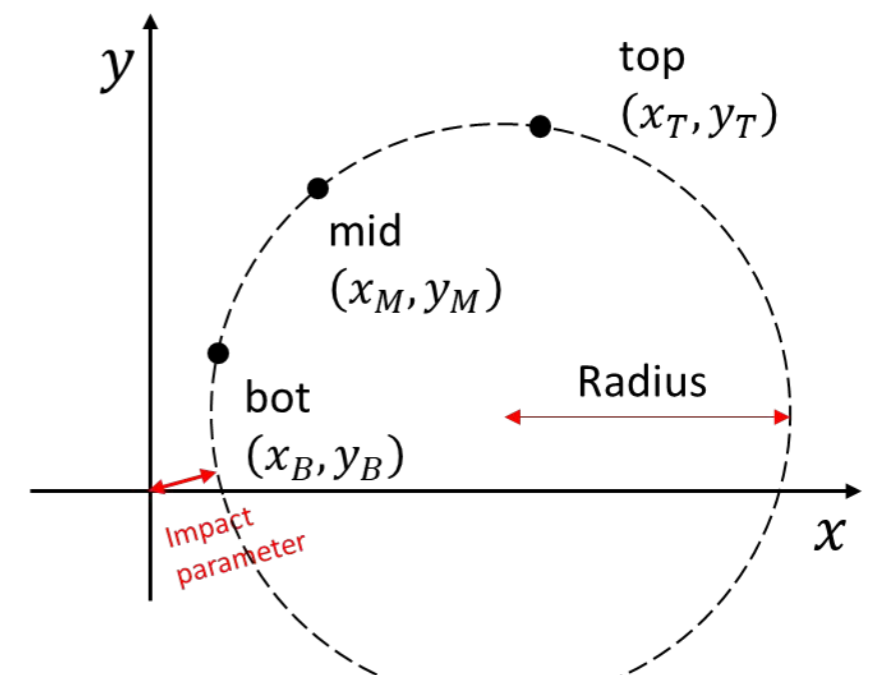
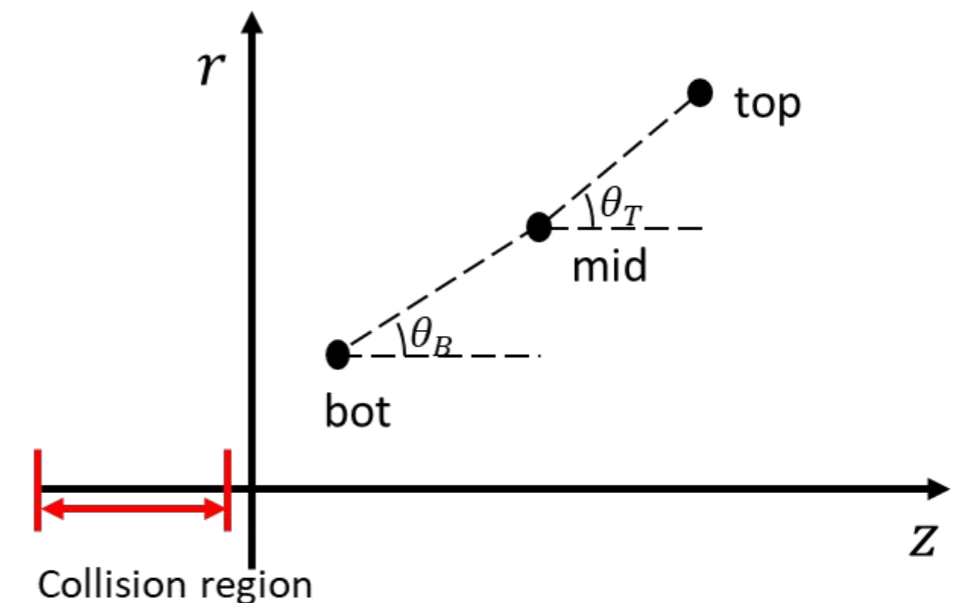
Doublet Finding

- Doublets are found by collecting hit pairs that satisfy certain criteria:
 - Distance to the beam spot
 - Enough large pitch angle
- Like spacepoint binning, there are two kernels:
 - Doublet counting for memory allocation
 - Doublet finding
- For all spacepoints, iterates over other spacepoints in *current* and *neighbor* bins to find doublets
 - Form bottom-middle or middle-top doublet
- Each thread is assigned for each spacepoint



Triplet Finding

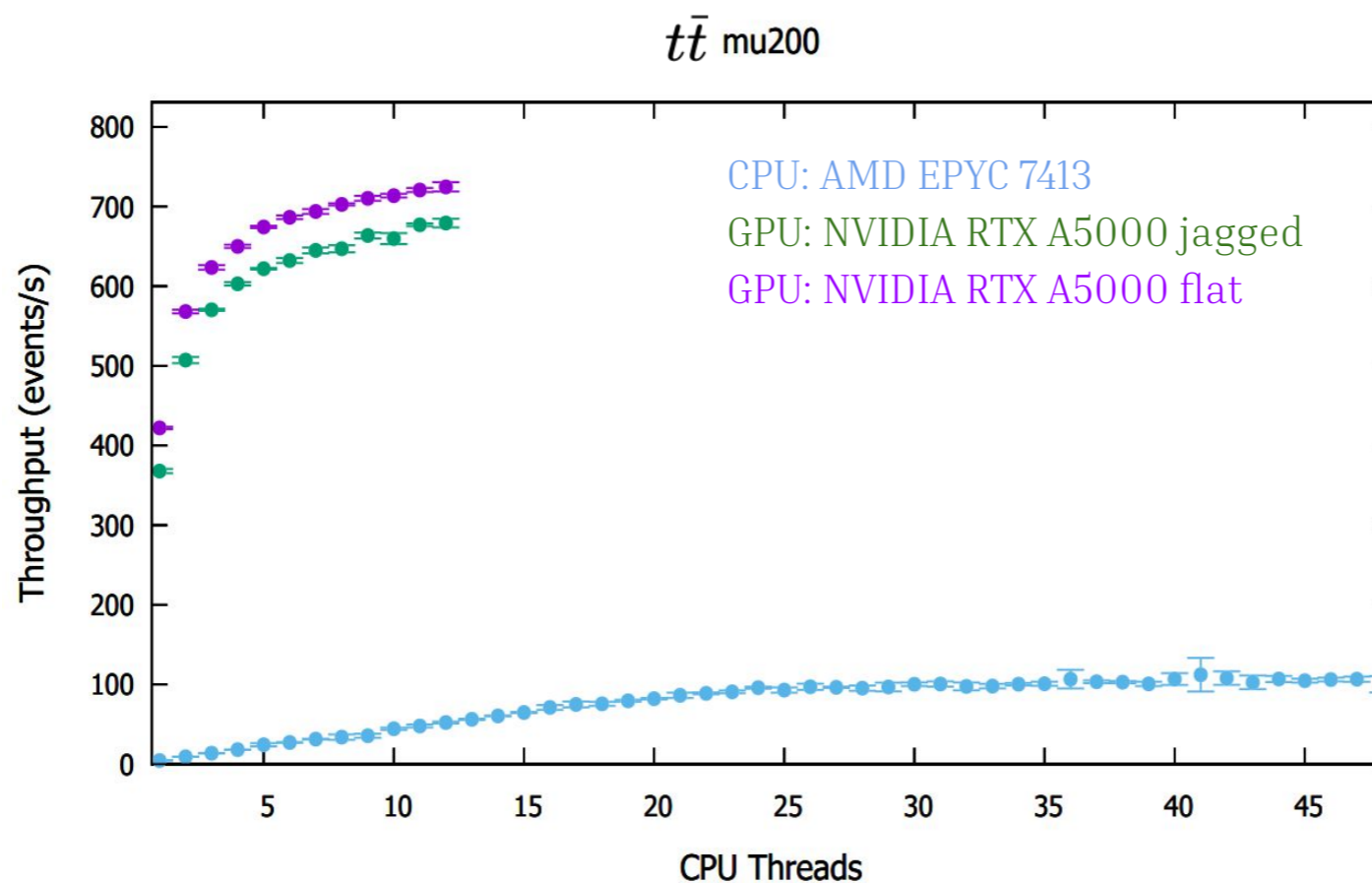
- Triplets are found by combining doublets that share the same spacepoint and satisfy certain criteria:
 - Min pT
 - Distance to the beam spot
 - $\theta_B - \theta_T < \text{tolerance}$
- Like spacepoint binning, there are two kernels:
 - Triplet counting for memory allocation
 - Triplet finding
- For all middle-bottom doublet, iterates over middle-top doublets that share the same middle spacepoint
- Each thread is assigned for each middle-bottom doublet



Seeding Performance

ratio between ACTS CPU and tracc CPU is around 98% due to different sorting criteria but it's trivial

- The seed matching ratio between tracc CPU and tracc CUDA is 99 – 100 %
- For $t\bar{t}$ events, CUDA speedup is about 13 compared to single core CPU



traccc vs Patatrack

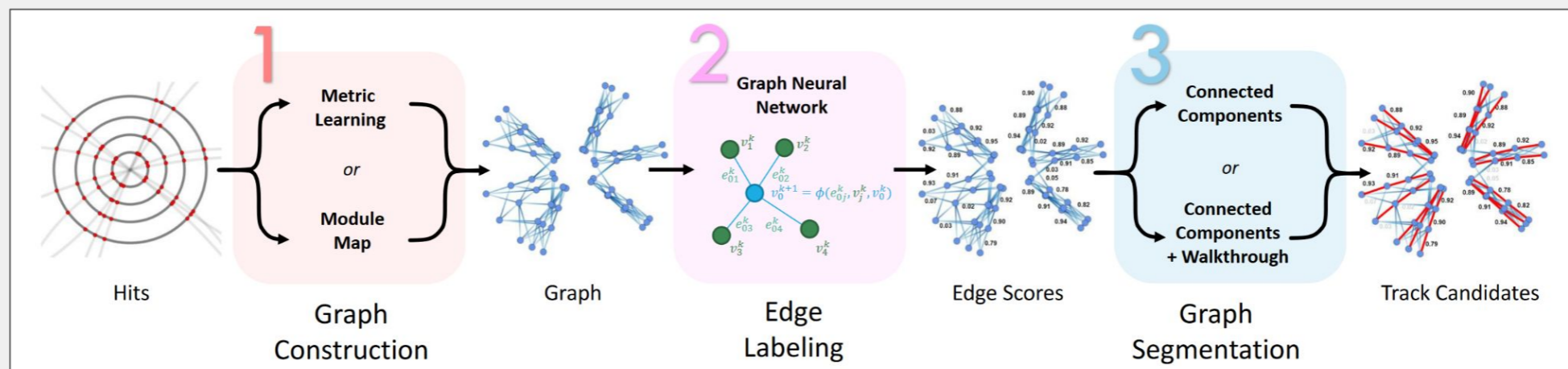
- The general flow of tracking chain is similar with ATLAS's one
 - hit clusterization / track finding / track fitting
- However, the detailed algorithms are different:
 - In ACTS, the triplets from seeding are extended to n-tuplets with Combinatorial Kalman filtering
 - In Patatrack, the n-tuplets are found in seeding and they are fitted directly with Broken Line Fit

