Frameworks

Adam Barton
Lancaster University

Acknowledgments

CMS Heterogeneous computing: Felice Pantaleo, Dr Andrea Bocci

ATLAS GPU Tests: Attila Krasznahorkay

LHCb Framework: Marco Clemencic

ALICE: Giulio Eulisse

Graeme A Stewart

Introduction

- With the advent of the clock speed crisis, hardware design has needed to exploit alternative designs to increase performance:
  - Multi-cores
  - Wide vector operations
  - Pre-fetching caches

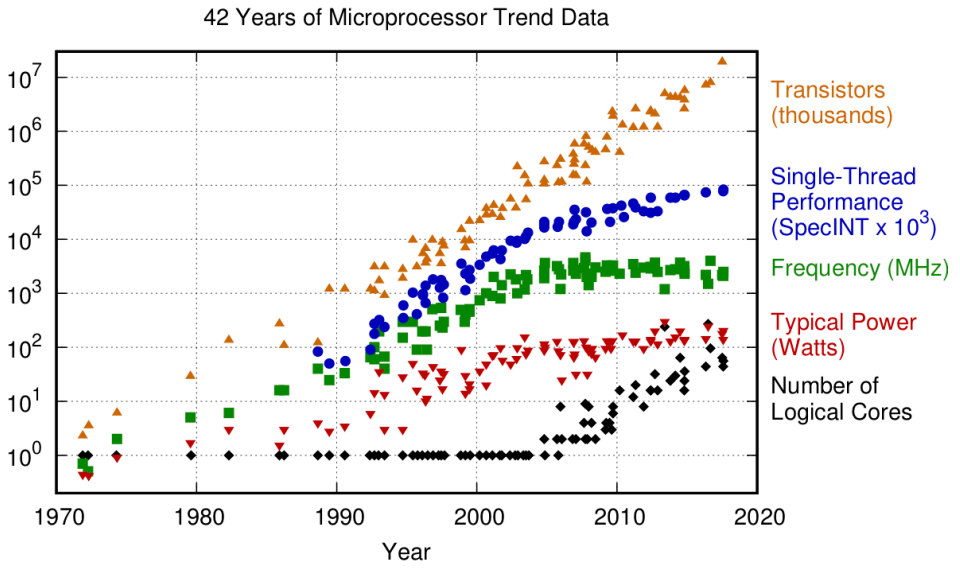
Alternative processors

- GPUs
- FPGAs
- TPUs

- These technologies usually require specific coding styles or APIs to properly exploit. This has led to an explosion in languages and frameworks and extra work porting code between them.

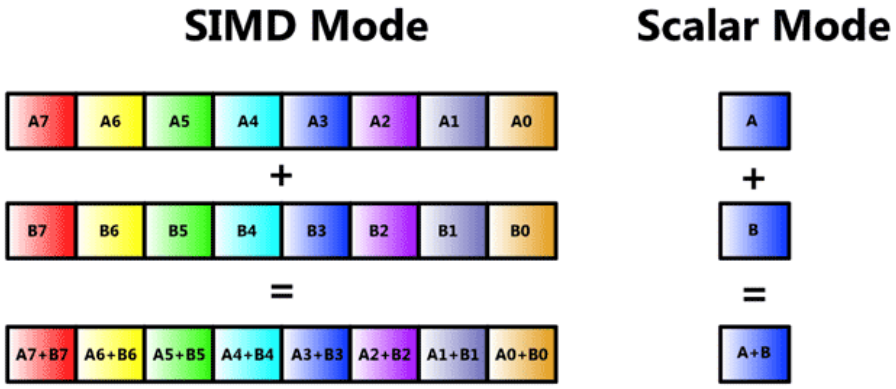
What is the clock speed crisis?

- From 1970 to 2005 processor/core speed increased exponentially
- By 2000, programmers were getting disillusioned and lazy – why work to improve your code when you can wait 2 years and have it run twice as fast after you upgrade?
- During this period you got an emphasis on abstraction and ‘safe’ programming, languages like C# and Java were thought to be the future.
- By 2005 the clock speed maxed out at 3 to 4 GHz. To continue Moore’s law, chip makers starting producing multiple core chips. This stalled the performance of any serial codes (all HEP codes in those days).



