Using High-Level C++ for HEP Data Processing on Accelerators

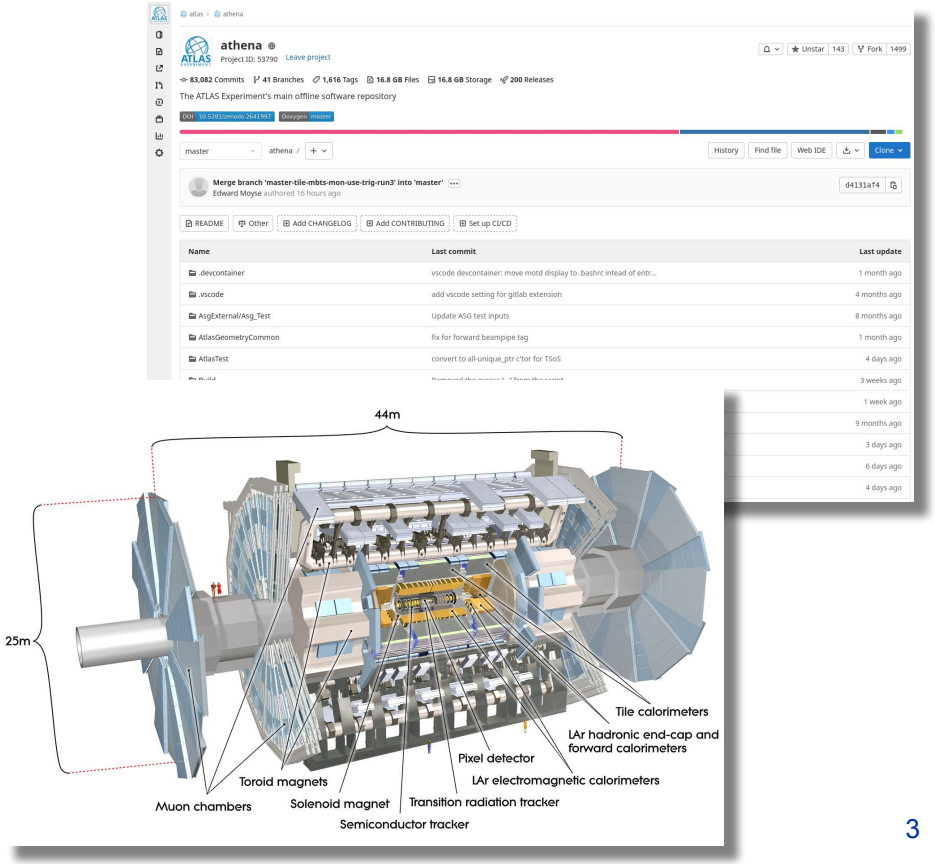
Attila Krasznahorkay

Outline

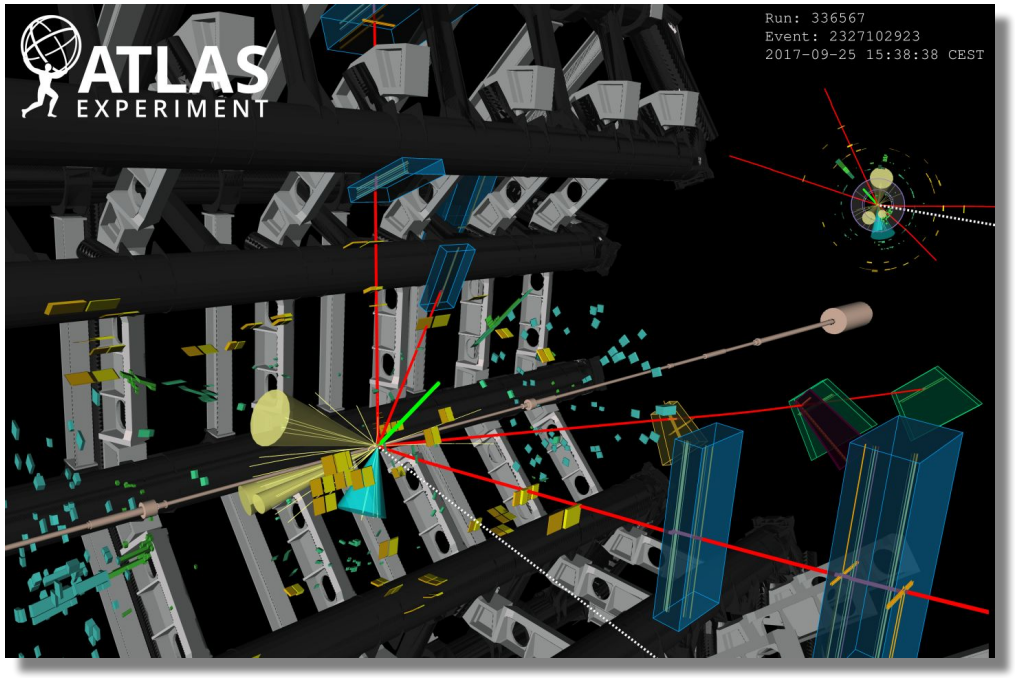
- Data processing software at the LHC
  - And why it needs to worry about accelerators
- An overview of current accelerators, and their programming languages
  - With some information on how we are using / planning to use these features
  - Putting special emphasis on memory management techniques with modern C++
- An insight in the kind of software R&D happening in HEP and in ATLAS at the moment

ATLAS And Its Offline Software

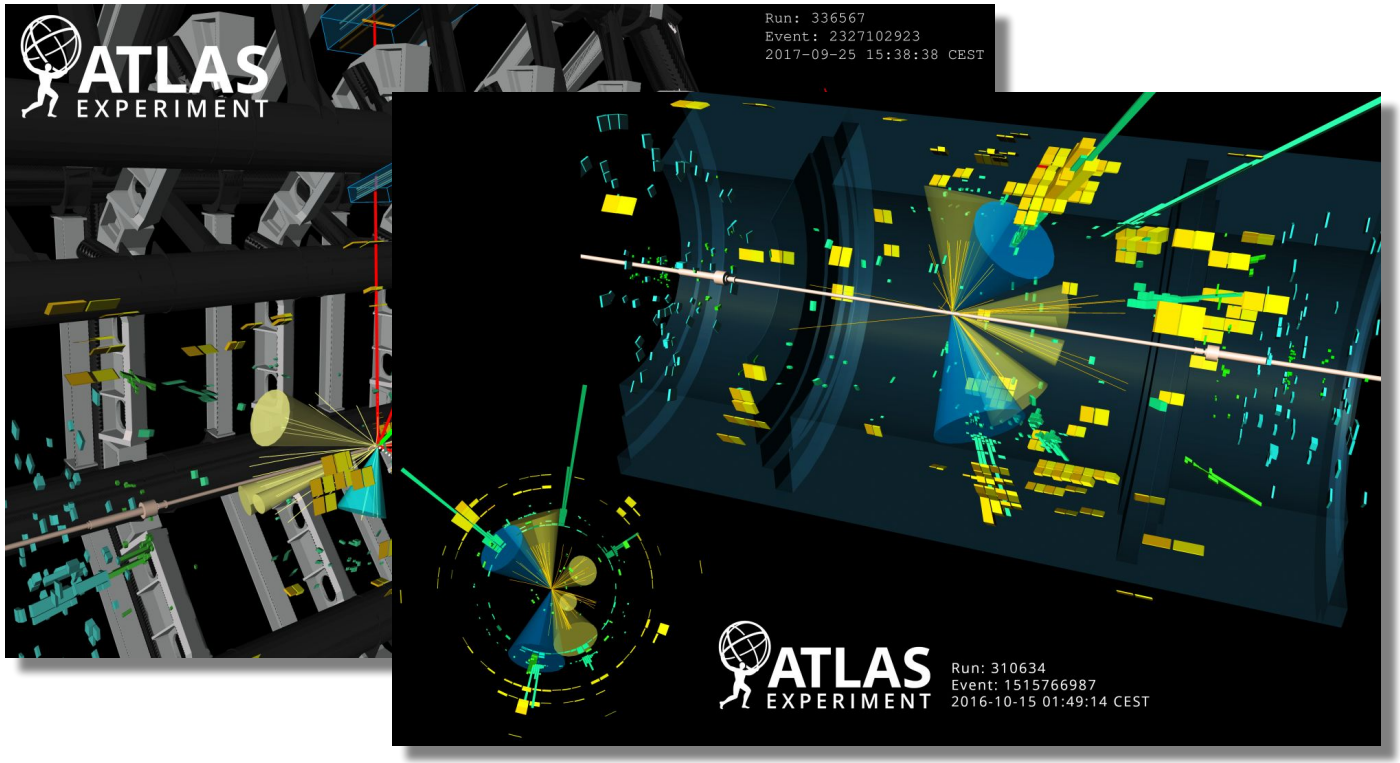
- ATLAS is one of the general-purpose experiments at the Large Hadron Collider
  - Built/operated by the largest collaboration for any physics experiment ever
- The software (atlas/athena, atlassoftwaredocs) written for processing its data is equally large
  - ~4 million lines of C++ and ~2 million lines of Python