	ACTS	Patatrack
Track Finding	doublet finding	doublet finding
	triplet finding	N-tuplet ($N \leq 4$) finding
Track Fitting	(Combinatorial) KF	Broken Line Fit

ML Tracking on GPUs

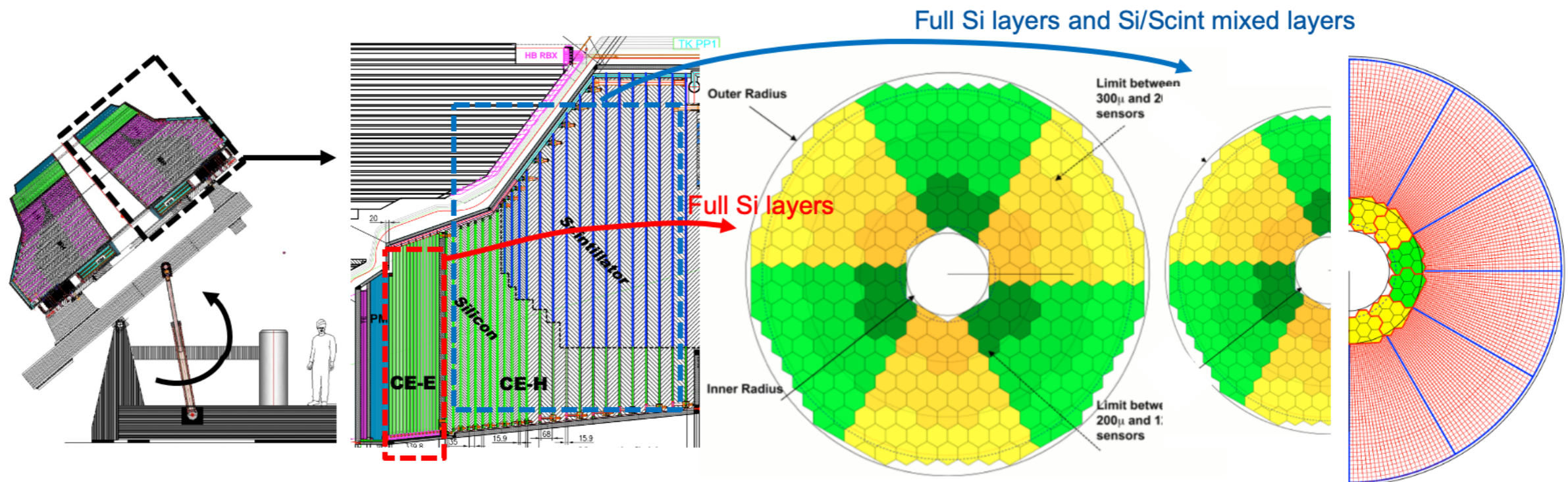
[Exa.TrkX](#) + ITk

- GNN based tracking for ITk offering [competitive](#) track efficiency and high quality track parameter resolution
- Available as a [Service](#) - no need to have a local GPU



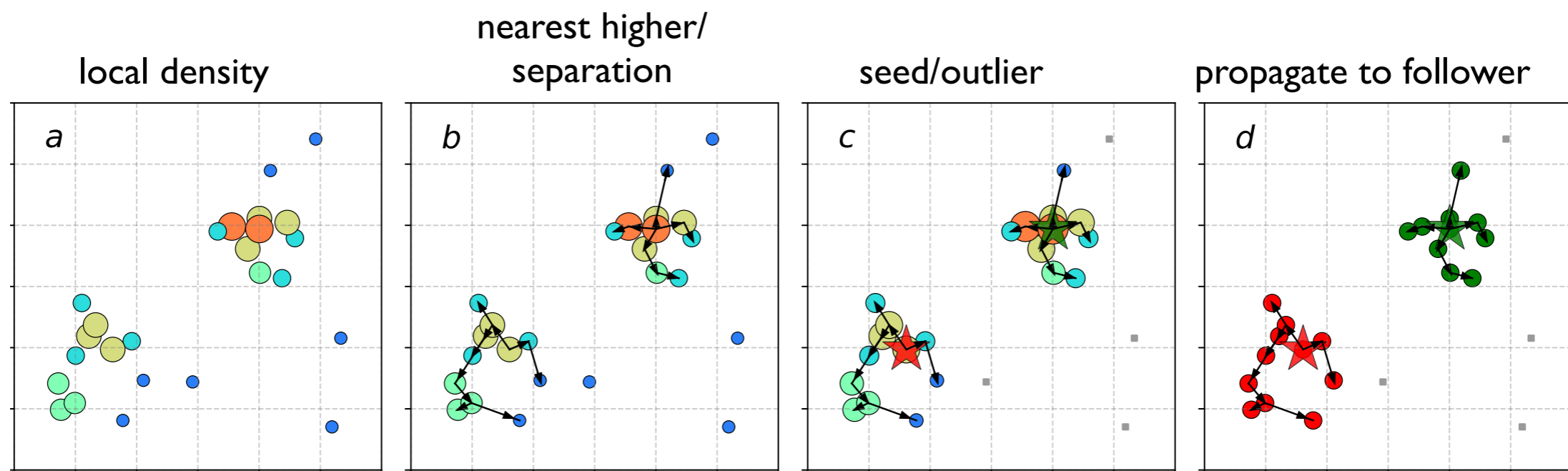
Clustering for the CMS HGCalorimeter

- The CMS High-Granularity Calorimeter (HGCal) is a replacement for the calorimeter end-cap based on highly-segmented silicon sensors and plastic scintillators
 - ~6 million silicon channels and ~240k scintillator channels
- Large number of channels and the high pile up expected
 - Clusterization will be a significant computational challenge



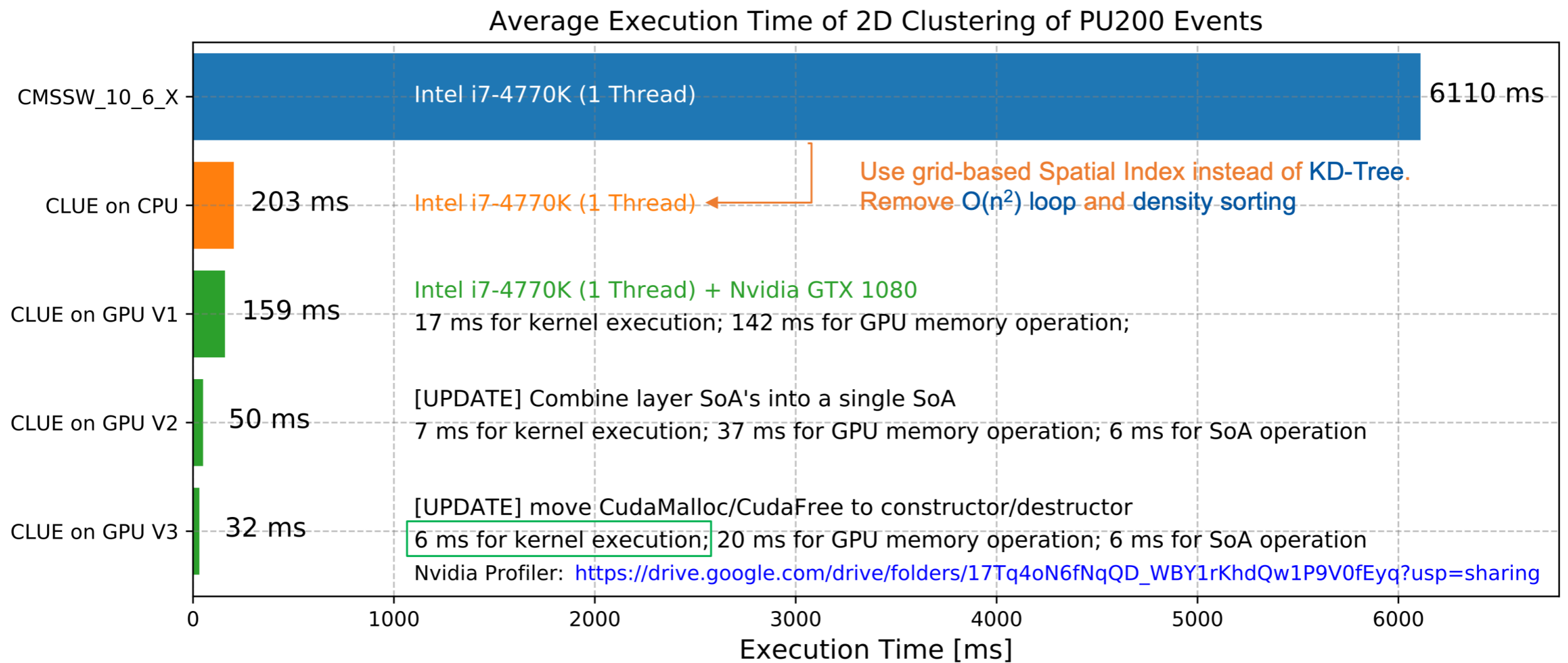
Clustering for the CMS Calorimeter

- A new parallelized algorithm, **CLUE**, has been developed
- Based on Clustering by Fast Search and Find Density Peak (**CFSFDP**)
- Calculate local density, ρ , and separation, δ : distance to nearest point with highest density (nearest higher)
- Identify **cluster seeds** based on ρ and δ
- Build **follower list** by registering each point as a follower to nearest higher
- Expand cluster by passing **cluster indices** from seeds to followers iteratively
- Four parameters
- Parallelization: Each cluster is expanded independently



Clustering for the CMS Calorimeter

- CLUE is **30x faster** than the previous algorithm on a single-threaded CPU
- GPU implementation obtains an additional speed up of **6x**