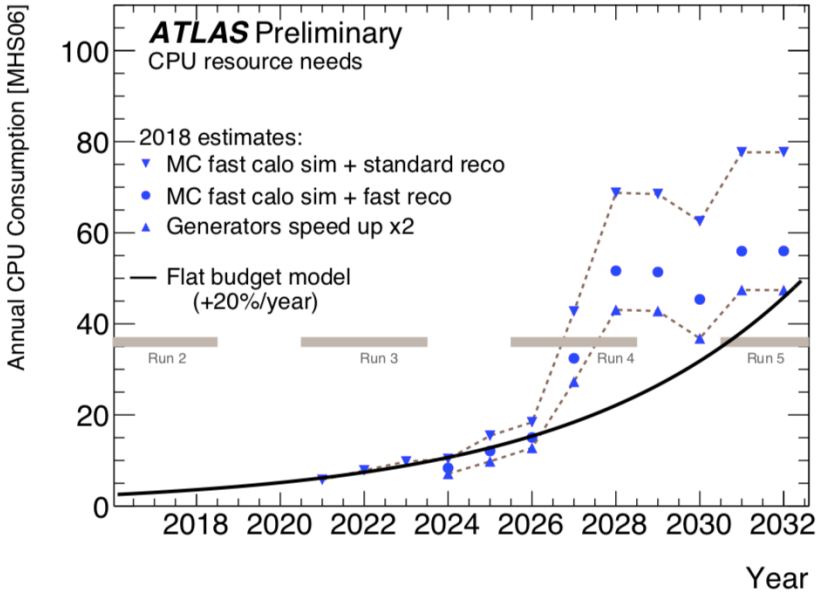
Vectorization (or SIMD)

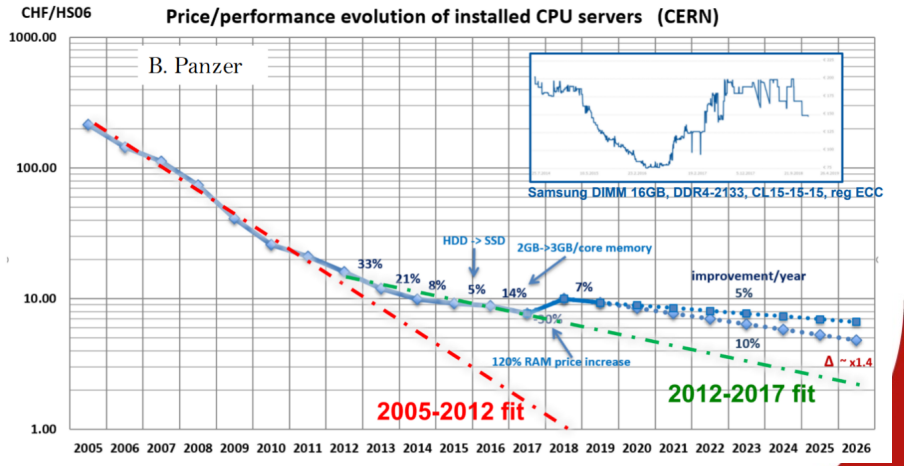
- This involves performing the same operation on multiple items on the processor in one step.
- But your code and memory must be arrange in a contiguous manner in order to exploit this.
- SIMD is also hard to exploit when you have a stochastic element or have a lot of logical branches in your logic. Which unfortunately applies to simulation and tracking quite heavily.
- ATLAS will not be exploiting this extensively until future phases, LHCb will be exploiting it much sooner



ATLAS Challenges

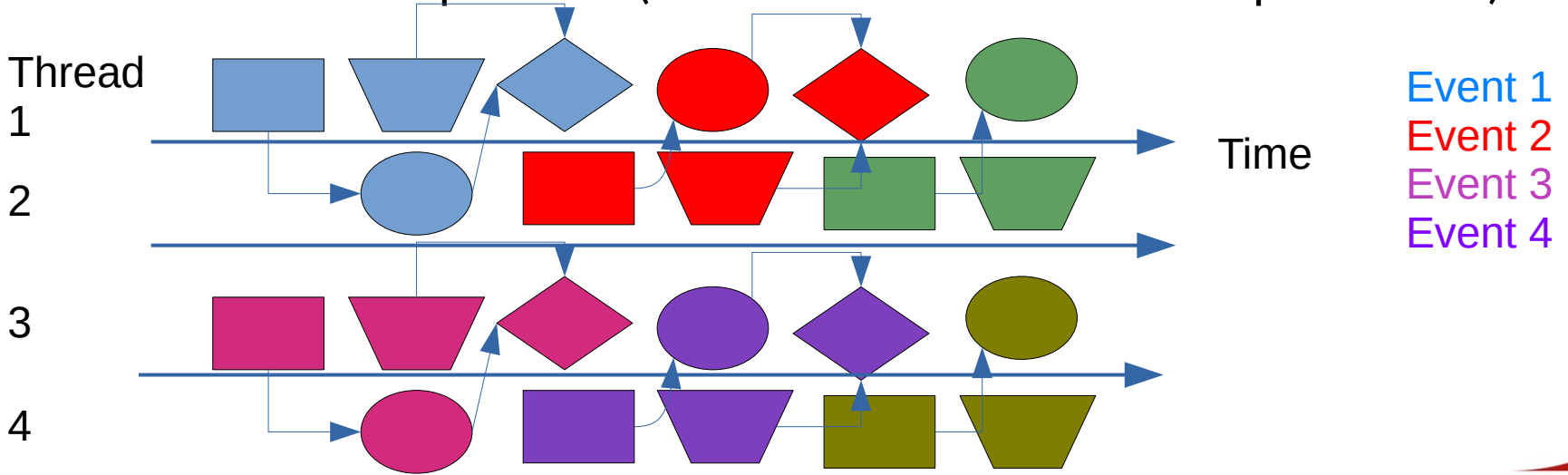
- Memory costs prohibit ATLAS running software in multiple processes.
  - Already in advanced development but incomplete - rewrite Athena to run multi-threaded.
- Run 4: 5 x Pileup, ~7 x HLT rate, more readout channels - increasing CPU requirements.
- Disk storage requirements exceed flat budget estimate.