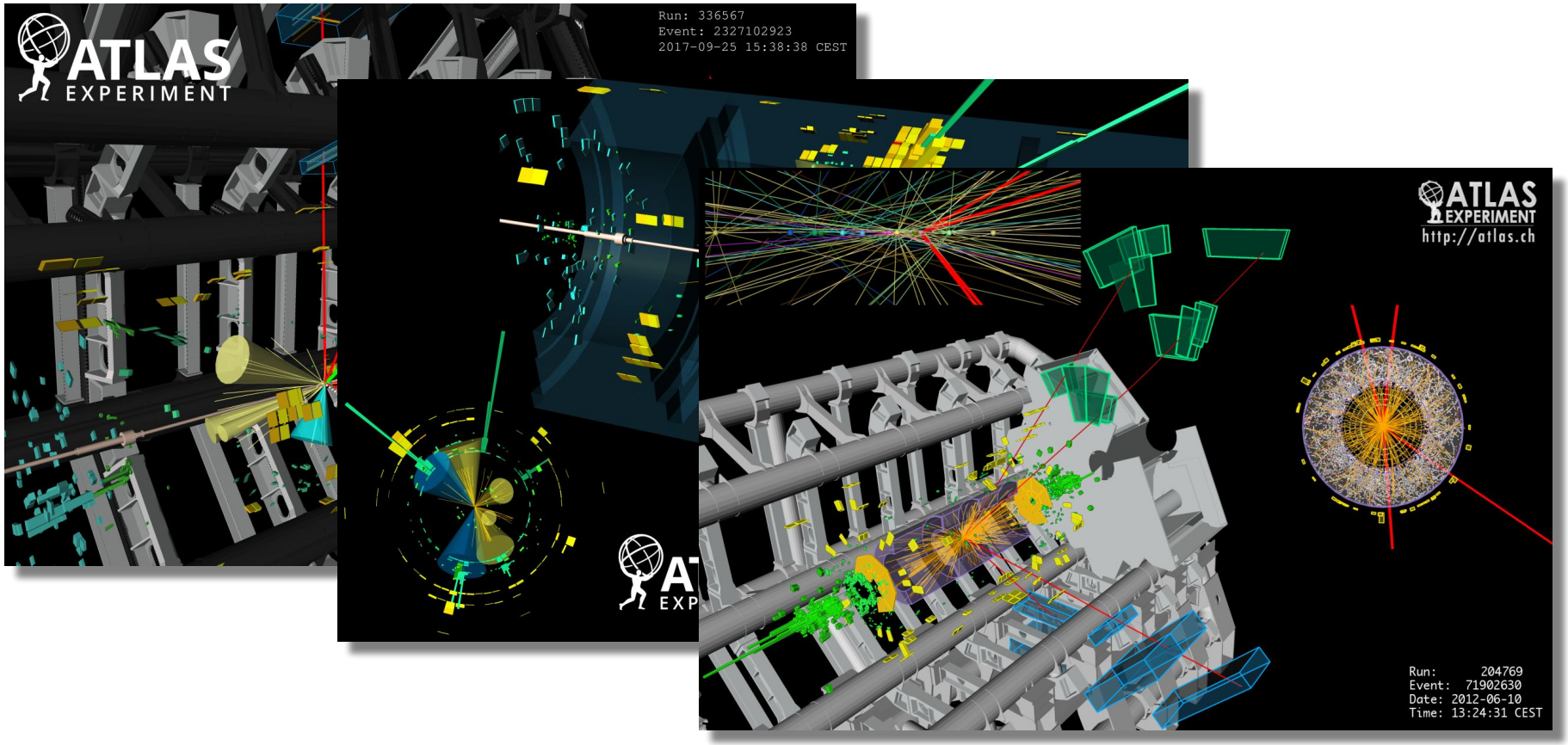
Data Reconstruction in ATLAS



Data Reconstruction in ATLAS



Data Reconstruction in ATLAS

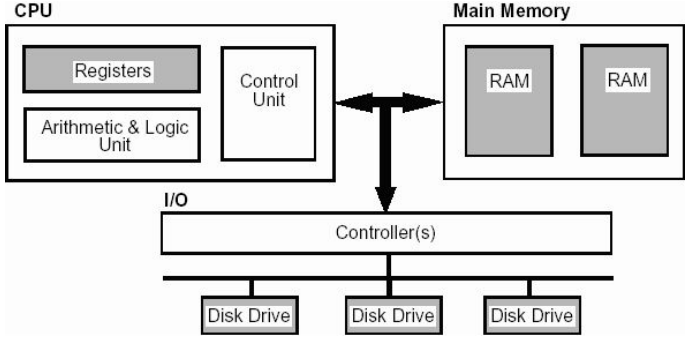


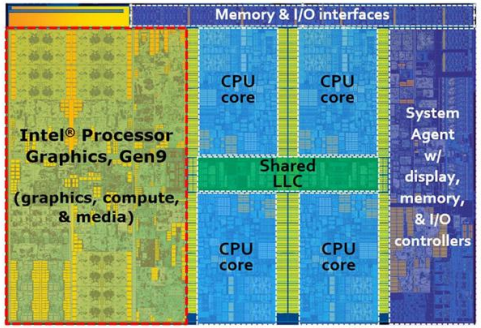
Why Accelerators?