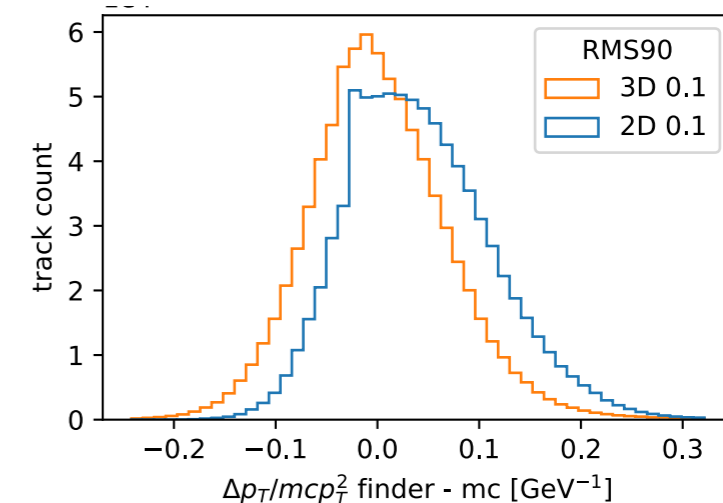
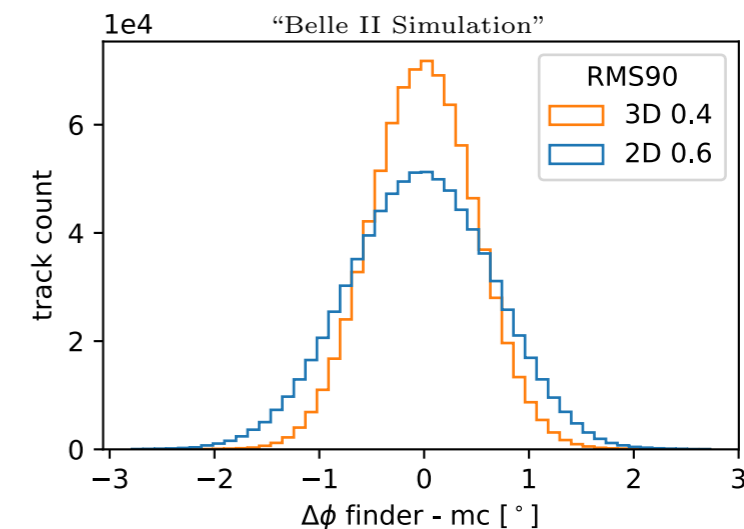
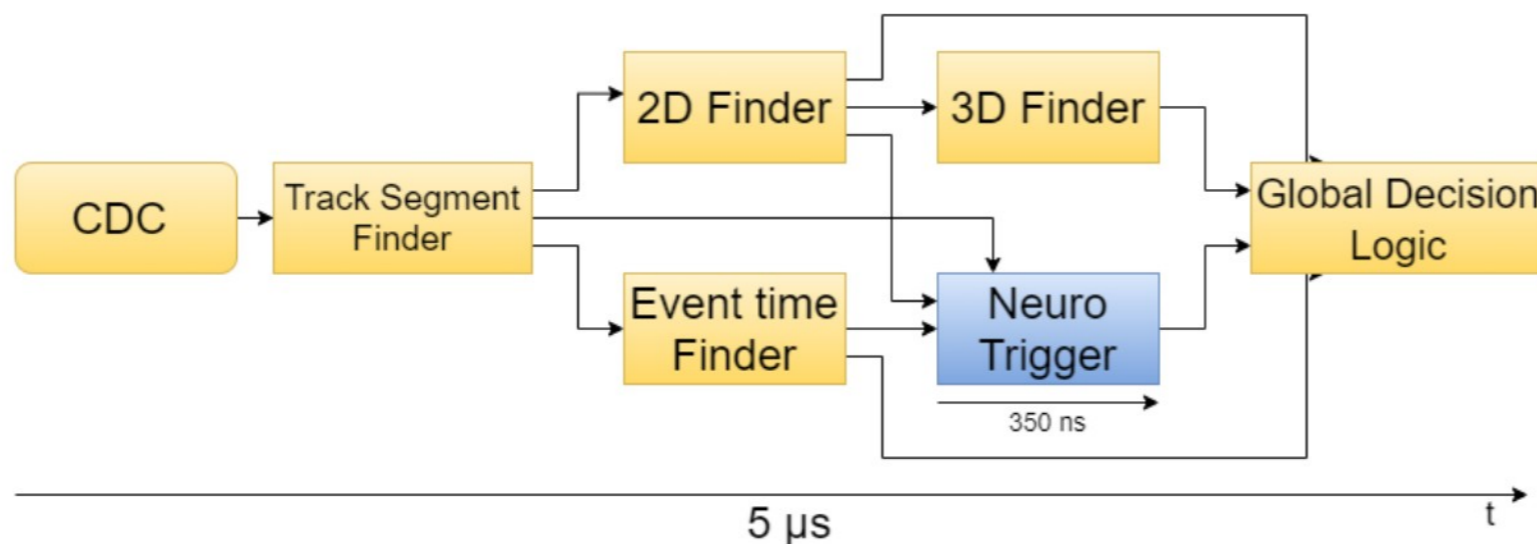
Trigger

Trigger Evolution

- Triggers have extremely **low latency** requirements
 - Track reconstruction can be a challenge
- HEP has a long history of tracking in the trigger (e.g. LEP, Tevatron) using **hardware track triggers** historically primarily relying on **FPGAs**
 - FPGAs meet the low latency requirements
- Algorithms are evolving in two primary directions
 - **More computation** and **more complex** algorithms (close to offline physics performance) for the hardware trigger
 - **Triggerless read-out**: no hardware trigger and the software trigger processes all events
- Of course, the approaches can be mixed, e.g. when using hardware accelerators in the software trigger

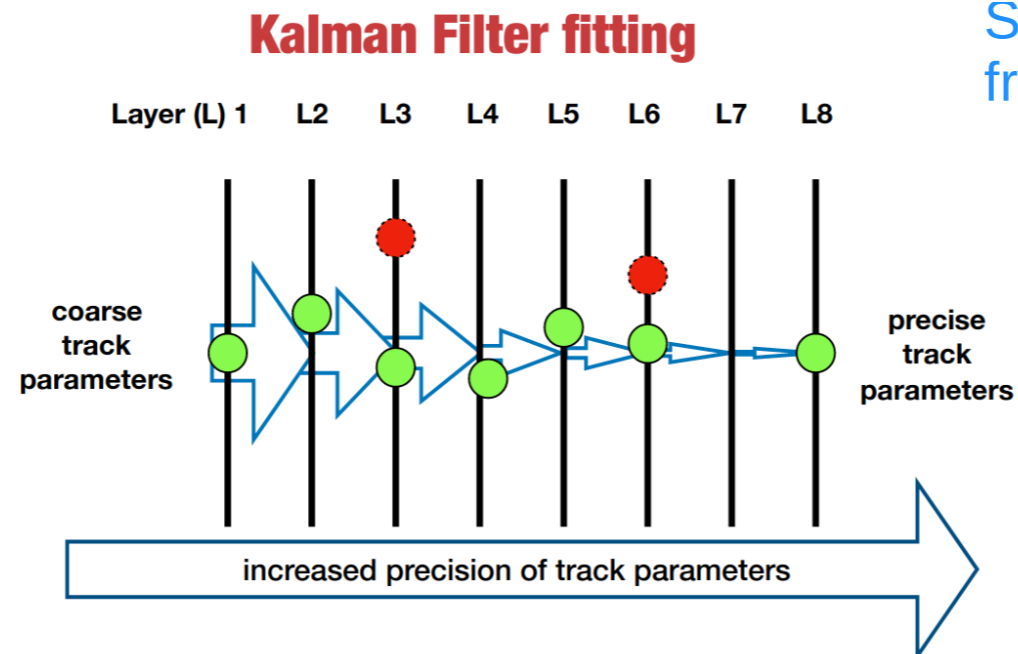
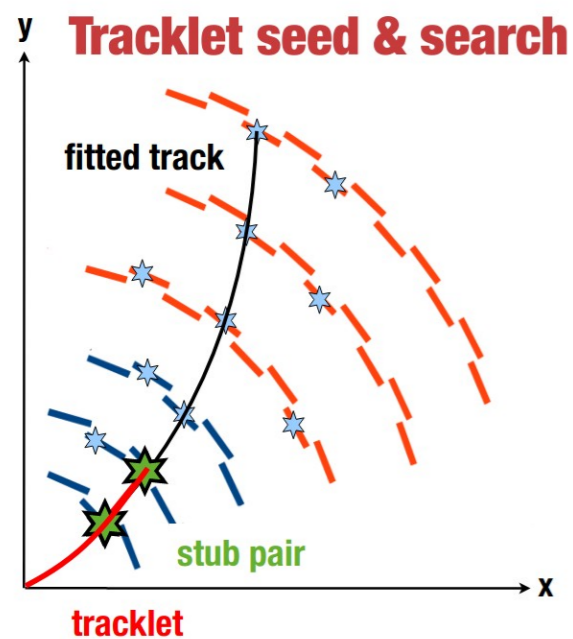
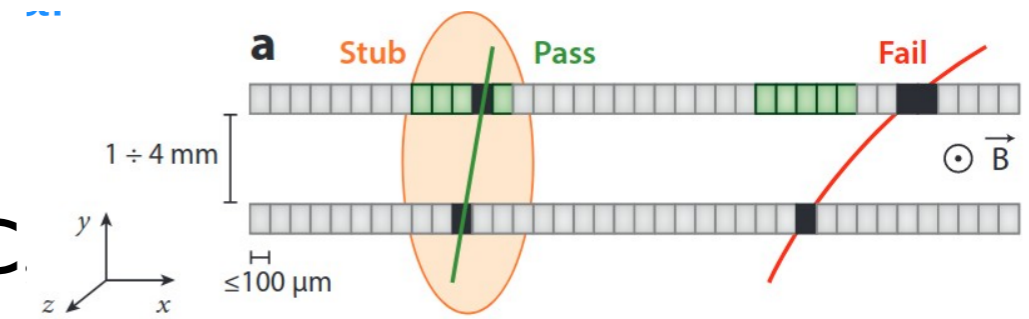
Belle-II: 3D hardware tracking at LI

- Belle-II, the b-physics experiment at the SuperKEKB accelerator in Tsukuba, Japan has full **3D hardware tracking** in the LI trigger
- Reject beam background using the Central Drift Chamber selecting on z
- 1.7 Tbps rate out of CDC and average of 11 tracks/event
- FPGA system with a 5 μs latency
- 2D/3D Hough transforms and a neural network in firmware

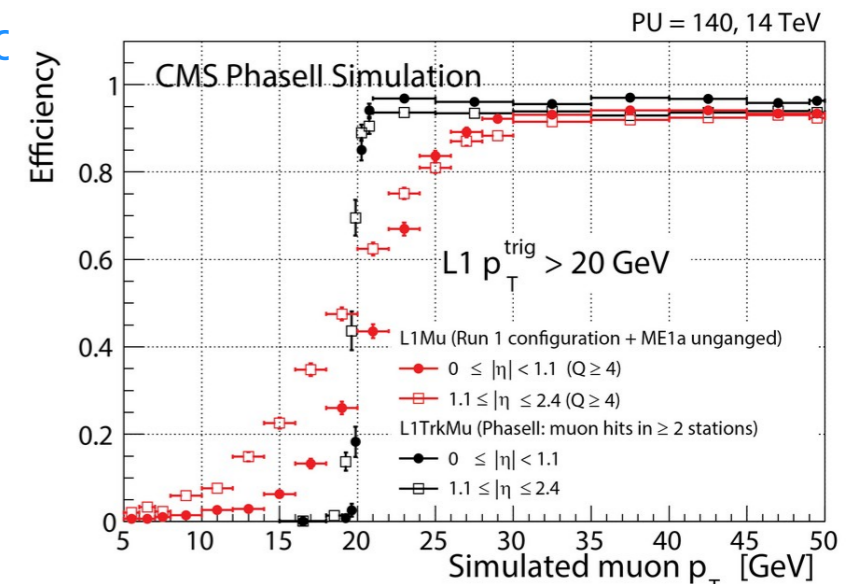


CMS: 3D hardware tracking at L1

- CMS upgrade will have hardware tracking at L1
- Algorithm relies on detector design
 - p_T modules to produce stubs for seeding
 - **10x reduction** in data, but $\sim 15k$ stubs/BC
 - **$O(\text{Tbps})$**
- Stubs passed to FPGA system with $4 \mu\text{s}$ latency
- Road search and Kalman filter