ATLAS: AthenaMT

- AthenaMT will use components of the multi-threaded framework Gaudi.
- AthenaMT will use the *Intel Thread Building Blocks* library to execute algorithms on available CPU threads.
- Algorithms using data from one event/collision can be parallelized and multi events can also run in parallel . (**Intra**-event and **Inter**-event parallelism).



Lancaster University

ATLAS: Portability Solutions

- More interested in “Portable” than “Performant”, only want to write code once.
- Want to use high level C++ if possible
- Looked at
  - Kokkos/Raja – currently only Nvidia, most performant
  - SYCL – can’t chain kernels, single source C++ can target CPU
  - OpenMP, ugly pramas, broadest support on HPCs
  - HIP, no intel support, good for AMD

ATLAS: Accelerators

- ATLAS doesn't use any accelerators in central production yet
- An evaluation was done in 2013-2015
  - Evaluated CUDA, concluded it was too much work to rewrite a significant amount of code give the relative low performance boost.
- Datacenters are providing GPUs as standard now
- ATLAS' computing model will be moving towards a more GPU friendly system anyway.

ATLAS GPU Tests

- ATLAS does a series of test on GPUs using OpenCL, CUDA (most mature) and SYCL.
- The tests were not using any “real” ATLAS reconstruction code.
- CPU-only algorithms are arbitrary “crunching” code.
- GPU emulated:
  - The test jobs measure during initialisation how many FLOPS the CPU can do in a single thread in a unit of time.
  - With this information FLOPS values are associated to the time values stored in the Gaudi data files.
  - The GPU tasks then execute this number of FLOPS on small arrays, with some configurable multipliers applied.

Lancaster University

ATLAS: Reconstruction Emulation Results

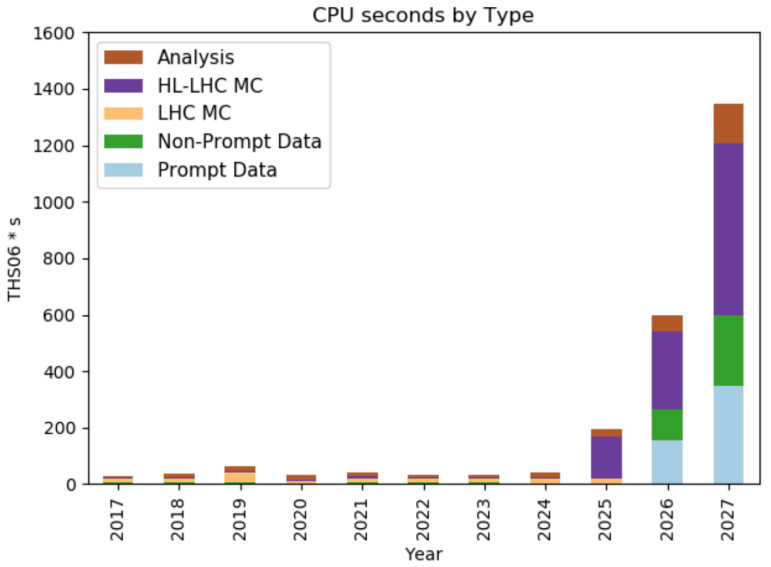
Attila Krasznahorkay

- Reference job: using only CPU code crunching and validated with portable code running on CPU.
- Configured 3 CPU intensive reconstruction to run on NVIDIA GPU with CUDA.
- Comments:
  - Algorithms off of the “critical path” can handle being executed less efficiently on an accelerator, but not by much.
  - Clearly not working as efficiently as it should.
- Early tests on Intel oneAPI are not working well.