(High Performance) Computing in 2021

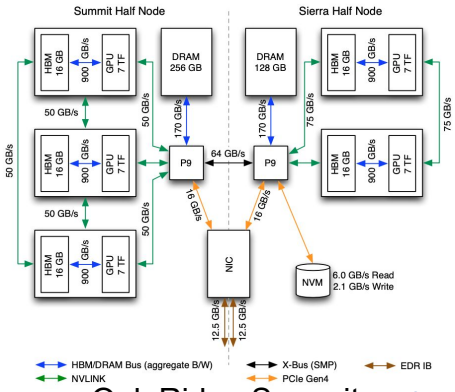
- Computing has been getting more and more complicated in the last decades
  - A modern CPU has a very complicated design, mainly to make sure that (our!) imperfect programs would execute fast on it
- Complexity shows up both “inside of single computers”, but also in the structure of computing clusters
  - A modern computing cluster has different nodes connected to each other in a non-trivial network
- All the added complexity is there to achieve the highest possible theoretical throughput “for certain calculations” on these machines

“Classical” computer architecture





Intel® Skylake™

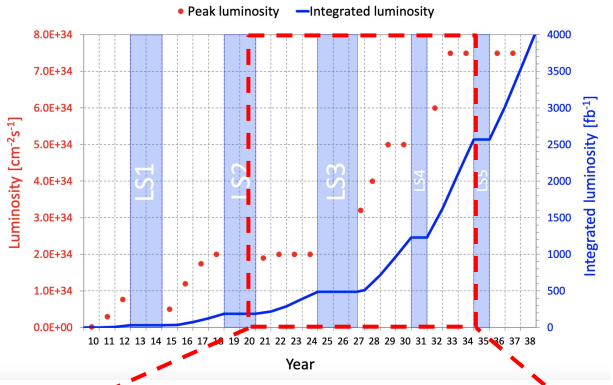


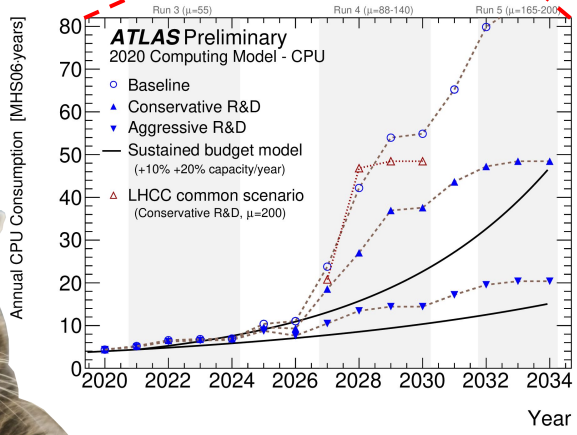
Oak Ridge Summit

(High Performance) Computing in 2021

- Supercomputers **all** use accelerators
- Which come in many shapes and sizes
  - NVidia GPUs are the most readily available in general, used/will be in Summit, Perlmutter, LEONARDO and MeluXina
  - AMD GPUs are not used too widely in comparison, but will be in Frontier, El Capitan and LUMI
  - Intel GPUs are used even less at the moment, but will get center stage in Aurora
  - FPGAs are getting more and more attention, and if anything, they are even more tricky to write (good) code for
- Beside HPCs, commercial cloud providers also offer an increasingly heterogeneous infrastructure

Why HEP/ATLAS Cares About Accelerators

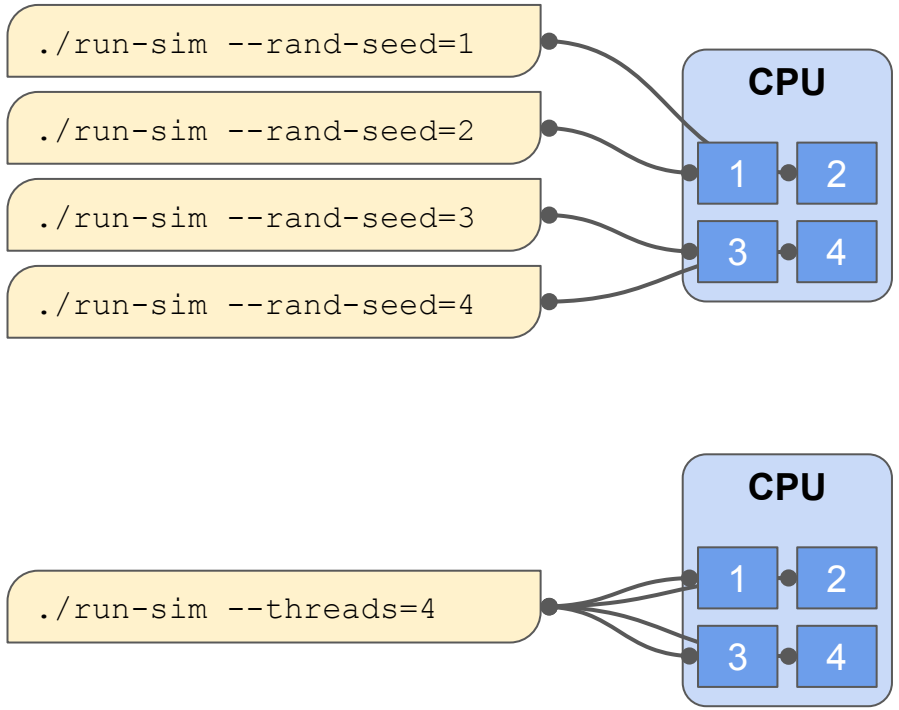




- As described in CERN-LHCC-2020-015, being able to process the data collected in LHC Run 4 (and beyond) in ATLAS requires major software developments
  - In order to fit into our “CPU budget”, we need to consider new approaches in our data processing
- One of these areas is to look at non-CPU resources

Multiprocessing, Multithreading

- “Simple” applications are almost always single threaded
  - This is what you get by default out of most programming languages. A single execution thread performing tasks one by one.
- Luckily many tasks in HEP are embarrassingly parallel
  - We can just start N instances of the application, all doing different things.
- Usually (at least in HEP) when memory usage becomes an issue, the application needs to become multi-threaded
  - Where a single process executes calculations on multiple threads in parallel.



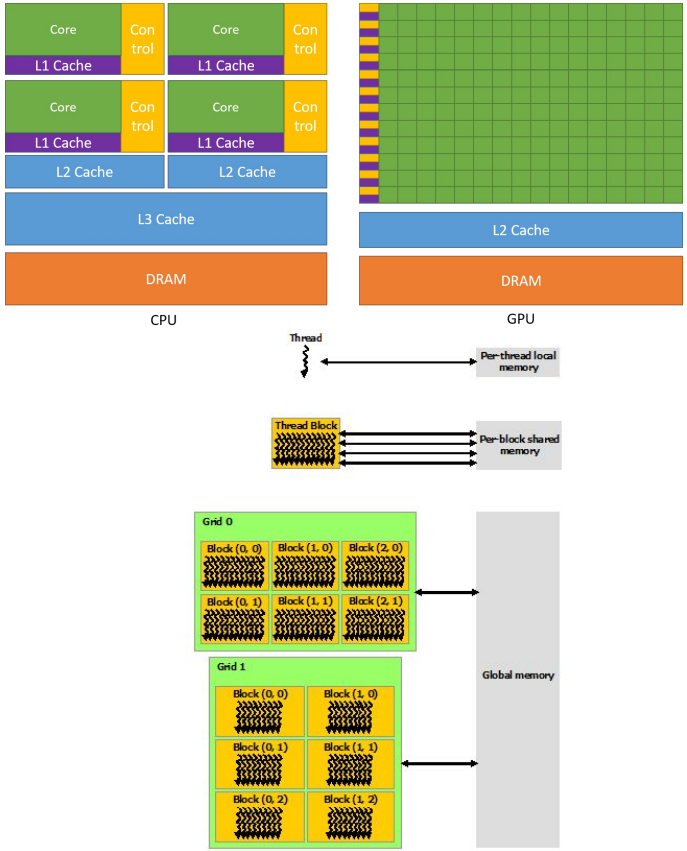
HEP Software

- Most (but not absolutely all) HEP software is written in C++ these days
  - We even agreed on a single platform (Threading Building Blocks) for our multithreading
- LHC experiments, mostly driven by their (our…) memory hungry applications, are all migrating to multithreaded workflows by now
  - ATLAS will use a multithreaded framework for triggering and reconstructing its data during LHC Run-3
  - However smaller HEP/NP experiments are still happily using multiprocessing to parallelise their data processing
- It is in this context that we are looking towards upgrading our software to use non-x86 computing as well

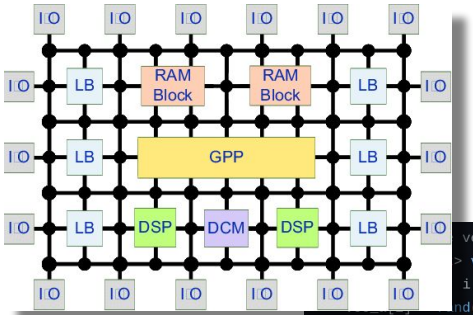
What Accelerators?