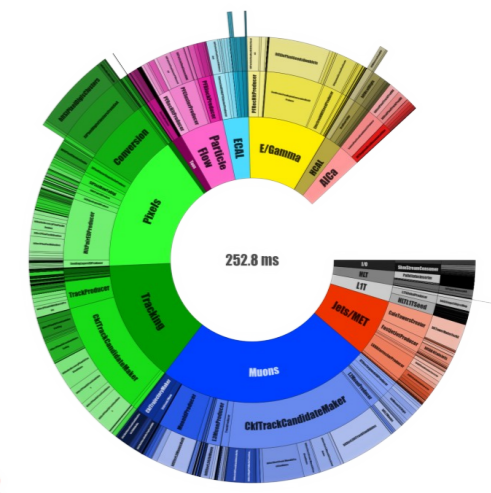
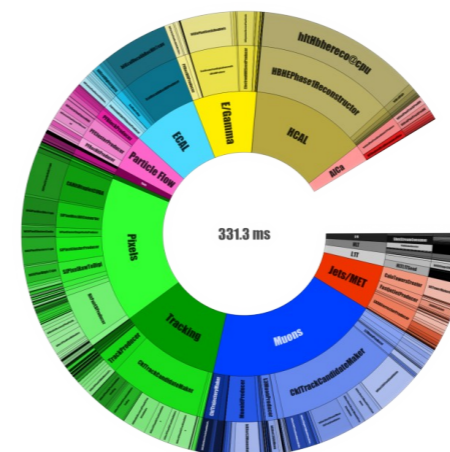
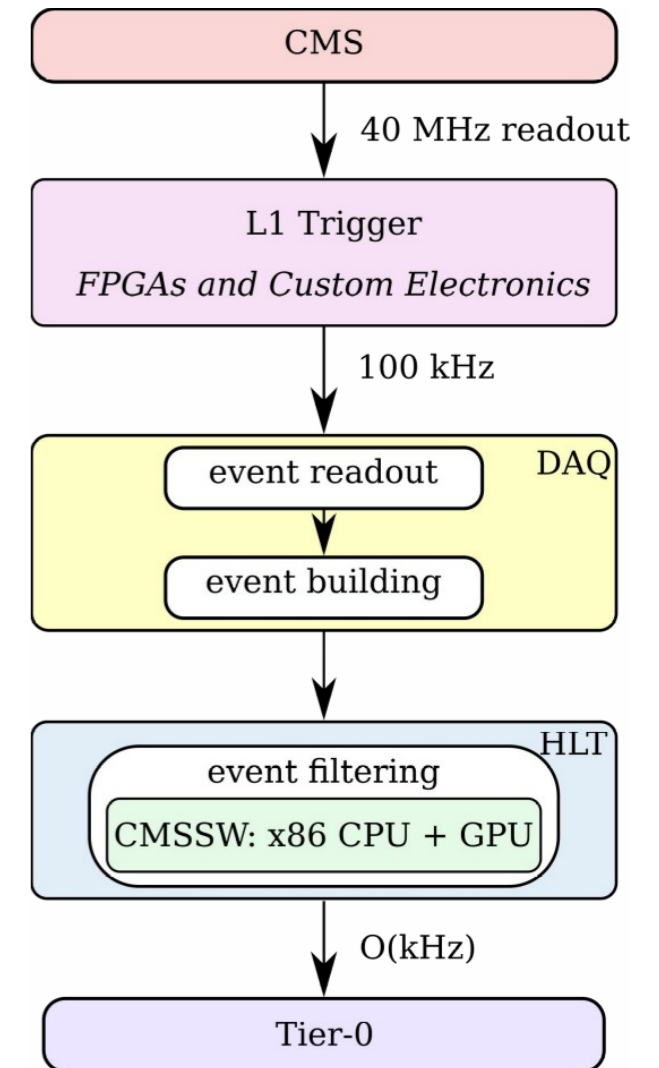
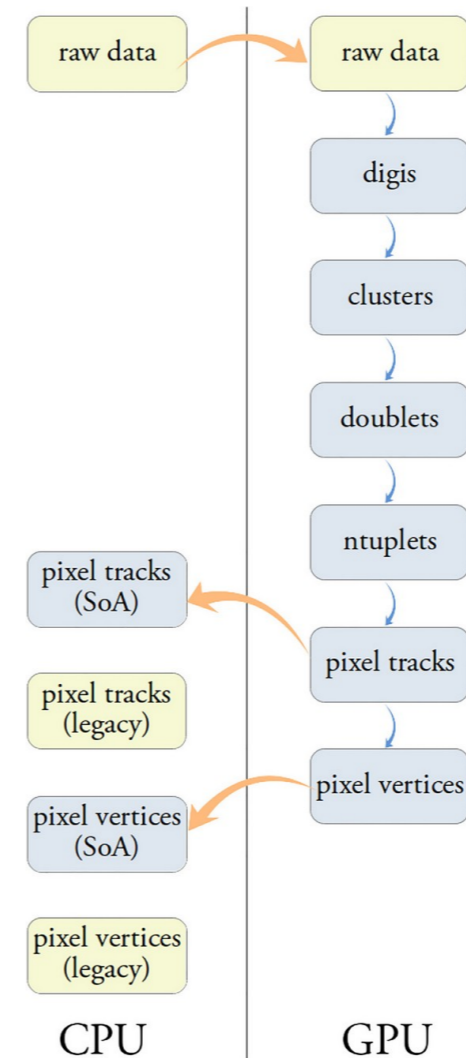
Se
fr



CMS HLT on GPUs

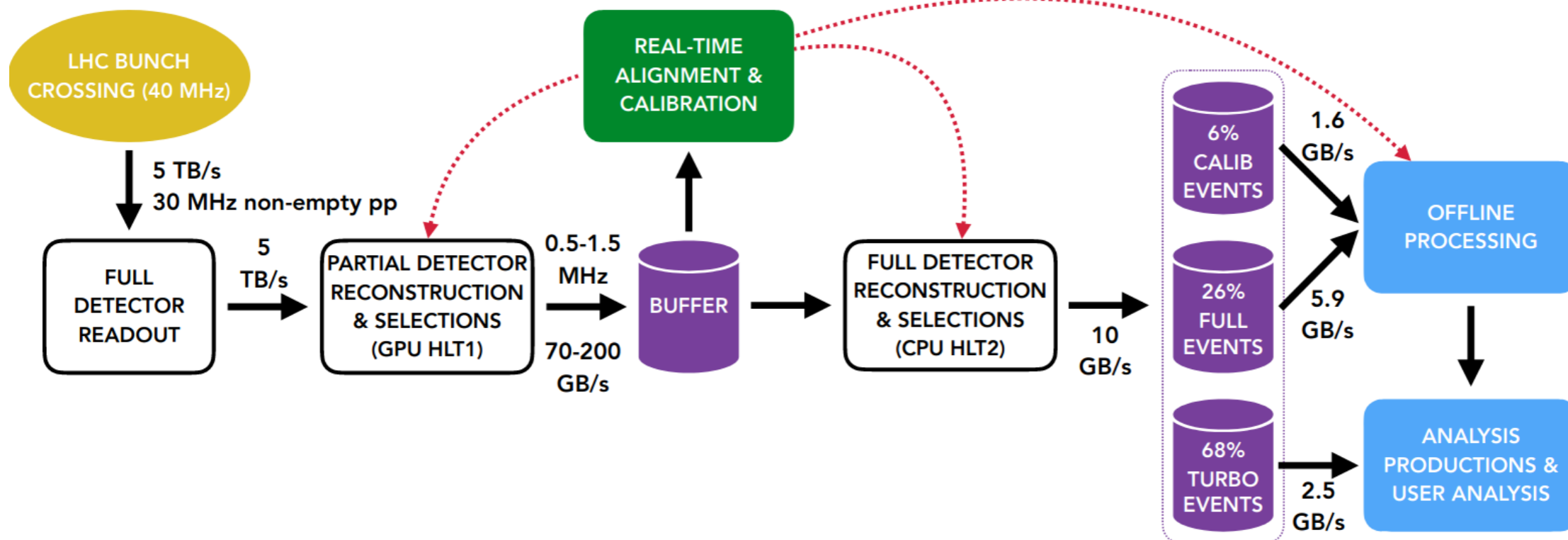
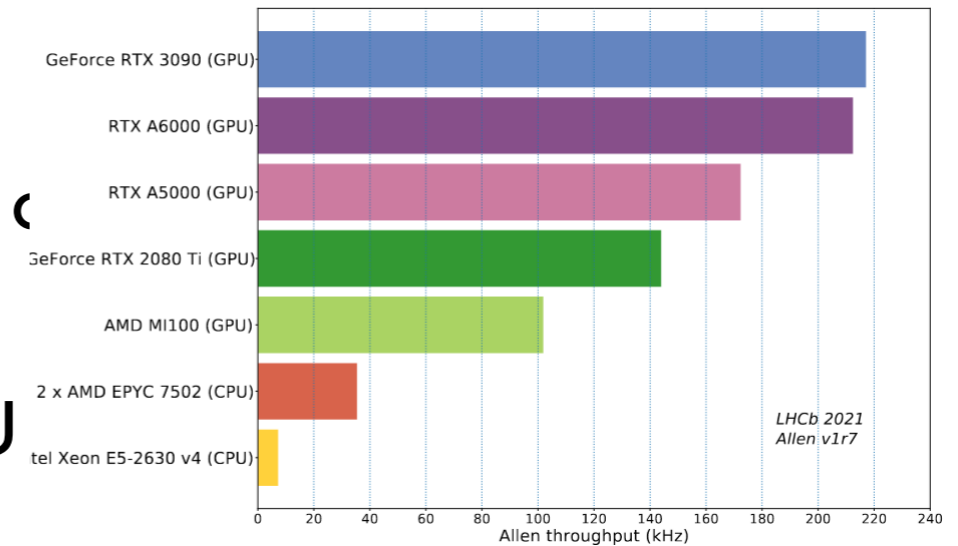
- CMS have already ported several HLT algorithms to run on GPUs for Run 3
 - Pixel track & vertex reconstruction
 - Local calorimeter reconstruction
 - Out-of-time pileup subtraction
- Increases HLT throughput by 25%
- Ongoing work for additional algorithms

Front. Big Data 3 (2020) 601728



LHCb: Triggerless Software Tracking

- LHCb L0 hardware trigger was removed for run-3
 - 30 MHz (32 Tbps) to the HLT
- Allen framework used to perform full tracking on the GPU in HLT
 - Offline quality track reconstruction on GPU in HLT