Setup	Time [s]
50 events, 8 threads, CPU-only algorithms	68.3 ± 0.47
50 events, 8 threads, 3 “critical-path” CPU/GPU algorithms, run only on CPUs	68.1 ± 0.66
50 events, 8 threads, 3 “critical-path” algorithms offloaded with ideal FPOPS	54.5 ± 0.47
50 events, 8 threads, 3 “critical path” algorithms offloaded with 10x FPOPS	151.2 ± 27.2
50 event, 8 threads, 4 “heavy non-critical-path” algorithms offloaded with ideal FPOPS	49.5 ± 1.51
50 events, 8 threads, 4 “heavy non-critical-path” algorithms offloaded with 3x FPOPS	70.3 ± 10.0

CMS

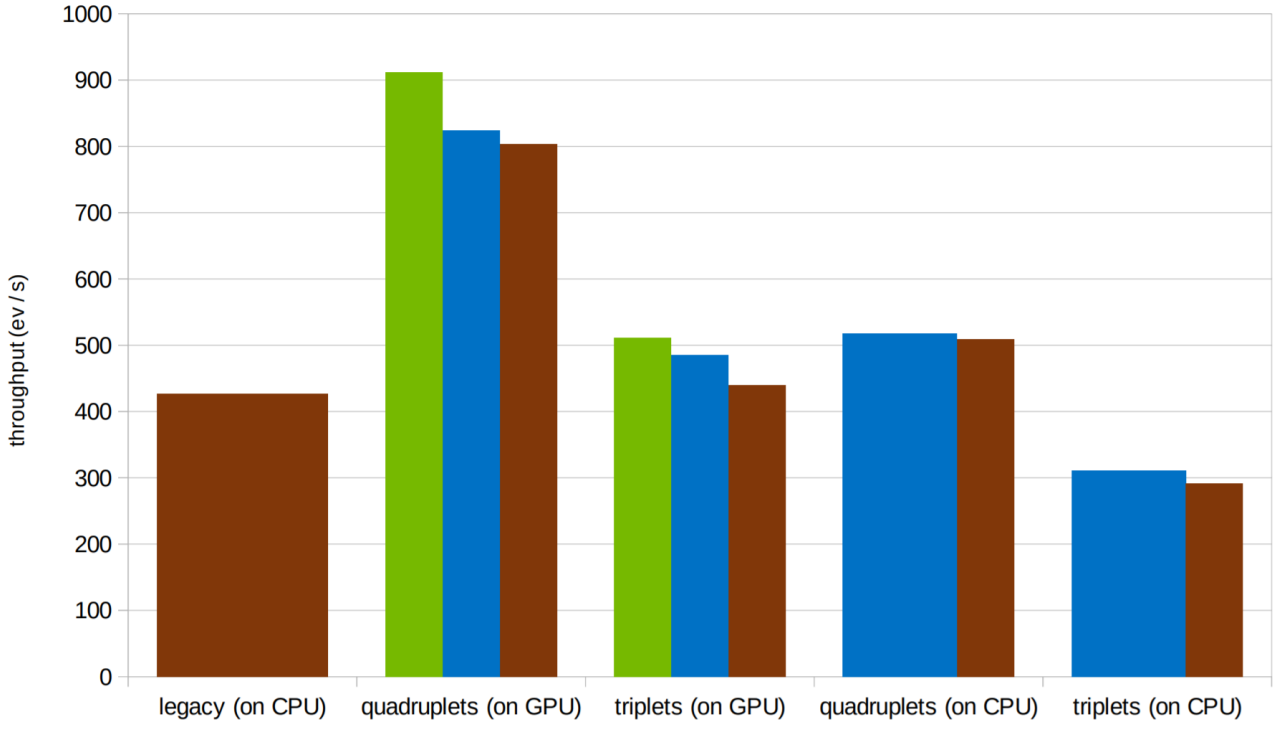
- CMS has similar challenges to ATLAS and similar solutions in terms of multi-threading.
- CMS already had multithreading during run2
- Immediate plans includes GPU heterogeneous offloading.
- Run 3 (2021 – 2024)
  - online reconstruction : offload 20% to 30% of the computing needs to NVIDIA GPUs
  - Offline reconstruction: leverage opportunistic resources, Intel, NVIDIA, AMD GPUs
- Run 4+ (2027-…)
  - 30x more CPU performance online and offline
  - fully heterogeneous online and offline reconstruction
- Needs way to write portable code for this to be viable.