GPGPUs

- General Purpose GPUs (GPGPUs) are the “most common” accelerators
- They can achieve **very** high theoretical FLOPs because they have a **lot** of units for performing floating point calculations
- But unlike CPUs, these cores are not independent of each other
  - Control units exist for large groups of computing cores, forcing the cores to all do the same thing at any given time
  - Memory caching is implemented in a much simpler way for these computing cores than for CPUs
- Coming even close to the theoretical limits of accelerators is only possible with purpose designed algorithms



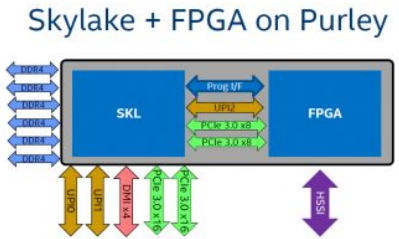
FPGAs / ASICs




```
vectors and fill two with random values.
> vec_a(kSize), vec_b(kSize), vec_r(kSize);
  i < kSize; i++) {
nd();
  vec_b[i] = rand();
}

// Select either:
//   - the FPGA emulator device (CPU emulation of the FPGA)
//   - the FPGA device (a real FPGA)
#if defined(FPGA_EMULATOR)
  ext::intel::fpga_emulator_selector device_selector;
#else
  ext::intel::fpga_selector device_selector;
#endif

  try {

    // Create a queue bound to the chosen device.
    // If the device is unavailable, a SYCL runtime exception is thrown.
    queue q(device_selector, dpc_common::exception_handler);

    // Print out the device information.
    std::cout << "Running on device: "
              << q.get_device().get_info<info::device::name>() << "\n";
```

- Will become important as well, but at the moment are a bit less important with “generic algorithms”
  - They are normally suited better for well-defined/understood data processing steps. For instance decoding data coming from the detector. 😉
- The software projects to know about with these are Intel’s oneAPI and various High Level Synthesis (HLS) implementations

The Future of CPUs/GPUs (?)



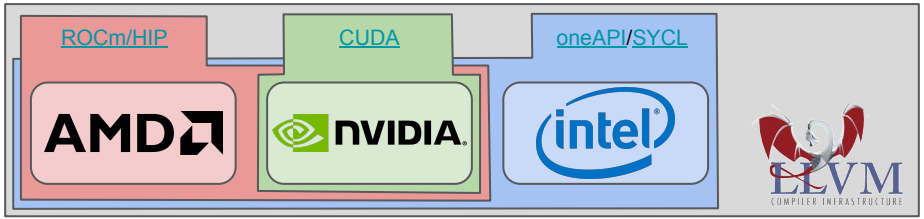
Cores	Up to 28C with Intel® HT Technology	
FPGA	Altera® Arria 10 GX 1150	
Socket TDP	Shared socket TDP Up to 165W SKL & Up to 90W FPGA	
Socket	Socket P	
Scalability	Up to 2S – with SKL-SP or SKL + FPGA SKUs	
PCH	Lewisburg: DMI3 – 4 lanes; 14xUSB2 ports Up to: 10xUSB3; 14xSATA3, 20xPCIe*3 Newc Innovation Engine, 4x10GbE ports, Intel® QuickAssist Technology	
	For CPU	**For FPGA**
Memory	6 channels DDR4 RDIMM, LRDIMM, 2666 1DPC, 2133, 2400 2DPC	Low latency access to system memory via UPI & PCIe interconnect
Intel® UPI	2 channels (10.4, 9.6 GT/s)	1 channel (9.6 GT/s)
PCIe*	PCIe* 3.0 (8.0, 5.0, 2.5 GT/s) 32 lanes per CPU Bifurcation support: x16, x8, x4	PCIe* 3.0 (8.0, 5.0, 2.5 GT/s) 16 lanes per FPGA Bifurcation support: x8
High Speed Serial Interface (Different board design based on HSSI config)	N/A	2xPCIe 3.0 x8 Direct Ethernet (4x10 GbE, 2x40 GbE, 10x10 GbE, 2x25 GbE)

- Is quite uncertain…
  - These days even the future of x86 seems to be in some jeopardy 🤔
- Heterogeneous seems to be the key
  - Some CPUs already have different cores, meant for different tasks
  - CPU+GPU combinations will likely become more and more popular in HPCs
    - Making it possible to manage the memory of applications more easily
  - GPUs are not even the only game in town
    - “FPGA inserts” may become a part of future high-performance CPUs/GPUs…

(Current) Programming Languages

C++...?

- Just as with “CPU languages”, there is no single language for writing accelerator code with
  - But while HEP settled on C++ for CPUs, at this point the whole community just can’t settle on a single language for accelerators yet
- However most of these languages are at least C/C++ based
  - But unfortunately each of them have (slightly) different capabilities



- Multiple projects exist / are actively developed for hiding this complexity from the programmers (Kokkos, Alpaka, Thrust, Parallel STL, etc.)
- Eventually the goal is to make heterogeneous programming part of the ISO C++ standard
  - I will try to show the most interesting/important fronts on which this is happening

CUDA

- NVidia/CUDA is the most established player in this game
  - As such they have the most support in existing applications, the best documentation, etc.
- Originally designed as a C language/library
  - Over the years getting more and more C++ support
  - By now supporting even some C++17 features in “device code”, including some “light amount” of virtualisation
- Practically only supported on NVidia hardware


```
/// Very simple kernel performing a multiplication on an array.
__global__
void cudaMultiplyKernel( int n, float* array, float multiplier ) {

   const int index = blockIdx.x * blockDim.x + threadIdx.x;
   if( index >= n ) {
      return;
   }

   array[ index ] *= multiplier;
   return;
}

/// GPU implementation of @c cudaMultiply
void cudaMultiply( std::vector< float >& array, float multiplier ) {

   // If no CUDA device is available, complain.
   int nCudaDevices = 0;
   CUDA_CHECK( cudaGetDeviceCount( &nCudaDevices ) );
   if( nCudaDevices == 0 ) {
      return;
   }

   // Allocate the array on the/a device, and copy the host array's content
   // to the device.
   float* deviceArray = nullptr;
   CUDA_CHECK( cudaMalloc( &deviceArray, sizeof( float ) * array.size() ) );
   CUDA_CHECK( cudaMemcpy( deviceArray, array.data(),
                           sizeof( float ) * array.size(),
                           cudaMemcpyHostToDevice ) );

   // Run the kernel.
   static const int blockSize = 256;
   const int numBlocks = ( array.size() + blockSize - 1 ) / blockSize;
   cudaMultiplyKernel<<< numBlocks, blockSize >>>( array.size(),
                                                   deviceArray,
                                                   multiplier );
   CUDA_CHECK( cudaDeviceSynchronize() );

   // Copy the array back to the host's memory.
   CUDA_CHECK( cudaMemcpy( array.data(), deviceArray,
                           sizeof( float ) * array.size(),
                           cudaMemcpyDeviceToHost ) );

   // Free the memory on the device.
   CUDA_CHECK( cudaFree( deviceArray ) );
   return;
}
```