Conclusion

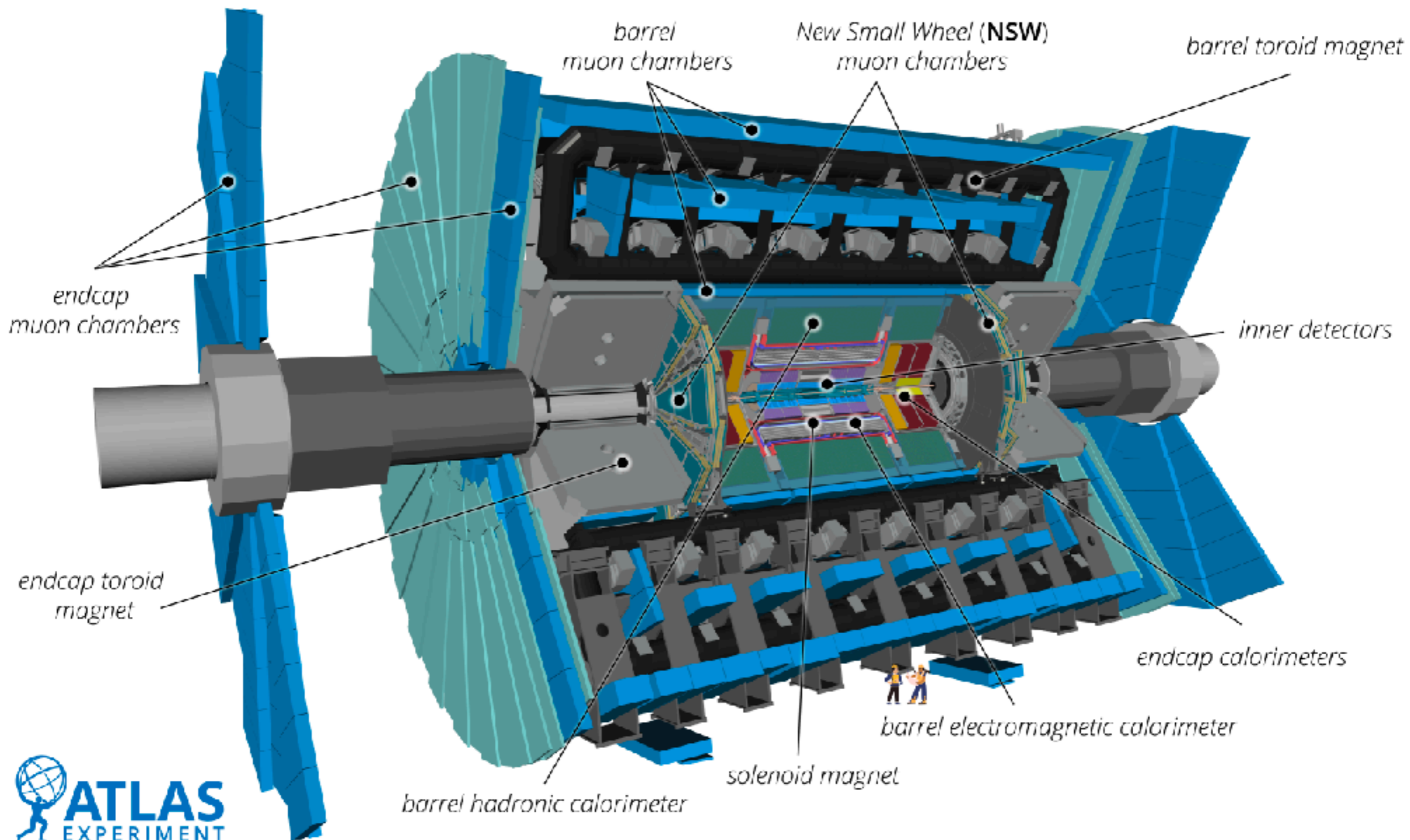
- With Moore's Law coming to its (long prophesied end) beating flat computing budgets with increasing computational demands requires ingenuity
- Novel computing hardware presents one such possibility
- These lectures have provided a brief introduction to such hardware, focussing on the types more commonly used in HEP
- We've also seen examples of how such hardware can be used in HEP
 - But I'm sure there are many more possibilities

Acknowledgements

- These slides are largely based on slides from C. Legget, C. Koraka, D.Vom Bruch [1], [2] and K. Hahn
- Additional material taken from S. Lantz, A. Gheata & S. Hageboek, C. Leggett, B.Yeo
- Further material:
 - C. Koraka, Introduction to Accelerated Computing, CERN Inverted School of Computing 2023
 - J. Lebar, Bringing Clang and C++ to GPUs: An Open-Source, CUDA-Compatible GPU C++ Compiler at CppCon 20156
 - NVidia Cuda Guide

Back Up

ATLAS Detector



The Compact Muon Solenoid (CMS)

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 ($100 \times 150 \mu\text{m}$) $\sim 16\text{m}^2 \sim 66\text{M}$ channels
Microstrips ($80 \times 180 \mu\text{m}$) $\sim 200\text{m}^2 \sim 9.6\text{M}$ channels

SUPERCONDUCTING SOLENOID
Niobium titanium coil carrying $\sim 18,000\text{A}$

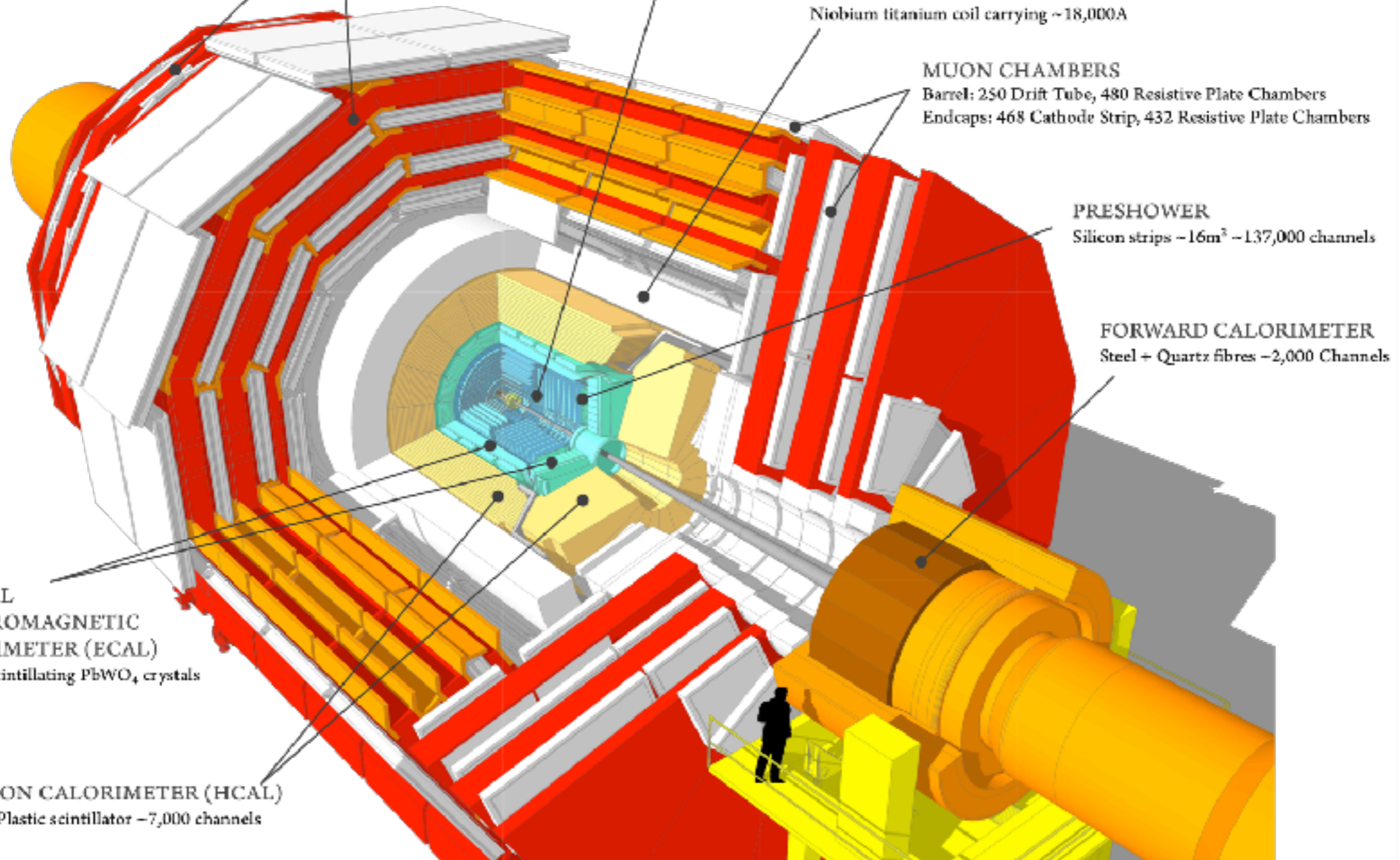
MUON CHAMBERS
Barrel: 250 Drift Tube, 480 Resistive Plate Chambers
Endcaps: 468 Cathode Strip, 432 Resistive Plate Chambers