CMS: Heterogeneous Run3 HLT Farm

- CMS wants a heterogeneous HLT farm well before Run-4
- 30% of the HLT reconstruction algorithms seem like a good candidate for porting
- What does CMS aim to gain in the short term?
  - Better physics performance
  - Reconstruction able to run on Supercomputers
  - Expertise in the “inevitable” Heterogeneous computing

CMS Heterogeneous performance



pixel tracks and vertices global reco

CPU

- dual socket Xeon Gold 6130
- 2 × 16 cores (2 x 32 threads)
- throughput measured on a full node
- 4 jobs with 16 threads

GPU

- single NVIDIA Tesla T4
- 2560 CUDA cores
- single job with 10-16 concurrent events

transfer from GPU to CPU

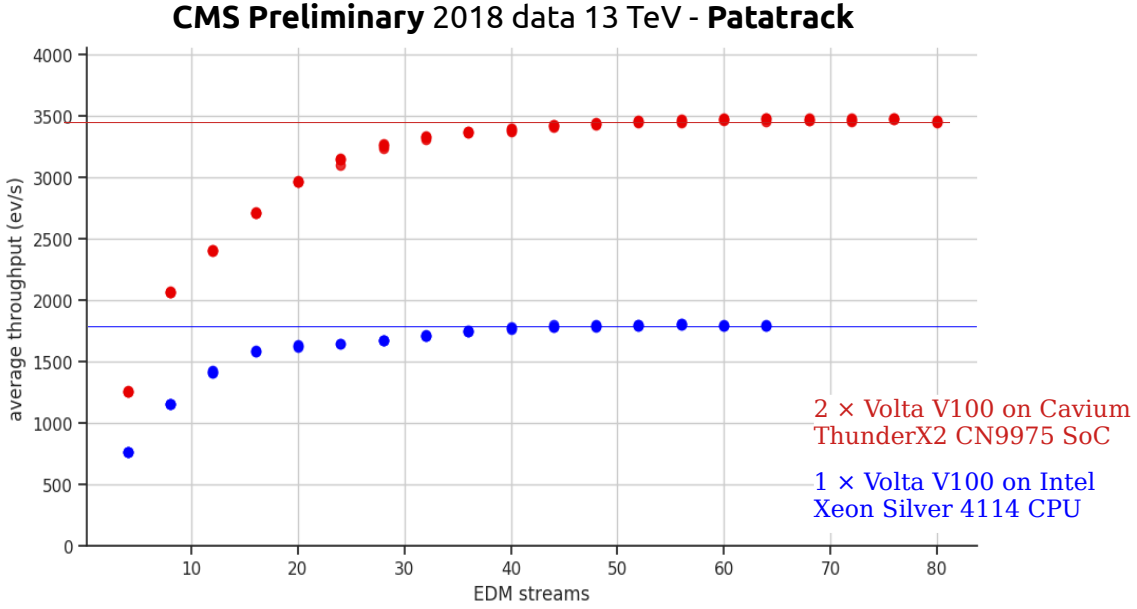
- on demand
- small impact on event throughput

conversion to legacy data formats

- on demand, to be minimised
- small impact on event throughput
- high cost in CPU usage

ARM with powerful GPU

- The CPU market may start favouring ARM CPUs
- If your work can be offset to GPUs you can forgo expensive x86 processors all together



Cavium ThunderX2 CN9975 SoC

- 2 × Volta V100 32GB
- 2 x 5120 Volta cores
- 4 jobs per GPU
- 1737 ± 6 ev/s / GPU

Intel Xeon Silver 4114

- 1 × Volta V100 32GB
- 1 x 5120 Volta cores
- 4 jobs per GPU
- 1800 ± 5 ev/s

CMS: Intel OneAPI

- CMS' raw pixel data decoding is ported to different frameworks
  - Intel oneAPI
    - "host" platform emulation
    - OpenCL Intel Core or Xeon CPUs
    - Open Intel gen9 GPUs
    - Work in progress: Intel Arria 10 FPGAs
  - OneAPI beta 3: Stable distribution and documentation, compilers, CUDA to oneAPI tools.
- CMS' software is already in a very good position to exploit oneAPI.
- Extremely parallelisable software that can be fully heterogeneous in the medium to long term.

Device	OneApi Test 1699 modules
Host emulation	5888.52 us
CPU	5728.53 us
GPU (3.5x faster)	**1623.66 us**

LHCb

- LHCb are aiming to upgrade all their code for run 3:
  - Use **Gaudi**
  - All algorithms will be used as pure functions (constrains users)
  - All algorithms are re-entrant
- LHCb have small events with extremely tight time budget
  - Overhead of Gaudi Avalanche Scheduler is not acceptable
  - We now have *HLTControlFlowMgr*
- There is a prototype GPU port for HLT1, but not yet decided if it will go into production