ROCm / HIP

```
namespace {
   /// Linear transformation kernel
   __global__
   void hipLinearTransform( std::size_t size, float* data, float a, float b ) {

      // Get the current index.
      const std::size_t index = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x;
      if( index >= size ) {
         return;
      }

      // Perform the linear transformation.
      data[ index ] = a * data[ index ] + b;
      return;
   }
} // private namespace
```

```
void linearTransform( int deviceId, std::vector< float >& data, float a,
                      float b ) {

   // Select the requested device.
   HIP_CHECK( hipSetDevice( deviceId ) );

   // Allocate memory on the device for this array.
   float* deviceData = nullptr;
   const std::size_t dataSize = data.size() * sizeof( float );
   HIP_CHECK( hipMalloc( &deviceData, dataSize ) );

   // Copy the contents of the dummy array to the device.
   HIP_CHECK( hipMemcpy( deviceData, data.data(), dataSize,
                         hipMemcpyHostToDevice ) );

   // Launch the linear transformation kernel.
   static constexpr int blockSize = 256;
   const int numBlocks = ( data.size() + blockSize - 1 ) / blockSize;
   static constexpr int sharedMem = 0;
   static constexpr hipStream_t stream = nullptr;
   hipLaunchKernelGGL( hipLinearTransform, numBlocks, blockSize, sharedMem,
                       stream, data.size(), deviceData, a, b );
   HIP_CHECK( hipGetLastError() );
   HIP_CHECK( hipDeviceSynchronize() );

   // Copy the memory back from the device.
   HIP_CHECK( hipMemcpy( data.data(), deviceData, dataSize,
                         hipMemcpyDeviceToHost ) );
   // Free the memory on the device.
   HIP_CHECK( hipFree( deviceData ) );

   return;
}
```

- Is basically a copy-paste of CUDA
  - The concepts are all the same
  - CUDA functions exist in 99% in HIP, with a slightly different name
- Support/documentation is far inferior to that of CUDA
- Code written in HIP is relatively easy to compile for both AMD and NVidia backends
  - When compiling for an NVidia backend, the HIP headers basically include the CUDA backends, and declare a lot of typedefs…

oneAPI / SYCL

- Intel's answer to the programming language question
- Unlike CUDA, does not require an extension to the C++ language
  - Which means that it's possible to provide support for SYCL code using "a library" with any compiler
    - As long as GPU support is not required
- Very strong design-wise, built on top of the latest C++ capabilities
- Technically it's possible to compile SYCL code for Intel (CPU, GPU, FPGA), NVidia and AMD backends
  - However the AMD backend's support is a bit flakier than the others

oneAPI

```
    // Create a vector array that would be manipulated.
    std::vector< float > dummyArray;
    static const std::size_t ARRAY_SIZE = 10000;
    dummyArray.reserve( ARRAY_SIZE );
    static const float ARRAY_ELEMENT = 3.141592f;
    for( std::size_t i = 0; i < ARRAY_SIZE; ++i ) {
       dummyArray.push_back( ARRAY_ELEMENT );
    }

    // Set up a SYCL buffer on top of this STL object.
    cl::sycl::buffer< cl::sycl::cl_float > buffer( dummyArray.begin(),
                                                   dummyArray.end() );

    // Set up the SYCL queue.
    cl::sycl::queue queue( m_deviceSelector );
    cl::sycl::range< 1 > workItems( buffer.get_count() );

#ifndef TRISYCL_CL_SYCL_HPP
    // Let the user know what device the calculation is running on.
    const cl::sycl::device& device = queue.get_device();
    ATH_MSG_DEBUG( "Using device "
                   << device.get_info< cl::sycl::info::device::name >()
                   << " ("
                   << device.get_info< cl::sycl::info::device::version >()
                   << ")" );
#endif // not TRISYCL_CL_SYCL_HPP

    // Multiply these values using SYCL.
    static const float MULTIPLIER = 1.23f;
    queue.submit( [&]( cl::sycl::handler& handler ) {
          auto acc =
             buffer.get_access< cl::sycl::access::mode::read_write >( handler );
          handler.parallel_for< class SYCLMultiply >( workItems,
             [=]( cl::sycl::id< 1 > id ) {
                acc[ id ] *= MULTIPLIER;
             } );
       } );
```

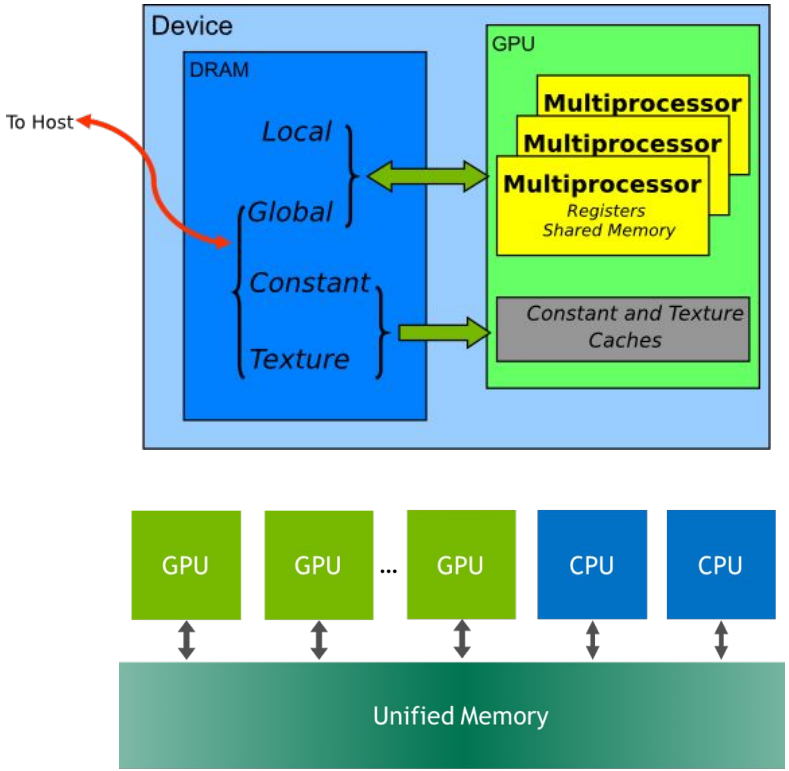
C++ / Host Code

- One of the first idea from everybody who starts working on this type of code is to make it possible to run the exact same code on accelerators and on the host
  - And for a good number of calculations this can be a good idea, especially for making certain parts of debugging a little easier
- However many algorithms in HEP do not factorise well like this
  - Any “combinatorial” code usually has to be implemented with a different logic for CPUs (where you want to minimise FLOPs with conditionals) and GPUs (where you want to minimise conditionals, while not caring about FLOPs all that much)
  - Because of this, even when using oneAPI/SYCL, we still implement separate algorithms for CPUs and GPUs for most things

Memory Management (in C++)

Memory Management

- Modern CPUs have a very complicated memory management system
  - Which we can in most cases avoid knowing about
- GPUs have a complicated system of their own
  - However this we can not avoid knowing more about to use GPUs efficiently ☹
  - Most importantly, caching is much less automated than on modern CPUs
- In some cases however you can get away with not knowing everything
  - For a performance penalty...