PRESHOWER
Silicon strips $\sim 16\text{m}^2 \sim 137,000$ channels

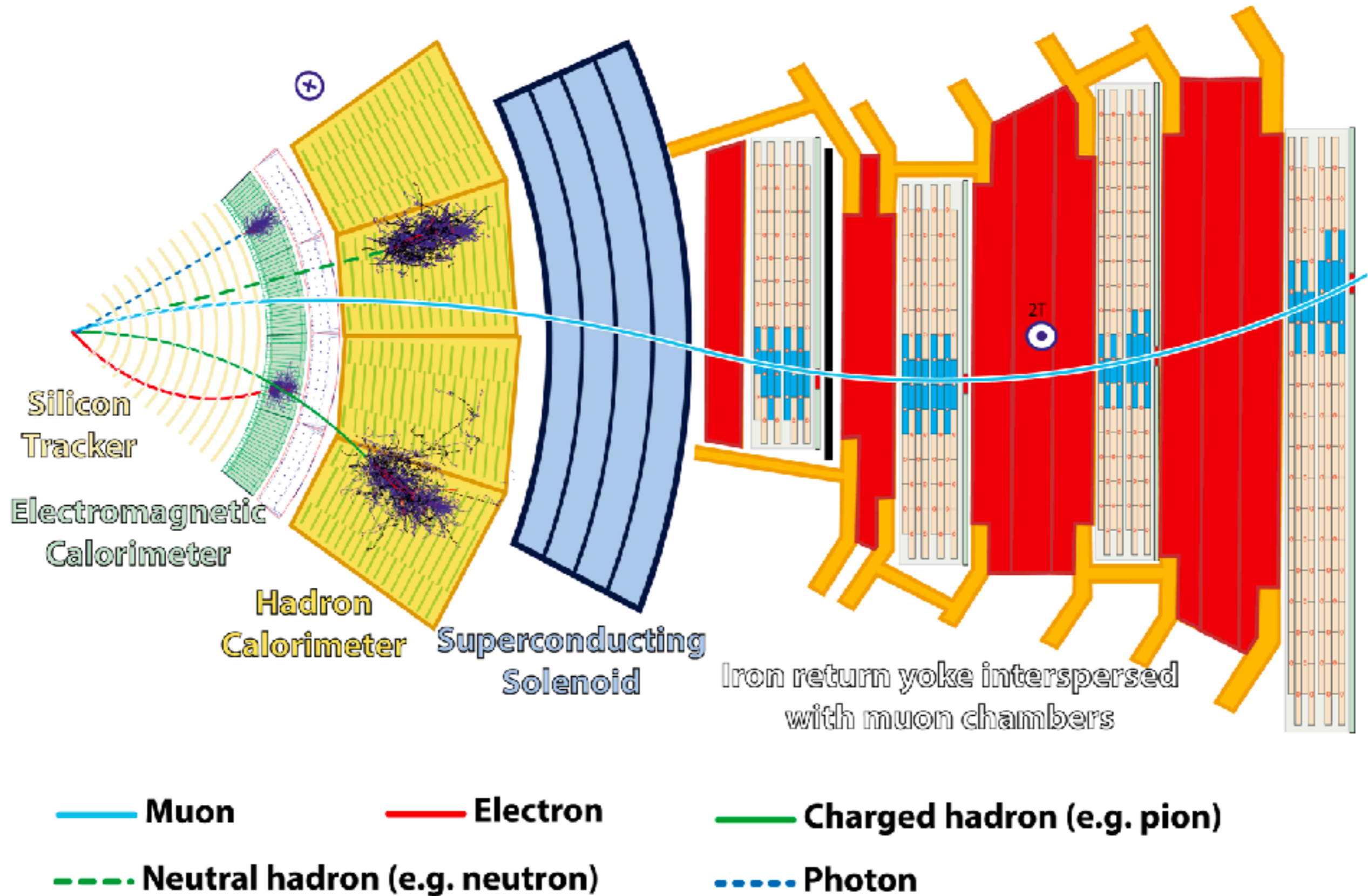
FORWARD CALORIMETER
Steel + Quartz fibres $\sim 2,000$ Channels

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

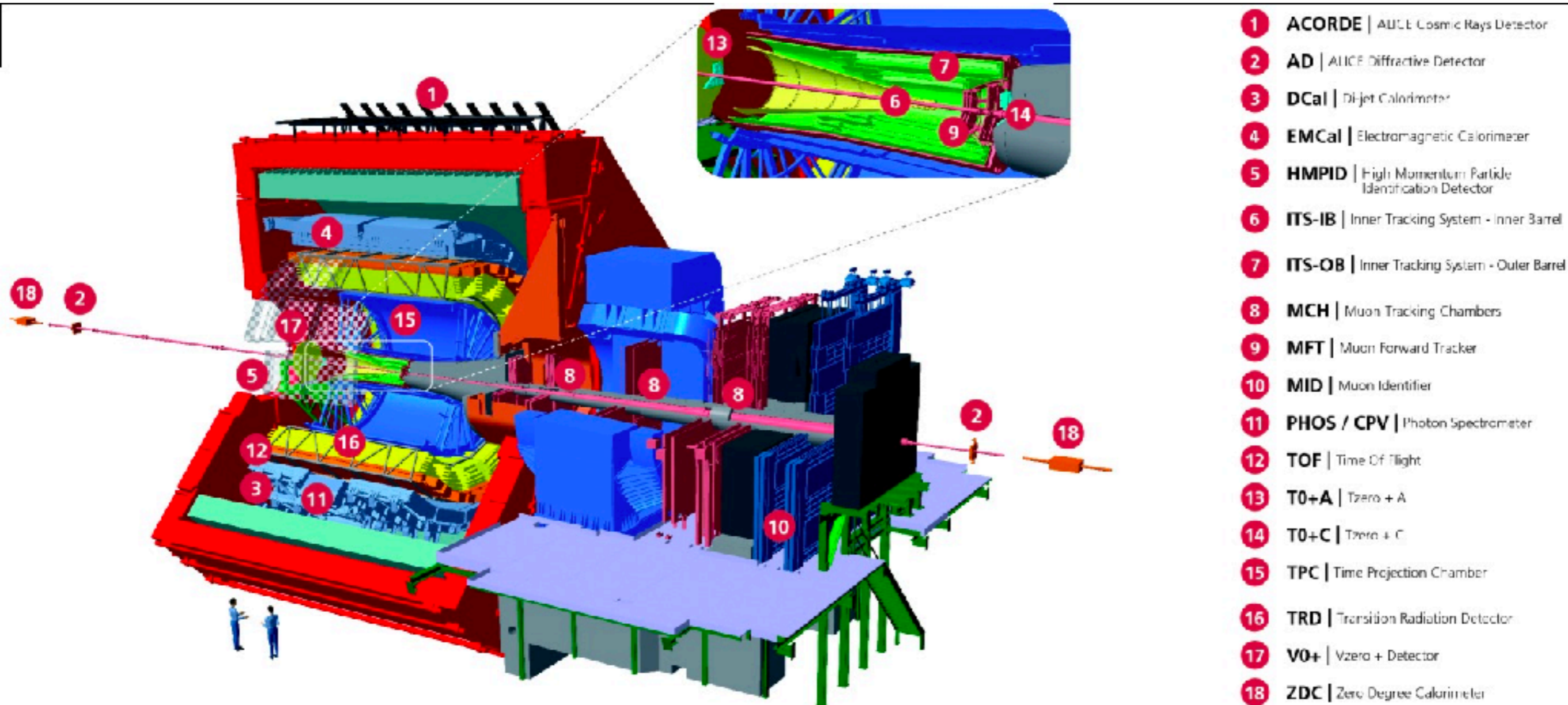
HADRON CALORIMETER (HCAL)
Brass + Plastic scintillator $\sim 7,000$ channels