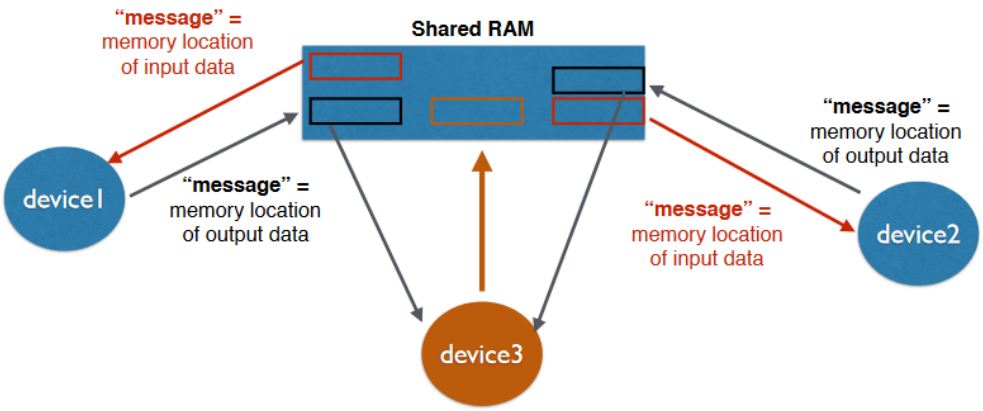
Lancaster University

HLTControlFlowMgr: a Low Overhead Scheduler

- One event per thread serialized execution of Algorithms on each event.
- Order of execution:
  - Based on data and control flow dependencies
  - No need for intra-event synchronization
  - Early exit from chains implemented as a jump
- Optimization work already in progress

Alice - Fair Framework Collaboration

- Goal: develop and support common software solutions for the Run3 of the ALICE LHC experiment and upcoming experiments.
- Based on the experiences of ALICE HLT in Run1 / Run2 and the of the FairRoot framework.
- Data processing happens in separate processes, called devices, exchanging data via a shared memory backed Message Passing paradigm.



Alice framework in one slide

Data Processing Layer (DPL)

Abstracts away the hiccups of a distributed system, presenting the user a familiar "Data Flow" system.

- ***Reactive-like design*** *(push data, don't pull)*
- ***Declarative Domain Specific Language*** *for topology configuration (C++17 based).*
- ***Integration*** *with the rest of the production system, e.g. Monitoring, Logging, Control.*
- ***Laptop mode****, including graphical debugging tools.*

Data Layer: O2 Data Model

Message passing aware data model. Support for multiple backends:

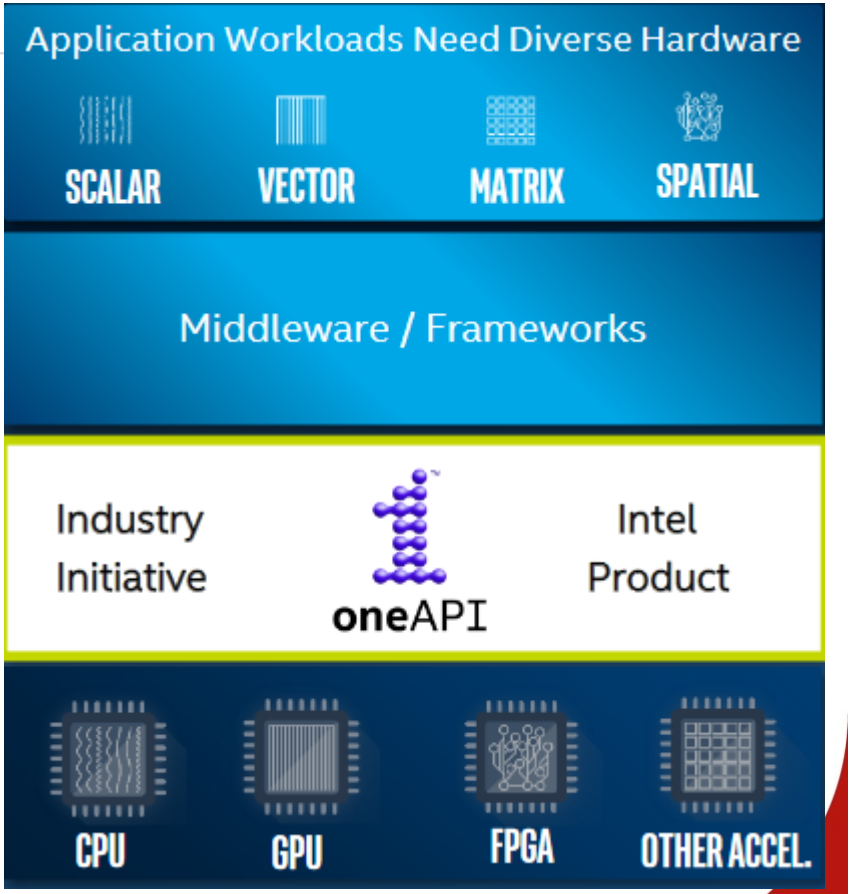
- **Simplified, zero-copy** *format optimised for performance and direct GPU usage. Useful e.g. for TPC reconstruction on the GPU.*
- **ROOT based serialisation.** *Useful for QA and final results.*
- **Apache Arrow based.** *Useful as backend of the analysis ntuples and for integration with other tools.*