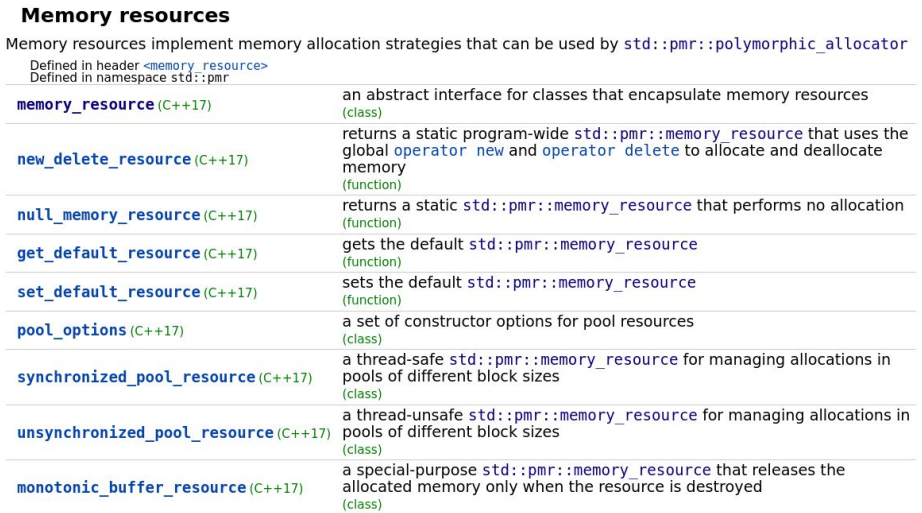
Memory (De-)Allocation

```
const std::vector< float >& input = …;

float *a = nullptr, *b = nullptr;
CUDA_CHECK( cudaMallocManaged( &a,
              input.size() * sizeof( float ) ) );
CUDA_CHECK( cudaMallocManaged( &b,
              input.size() * sizeof( float ) ) );

for( std::size_t i = 0; i < input.size(); ++i ) {
   a[ i ] = input[ i ];
}

linearTransform<<< 1, input.size() >>>( a, b, … );
CUDA_CHECK( cudaGetLastError() );
CUDA_CHECK( cudaDeviceSynchornize() );

std::vector< float > output;
output.reserve( input.size() );
for( std::size_t i = 0; i < input.size(); ++i ) {
   output.push_back( b[ i ] );
}

CUDA_CHECK( cudaFree( a ) );
CUDA_CHECK( cudaFree( b ) );
```

- CUDA started by providing C-style memory allocation/deallocation functions
  - Eventually every other language followed this design as well
- Allows for a precise management of the memory resources
- But it is in stark contrast with modern C++ design guidelines
  - Modern C++ code should not even have new/delete statements in it, let alone malloc(...)/free(...)

C++17 Dynamic Memory Management

- STL-friendly “adapter code” has been developed for a long time for this, using custom “container allocators”
- One important development came from NVidia, with Thrust
  - This was generalised to be part of C++17 as the “memory resource infrastructure”
- Which is something that we have been very actively using in the VecMem project



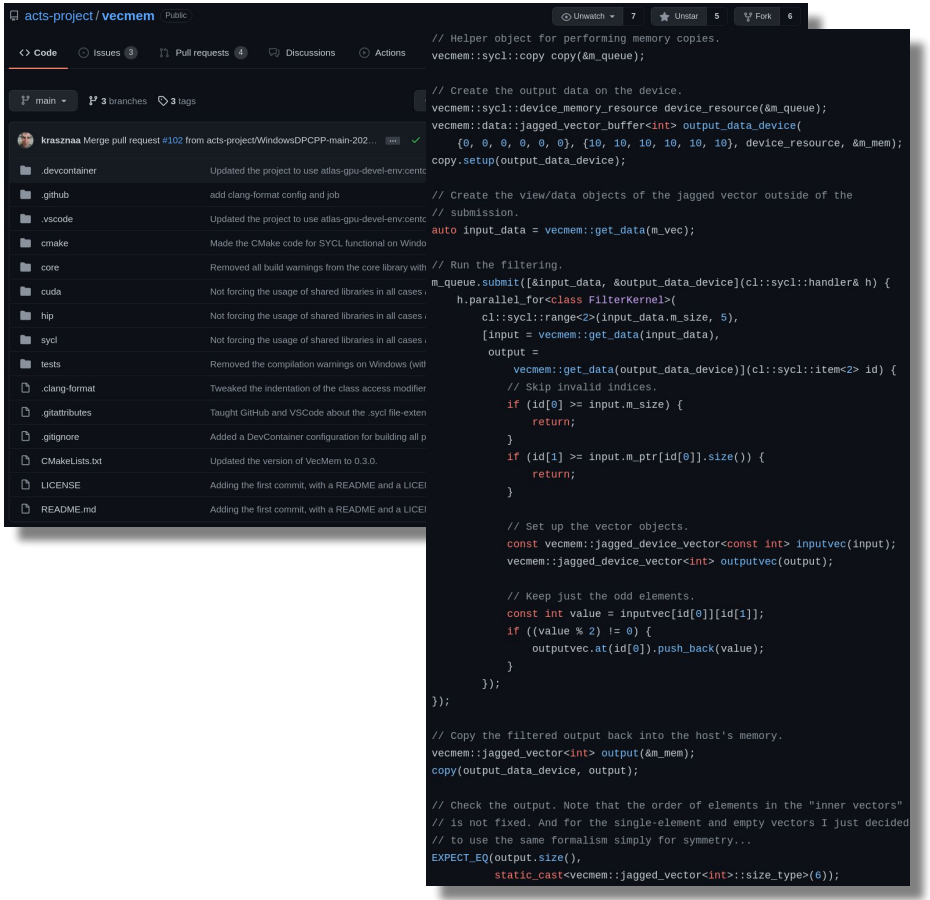
Memory resources

Memory resources implement memory allocation strategies that can be used by `std::pmr::polymorphic_allocator`

Defined in header `<memory_resource>`
Defined in namespace `std::pmr`

`memory_resource` (C++17)	an abstract interface for classes that encapsulate memory resources (class)
`new_delete_resource` (C++17)	returns a static program-wide `std::pmr::memory_resource` that uses the global `operator new` and `operator delete` to allocate and deallocate memory (function)
`null_memory_resource` (C++17)	returns a static `std::pmr::memory_resource` that performs no allocation (function)
`get_default_resource` (C++17)	gets the default `std::pmr::memory_resource` (function)
`set_default_resource` (C++17)	sets the default `std::pmr::memory_resource` (function)
`pool_options` (C++17)	a set of constructor options for pool resources (class)
`synchronized_pool_resource` (C++17)	a thread-safe `std::pmr::memory_resource` for managing allocations in pools of different block sizes (class)
`unsynchronized_pool_resource` (C++17)	a thread-unsafe `std::pmr::memory_resource` for managing allocations in pools of different block sizes (class)
`monotonic_buffer_resource` (C++17)	a special-purpose `std::pmr::memory_resource` that releases the allocated memory only when the resource is destroyed (class)