CMS Detector Slice

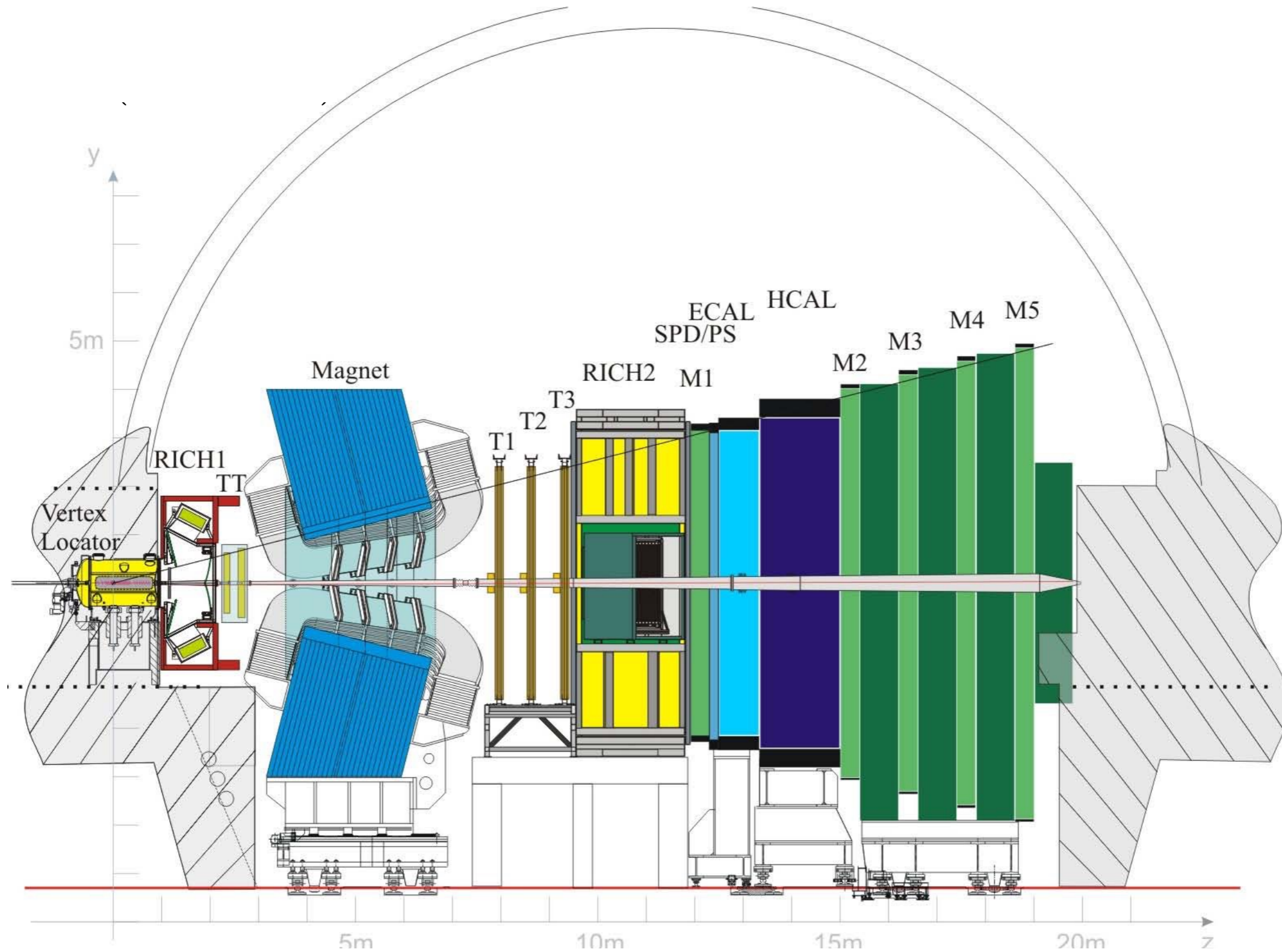


ALICE Detector



Source

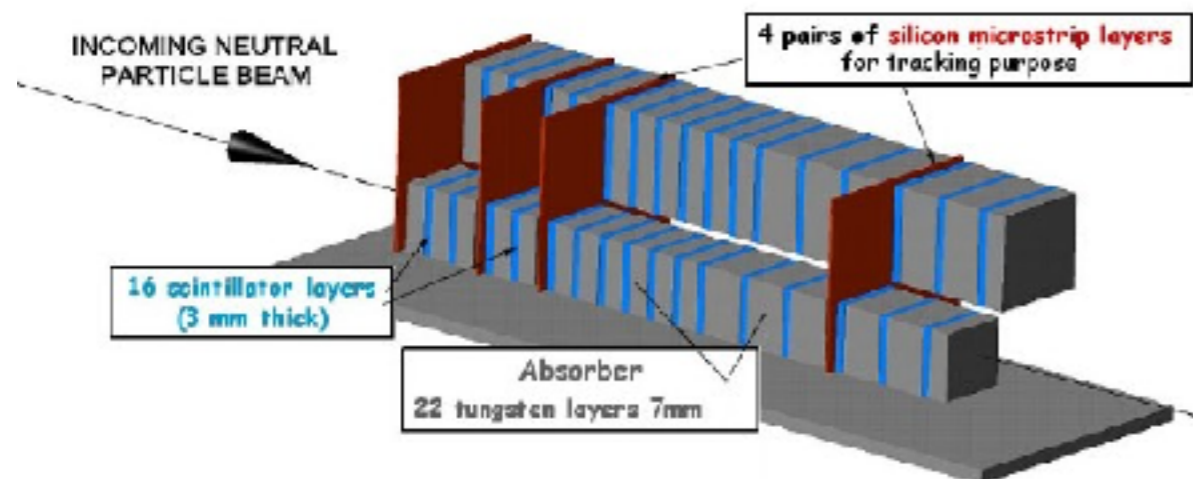
LHCb Detector



Smaller Experiments

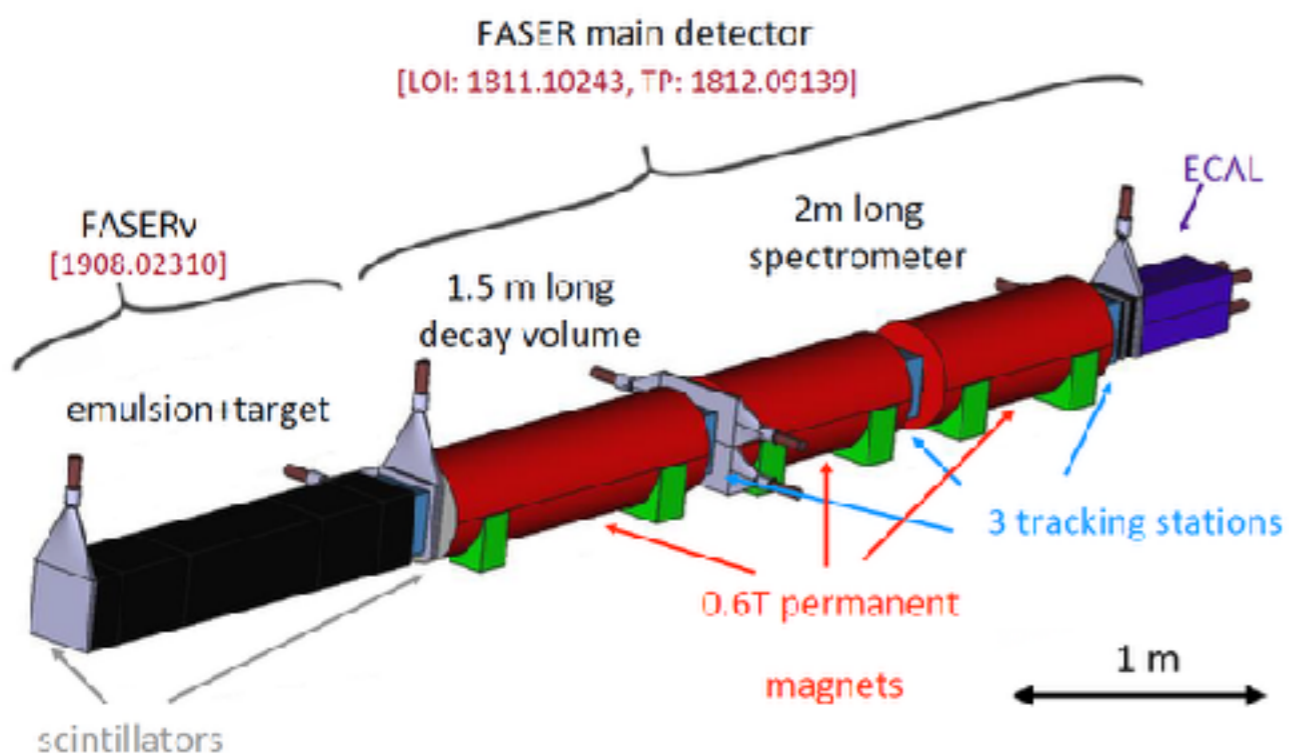
LHCf

neutral-particle production cross sections in the very forward region of proton-proton and nucleus-nucleus interactions



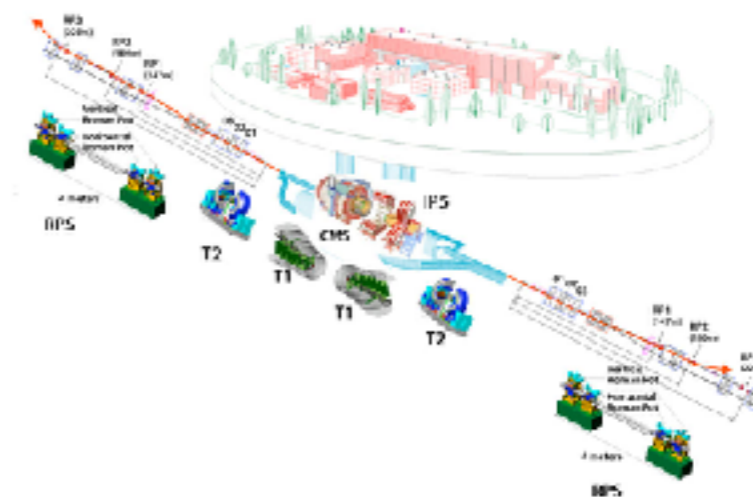
FASER

search for light and extremely weakly interacting particles



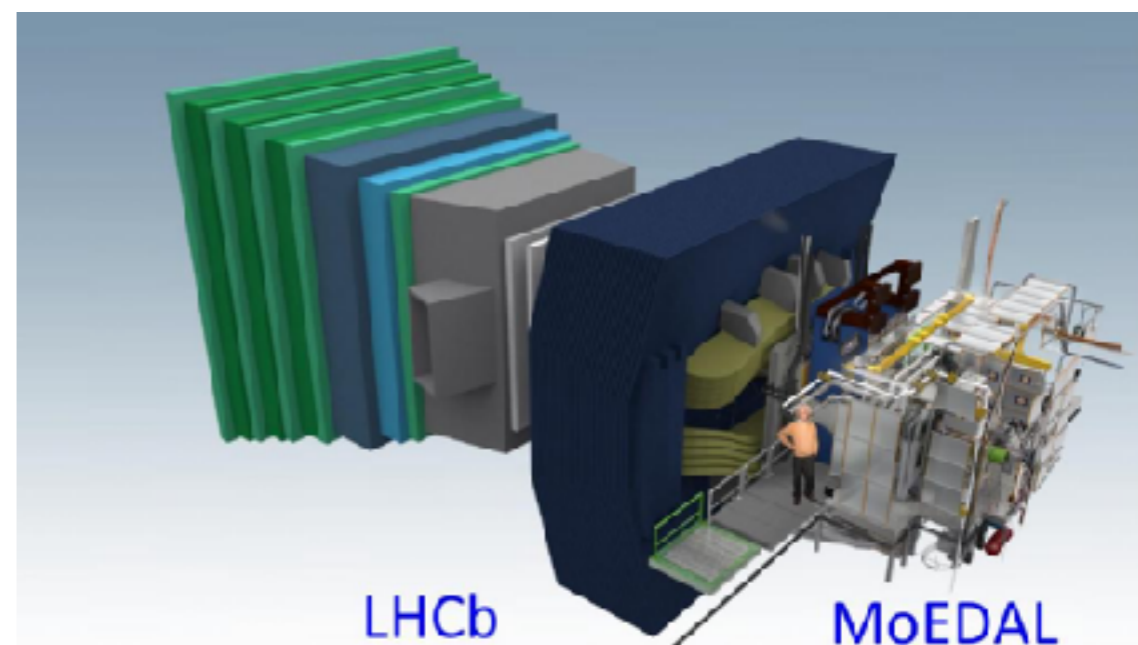
TOTEM

total cross section, elastic scattering and diffraction dissociation measurement at the LHC



MoEDAL

magnetic monopoles or massive (pseudo-)stable charged particles



Cellular Automaton Algorithm

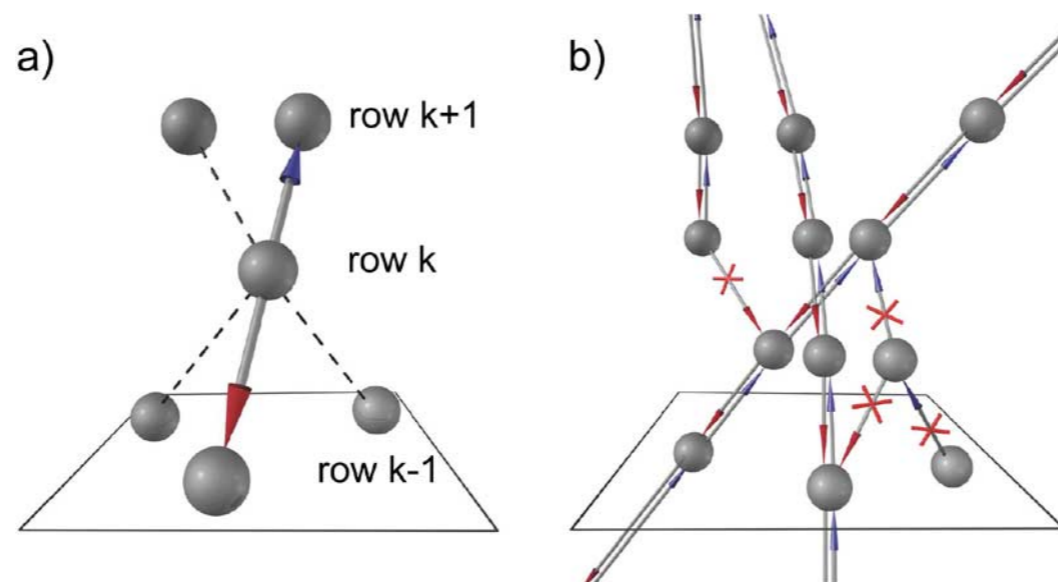


Fig. 7. a) Neighbors finder. b) Evolution step of the Cellular Automaton.

The tracking algorithm starts with a combinatorial search for track candidates (tracklets), which is based on the Cellular Automaton method [3]. Local parts of trajectories are created from geometrically nearby hits, thus eliminating unphysical hit combinations at the local level. The combinatorial processing composes the following two steps:

- 1. Neighbor finder: For each hit at a row k the best pair of neighboring hits from rows $k + 1$ and $k - 1$ is found, as it is shown in Fig. 7(a). The neighbor selection criteria requires the hit and its two best neighbors to form a straight line. The links to the best two neighbors are stored. Once the best pair of neighbors is found for each hit, the step is completed.
- 2. Evolution step: Reciprocal links are determined and saved, all the other links are removed (see Fig. 7(b)).