Transport Layer: ALFA / FairMQ[1]

- **Standalone processes** *for deployment flexibility.*
- **Message passing** *as a parallelism paradigm.*
- **Shared memory** *backend for reduced memory usage and improved performance.*

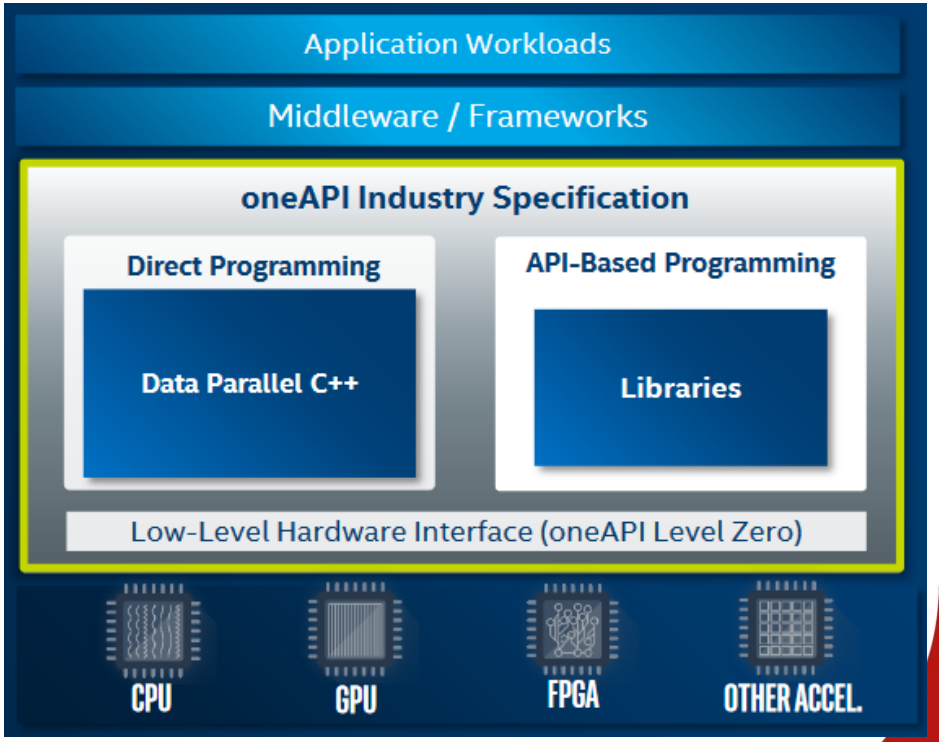
Introducing Intel oneAPI

- A project to deliver a unified software development environment across CPU and accelerator architectures.
- Unified and simplified language and libraries for expressing parallelism
- Delivering native high-level language performance
- Based on industry standards and open specifications