VecMem




```
// Helper object for performing memory copies.
vecmem::sycl::copy copy(&m_queue);

// Create the output data on the device.
vecmem::sycl::device_memory_resource device_resource(&m_queue);
vecmem::data::jagged_vector_buffer<int> output_data_device(
    {0, 0, 0, 0, 0, 0}, {10, 10, 10, 10, 10, 10}, device_resource, &m_mem);
copy.setup(output_data_device);

// Create the view/data objects of the jagged vector outside of the
// submission.
auto input_data = vecmem::get_data(m_vec);

// Run the filtering.
m_queue.submit([&input_data, &output_data_device](cl::sycl::handler& h) {
    h.parallel_for<class FilterKernel>(
        cl::sycl::range<2>(input_data.m_size, 5),
        [input = vecmem::get_data(input_data),
         output =
             vecmem::get_data(output_data_device)](cl::sycl::item<2> id) {
            // Skip invalid indices.
            if (id[0] >= input.m_size) {
                return;
            }
            if (id[1] >= input.m_ptr[id[0]].size()) {
                return;
            }

            // Set up the vector objects.
            const vecmem::jagged_device_vector<const int> inputvec(input);
            vecmem::jagged_device_vector<int> outputvec(output);

            // Keep just the odd elements.
            const int value = inputvec[id[0]][id[1]];
            if ((value % 2) != 0) {
                outputvec.at(id[0]).push_back(value);
            }
        });
});

// Copy the filtered output back into the host's memory.
vecmem::jagged_vector<int> output(&m_mem);
copy(output_data_device, output);

// Check the output. Note that the order of elements in the "inner vectors"
// is not fixed. And for the single-element and empty vectors I just decided
// to use the same formalism simply for symmetry...
EXPECT_EQ(output.size(),
          static_cast<vecmem::jagged_vector<int>::size_type>(6));
```

- As part of a larger effort in the Acts community, we are developing a library that could help with using containers of “simple” data in heterogeneous code
  - It provides a set of classes for use in host and device code, for simplifying common container access patterns
- Dedicated presentations about this project will be shown at future conferences/workshops

Atomic Memory Operations

- Many multi-threaded / GPU algorithms make use of atomic variables
  - GPU hardware allows for atomic updates to any variable in “global memory”. Which is unfortunately not possible to express with the current C++ std::atomic interface.
  - Projects like VecMem, and (very importantly!) Kokkos, had to work around this using their own atomic types.
- One important new feature in C++20 is std::atomic_ref, pushed into the standard by the Kokkos developers 🥳
  - It provides an interface that is finally appropriate for “device code” as well
  - Future versions of CUDA/HIP/SYCL shall be able to understand this type in “device code”, making code sharing between different platforms even easier

Offloaded Code Execution

Formalism

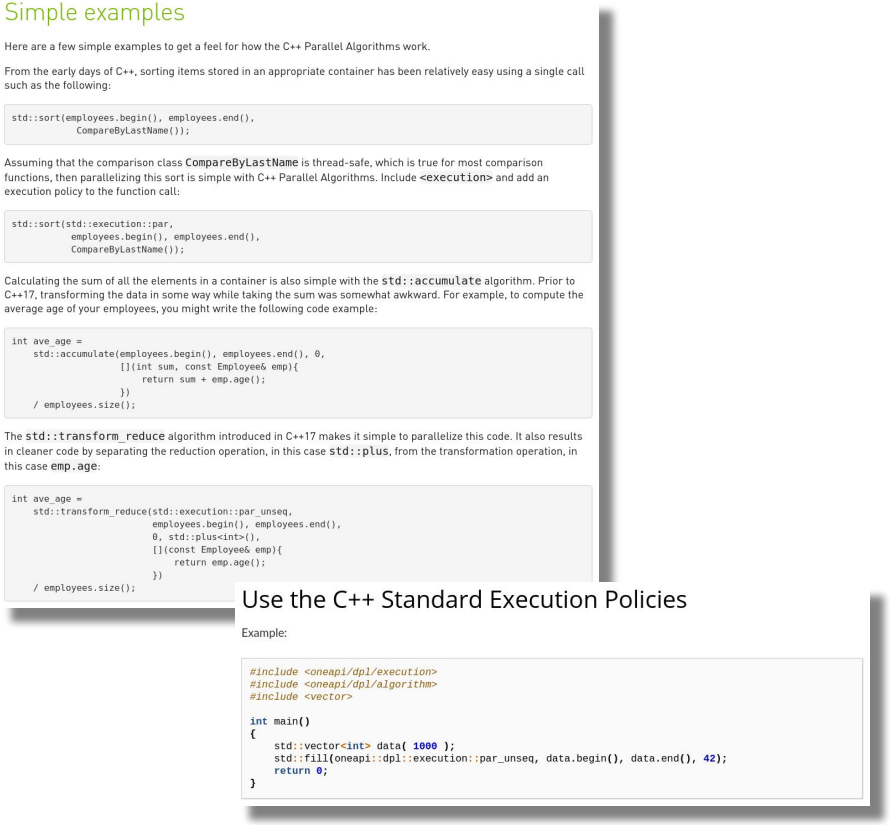
- CUDA, HIP and SYCL each have their own formalism for executing a “function” on many parallel threads
  - They all need to allow a detailed specification of how to launch the function on the hardware
- Since the concept is quite the same in all cases, a number of projects were written to create uniform interfaces on top of them
  - But while this can be very useful in some situations, having to launch a GPU kernel in slightly different ways in the different languages is rarely the difficult part in porting some code...

```
dim3 numBlocks(…), blockSize(…);
size_t sharedMem = …;
cudaStream_t stream = …;
mySuperKernel<<< numBlocks, blockSize,
                 sharedMem, stream >>>(…);
```

```
dim3 numBlocks(…), blockSize(…);
uint32_t sharedMem = …;
hipStream_t stream = …;
hipLaunchKernelGGL( mySuperKernel, numBlocks,
   blockSize, sharedMem, stream, … );
```

```
sycl::nd_range<3> range;
sycl::queue& q = …;
q.submit( [&]( sycl::handler& h ) {
  sycl::accessor<…,
    sycl::access::target::local> a(…);
  h.parallel_for( range,
                  [=]( sycl::nd_item<3> id ) {
                    …;
                  } );
} );
```

C++17 Parallel STL Algorithms



Simple examples

Here are a few simple examples to get a feel for how the C++ Parallel Algorithms work.

From the early days of C++, sorting items stored in an appropriate container has been relatively easy using a single call such as the following:

```
std::sort(employees.begin(), employees.end(),
          CompareByLastName());
```

Assuming that the comparison class `CompareByLastName` is thread-safe, which is true for most comparison functions, then parallelizing this sort is simple with C++ Parallel Algorithms. Include `<execution>` and add an execution policy to the function call:

```
std::sort(std::execution::par,
          employees.begin(), employees.end(),
          CompareByLastName());
```

Calculating the sum of all the elements in a container is also simple with the `std::accumulate` algorithm. Prior to C++17, transforming the data in some way while taking the sum was somewhat awkward. For example, to compute the average age of your employees, you might write the following code example:

```
int ave_age =
    std::accumulate(employees.begin(), employees.end(), 0,
                    [](int sum, const Employee& emp){
                        return sum + emp.age();
                    })
    / employees.size();
```

The `std::transform_reduce` algorithm introduced in C++17 makes it simple to parallelize this code. It also results in cleaner code by separating the reduction operation, in this case `std::plus`, from the transformation operation, in this case `emp.age`:

```
int ave_age =
    std::transform_reduce(std::execution::par_unseq,
                          employees.begin(), employees.end(),
                          0, std::plus<int>(),
                          [](const Employee& emp){
                              return emp.age();
                          })
    / employees.size();
```

Use the C++ Standard Execution Policies

Example:

```
#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>
#include <vector>

int main()
{
    std::vector<int> data( 1000 );
    std::fill(oneapi::dpl::execution::par_unseq, data.begin(), data.end(), 42);
    return 0;
}
```

- Purely numeric calculations can even be expressed without writing any accelerator code directly
  - If your calculation can be expressed purely through STL algorithms, it is likely that it can be executed on an accelerator as well
- It very much relies on compiler support
  - Even more, while the NVidia HPC SDK allows you to run “more-or-less-standard” C++17 code on your GPU, Intel oneAPI requires you to use some Intel specific includes…
- Still, it is one of the most platform independent ways of writing accelerated code at the moment