Every saved one-to-one link defines a part of the trajectory between the two neighboring hits. Chains of consecutive one-to-one links define the tracklets. One can see from Fig. 7(b) that each hit can belong to only one tracklet because of the strong evolution criteria. This uncommon approach is possible due to the abundance of hits on every TPC track. Such a strong selection of tracklets results in a linear dependence of the processing time on the number of track candidates. When the tracklets are created, the sequential part of the reconstruction starts, implementing the following two steps:

Fig. 9. Reconstruction performance for central heavy ion collisions at 5 TeV.

- 3. Tracklet construction: The tracklets are created by following the hit-to-hit links as it is described above. The geometrical trajectories are fit using a Kalman Filter, with a χ^2 quality check. Each tracklet is extended in order to collect hits being close to its trajectory.
- 4. Tracklet selection: Some of the track candidates can have intersected parts. In this case the longest track is saved, the shortest removed. A final quality check is applied to the reconstructed tracks, including a cut on the minimal number of hits and a cut for low momentum.

IV. TRACKER EFFICIENCY

The performance of the HLT track finder of 99.9% for proton-proton events and 98.5% for central Pb-Pb collisions has been verified on simulated events. Corresponding efficiency plots are shown on Figs. 8 and 9. In addition to the high efficiency, the real-time reconstruction is an order of magnitude faster than the off-line algorithm used as reference.

The described algorithm has the advantage of a high degree of locality and parallelism. Step one only searches for local neighbors to each hit. It can be done in parallel for all the hits as the result does not depend on the order of processing. Step three

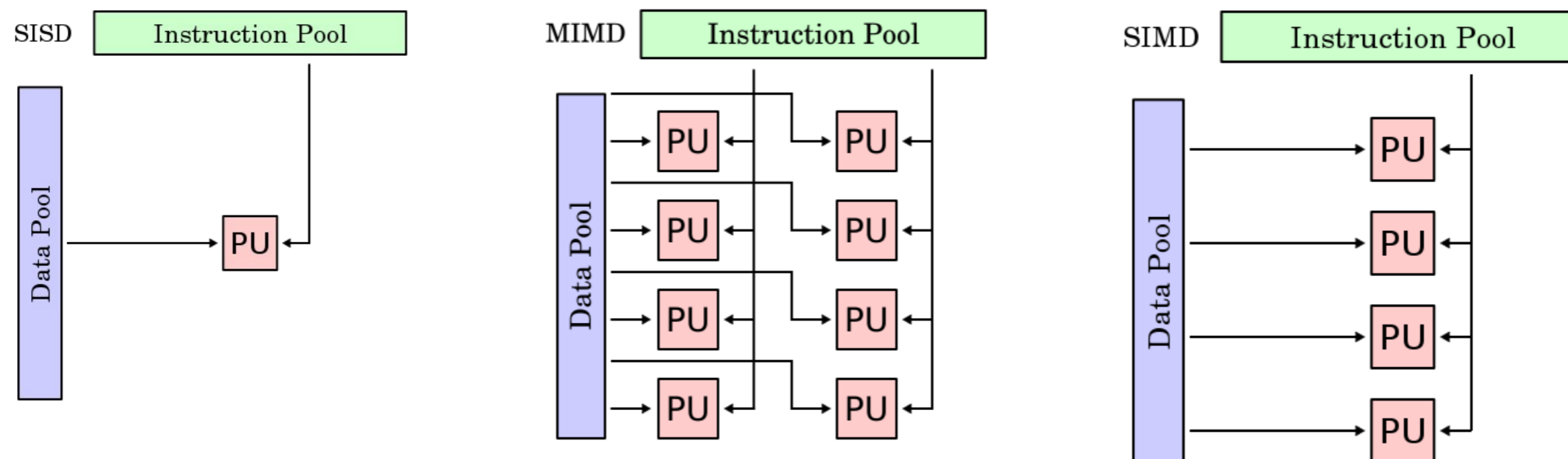
A Brief History of GPUs



- ▶ Original purpose: make pretty pictures on your screen, much faster and prettier than your CPU could do
 - first graphics cards appeared in 1970s in arcade games, and even earlier (1950s) in military flight sims. Mainly used to accelerate sprite graphics
 - 1981: IBM introduced Monochrome Display Adapter: 80 x 26 lines of text!
 - 1983: IBM ups resolution to 256x256x8 colors and monochrome at 512x512
 - 1985: ATI founded, introduces "Wonder" line of cards: start of consumer 2D acceleration
 - 1991: S3 Graphics introduces "86C911", major increase in 2D acceleration
 - OpenGL, and Windows APIs for graphics such as WinG and DirectDraw begin to appear
 - 1994: Sony coins "GPU" (**G**raphical **P**rocessing **U**nit) name for PlayStation graphics card
 - 1995: 3DFx "Voodoo": first mainstream 3D accelerated cards
 - 1999: NVidia introduces GeForce 256: hardware acceleration for transforms, lighting, triangle setup/clipping, rendering
 - 2000+: manufacturers add more hardware for specific tasks, eg shading, explosions, vertex blending, bump mapping, refraction
 - 2007: NVidia releases CUDA development environment: **G**eneral **P**urpose GPU computing
 - GPUs become less specialized, with more generalized computing elements
 - 2009: OpenCL (Open Compute Language) released. targets heterogeneous computing

SISD, MIMD & SIMD

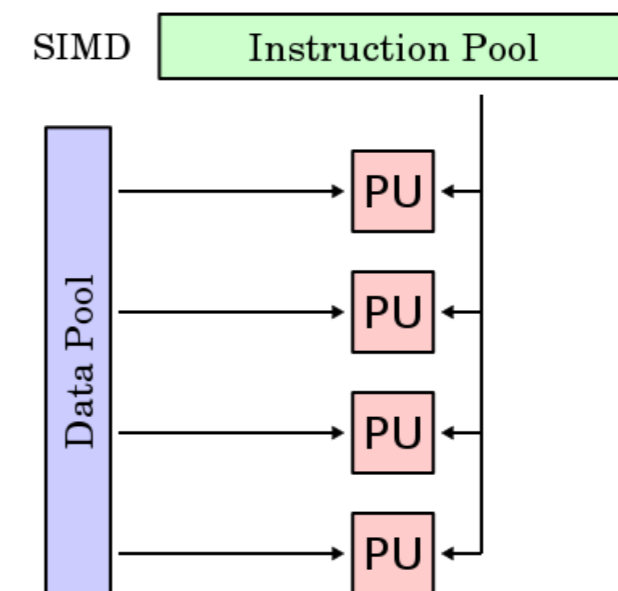
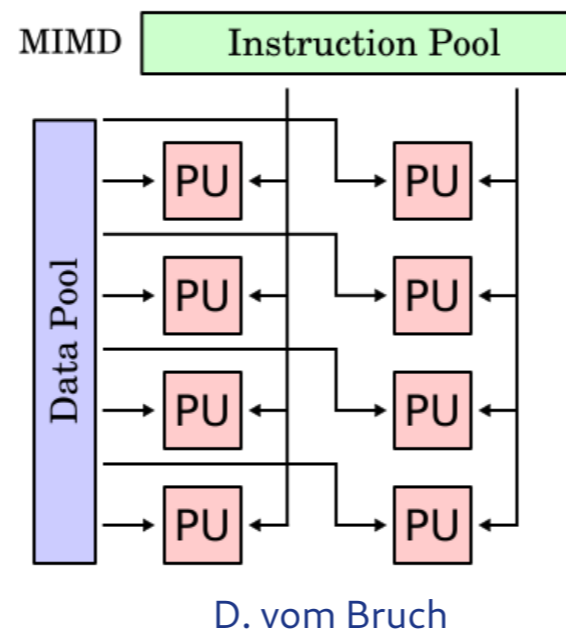
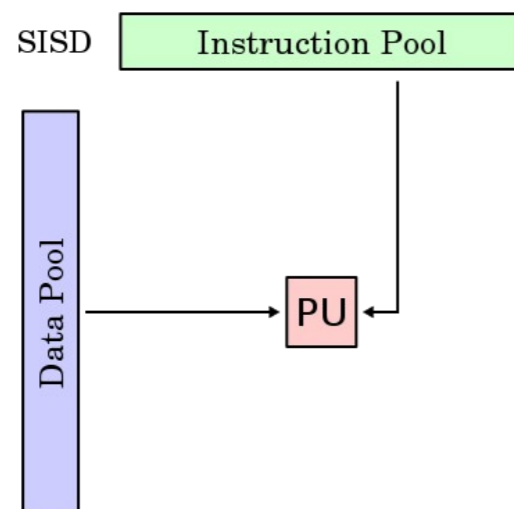
SISD	MIMD	SIMD
Single Instruction Single Data	Multiple Instruction Multiple Data	Single Instruction Multiple Data
Uniprocessor machines	Multi-core, grid-, cloud-computing	e.g. vector processors



D. vom Bruch

Single Instruction Multiple Threads (SIMT)

SISD	MIMD	SIMT
Single Instruction Single Data	Multiple Instruction Multiple Data	Single Instruction Multiple Threads
Uniprocessor machines	Multi-core, grid-, cloud-computing	GPUs



SIMD versus SIMT

SIMD

- Vectorized instructions executed on modern CPU
SIMD cores are executed in lockstep
- No synchronization barrier is needed, as all elements of the vector finish processing at the same time

SIMT

- Similar to programming a vector processor
- Use threads instead of vectors
- No need to read data into vector register
- GPUs consist of multiple processing elements, each with multiple SIMT GPU cores
→ not all threads are processed in lockstep
- A synchronization instruction is required on GPUs

Heterogeneous solutions & sustainability: Green500

Rank	TOP500 Rank	System	Cores	Rmax (TFlop/s)	Power (kW)	Power Efficiency (GFlops/watts)
1	301	MN-3 - MN-Core Server, Xeon Platinum 8260M 24C 2.4GHz, Preferred Networks MN-Core, MN-Core DirectConnect, Preferred Networks Preferred Networks Japan	1,664	2,181.2	55	39.379
2	291	SSC-21 Scalable Module - Apollo 6500 Gen10 plus, AMD EPYC 7543 32C 2.8GHz, NVIDIA A100 80GB, Infiniband HDR200, HPE Samsung Electronics South Korea	16,704	2,274.1	103	33.983
3	295	Tethys - NVIDIA DGX A100 Liquid Cooled Prototype, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100 80GB, Infiniband HDR, Nvidia NVIDIA Corporation United States	19,840	2,255.0	72	31.538
4	280	Wilkes-3 - PowerEdge XE8545, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 80GB, Infiniband HDR200 dual rail, DELL EMC University of Cambridge United Kingdom	26,880	2,287.0	74	30.797
5	30	HiPerGator AI - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Infiniband HDR, Nvidia University of Florida United States	138,880	17,200.0	583	29.521

- All top 5 Green500 use accelerators
- 4/5 use Nvidia GPUs combined with AMD Epyc
- MN-3 uses an accelerator optimized for matrix arithmetic
- Of the top 30 Green500:
 - 26 use Nvidia GPUs
 - 3 use A64FX vector-processors (ARM)
 - 1 uses a many-core microprocessor (PEZY-SC3)