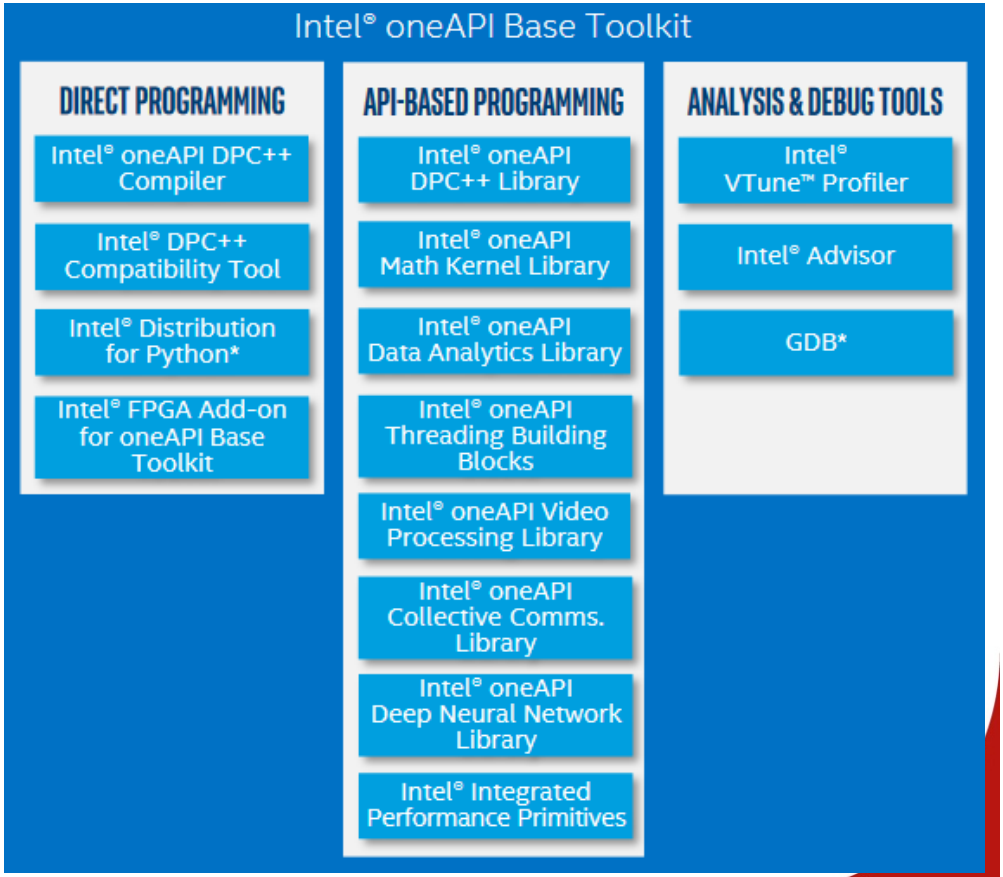
Introducing Intel oneAPI

- Open standard specification to promote community and industry vendors support and includes:
  - Direct programming flow with an open, unified language: DPC++ based on C++ with SYCL extensions
  - API-based programming flow with a set of powerful libraries designed for each hardware to accelerate key domain-specific functions, most of them open sourced
  - Specification of Low-level interface to provide a hardware abstraction layer to vendors



Intel Tools

- Top Features/Benefits
  - Data Parallel C++ compiler, library, and analysis tools
  - tool helps migrate existing code written in CUDA
  - Python distribution includes accelerated scikit-learn, NumPy, SciPy libraries
  - Optimised math libraries
  - Intel Vtune profiling tools
  - Intel Advisor – recommends action optimising code
  - Intel debugger GUI
  - DevCloud – sandbox for developers



Lancaster University

Conclusions

- The after the core boom the future appears to be heterogeneous computing
- CMS have started working on applying this
- ATLAS has been investigating various systems
- ALICE has an interesting alternative design
- Intel OneAPI is, on paper, a promising framework for handling heterogeneous hardware with “minimal” code rewrites

Lancaster University

Back Up

Lancaster University

Example code

```
std::unique_ptr<queue> q = initialize_device_queue();

// The range of the arrays managed by the buffer
range<1> num_items{ array_size };

// Buffers are used to tell DPC++ which data will be shared between the host
// and the devices because they usually don't share physical memory
// The pointer that's being passed as the first parameter transfers ownership
// of the data to DPC++ at runtime. The destructor is called when the buffer
// goes out of scope and the data is given back to the std::arrays.
// The second parameter specifies the range given to the buffer.
buffer<cl_int, 1> addend_1_buf(addend_1.data(), num_items);
buffer<cl_int, 1> addend_2_buf(addend_2.data(), num_items);
buffer<cl_int, 1> sum_buf(sum.data(), num_items);
```

Example Code

```
// queue::submit takes in a lambda that is passed in a command group handler
// constructed at runtime. The lambda also contains a command group, which
// contains the device-side operation and its dependencies
q->submit([&](handler &h) {
    // Accessors are the only way to get access to the memory owned
    // by the buffers initialized above. The first get_access template parameter
    // specifies the access mode for the memory and the second template
    // parameter is the type of memory to access the data from; this parameter
    // has a default value
    auto addend_1_accessor = addend_1_buf.template get_access<dp_read>(h);
    auto addend_2_accessor = addend_2_buf.template get_access<dp_read>(h);

    // Note: Can use access::mode::discard_write instead of access::mode::write
    // because we're replacing the contents of the entire buffer.
    auto sum_accessor = sum_buf.template get_access<dp_write>(h);

    // Use parallel_for to run array addition in parallel. This executes the
    // kernel. The first parameter is the number of work items to use and the
    // second is the kernel, a lambda that specifies what to do per work item.
    // The template parameter ArrayAdd is used to name the kernel at runtime.
    // The parameter passed to the lambda is the work item id of the current
    // item.
    //
    // To remove the requirement to specify the kernel name you can enable
    // unnamed lamdba kernels with the option:
    //     dpcpp -fsycl-unnamed-lambda
    h.parallel_for<class ArrayAdd>(num_items, [=](id<1> i) {
        sum_accessor[i] = addend_1_accessor[i] + addend_2_accessor[i];
    });
});
```

Lancaster University

Example Code

```
// call wait_and_throw to catch async exception
q->wait_and_throw();

// DPC++ will enqueue and run the kernel. Recall that the buffer's data is
// given back to the host at the end of the method's scope.
```