C++23(?) Executors

```
executor auto ex = ...;
execute(ex, []{
            cout << "Hello, executors!\n"; });
```

- P0443R14 proposes a unified interface for launching tasks on “some backend”
  - With a formalism a little reminiscent of SYCL
- The goal is of course to introduce a formalism that could describe CPU and accelerator multi-threading using a single interface
  - Allowing hardware makers to process code (with their own compilers, at least initially) that could look practically the same for all types of accelerators

Code Sharing

- Until the “device code launch” formalism is standardized, we can still organise our code in clever ways
  - As much code as possible should be delegated into “standard” functions, which kernels can call on to perform some task/calculation
  - This mainly requires a unified handling of memory in my opinion, which can already be done in clever ways
- We are currently experimenting with exactly how far we can take this, in [acts-project/traccc]

```
DEVICE_FUNCTION
float calculateSomething( const vecmem::device_vector<const float>& vec,
                          std::size_t index );

__global__
void cudaKernel( vecmem::vector_view<const float> vec_view, … ) {
   const std::size_t i = blockIdx.x * blockDim.x + threadIdx.x;
   vecmem::device_vector<const float> vec( vec_view );
   float foo = calculateSomething( vec, i );
   …
}

__global__
void hipKernel( vecmem::vector_view<const float> vec_view, … ) {
   const std::size_t i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x;
   vecmem::device_vector<const float> vec( vec_view );
   float foo = calculateSomething( vec, i );
   …
}

class SyclKernel {
public:
   SyclKernel( vecmem_vector_view<const float> vec_view )
      : m_vec_view( vec_view ), … {}
   void operator()( sycl::id<1> id ) {
      vecmem::device_vector<const float> vec( m_vec_view );
      float foo = calculateSomething( vec, id );
      …
   }
private:
   vecmem::vector_view<const float> m_vec_view;
};
```

Developments in ATLAS

Heterogeneous Computing and Accelerators Forum

Mandate for the Heterogeneous Computing and Accelerators Forum

(Updated on 14.1.2021)

Mandate:

The future of computing hardware is uncertain, but one global trend is towards heterogeneous resources and more specifically towards "accelerators": specialized (non-CPU) hardware that enhances performance for certain computations. One of the most obvious examples is the Graphics Processing Unit (GPU), which is adept at highly parallel, low-accuracy computations. Other popular examples include FPGAs and TPUs.

Within ATLAS, discussion and overall planning of work on heterogeneous resources should be within the Heterogeneous Computing and Accelerators Forum (HCAF) which includes efforts from both offline software and TDAQ. The conveners of the forum should maintain a list of high-level milestones towards the adoption of the technologies targeted by development within ATLAS.

The forum should meet at least once a month.

Reporting and Liaisons:

The HCAF conveners report to the ATLAS Computing Coordinator and the TDAQ Project, TDAQ Upgrade Project, and Upgrade Project Leaders. They may appoint liaisons or contacts as needed. They should ensure ATLAS is represented in collaborative forums focused on accelerators, like the HSF accelerators forum.

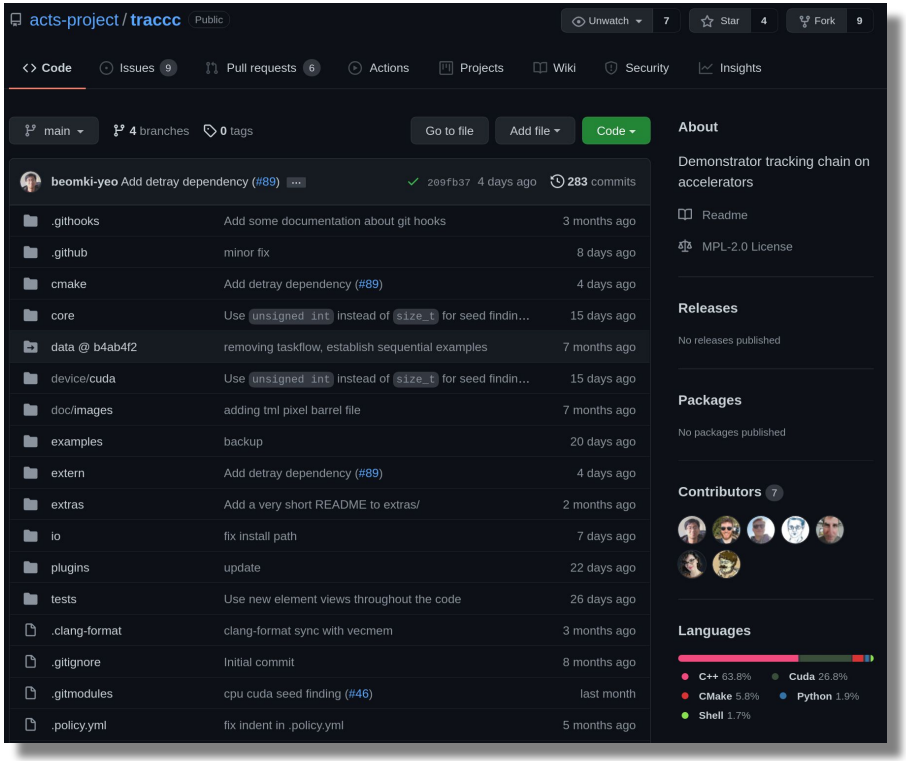
Term of Office:

The HCAF conveners are appointed by the ATLAS Computing Coordinator and TDAQ Upgrade Project Leader with a renewable one year term normally starting October 1st. At least two conveners are appointed. Between them, responsibilities are split; however, knowledge should be shared such that they can represent each other in case one is unavailable.

- To organise/oversee the developments in this area, the Heterogeneous Computing and Accelerators Forum (HCAF) was formed
  - Built on top of the previous separate groups overseeing the offline and TDAQ efforts in this area
- It is in this group that we try to organise all of these types of developments...

Current Studies/Developments

- R&D is happening in many areas of the ATLAS offline software
  - (Charged) track reconstruction
  - Calorimetry
  - Core Software
- Probably the most “public” development at the moment is happening in [acts-project/traccc]
  - Where we intend to demonstrate a “realistic setup” for performing charged track reconstruction on accelerators



Summary

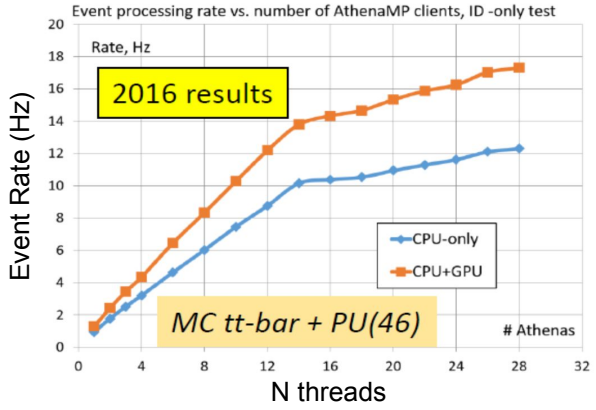
- After a calm period of homogeneous x86 computing, HEP will once again have to use a wide variety of heterogeneous hardware for a while
  - I believe there is a periodicity to this. Current accelerator technologies will inevitably become more homogeneous after a while.
- C++ will stay the “main” programming language of HEP for a long time to come
  - If things are done correctly, it shall event allow us to efficiently program all the emerging hardware variants by itself
- C++2X (C++3X?) will not have all the capabilities that the LHC experiments require by the start of HL-LHC
  - We need to make sure in the next few years that we choose a programming method that will be as close to the eventual C++ standard as possible
- There is a lot of work to be done! If you’re interested, ATLAS is certainly welcoming enthusiastic software developers! 😉

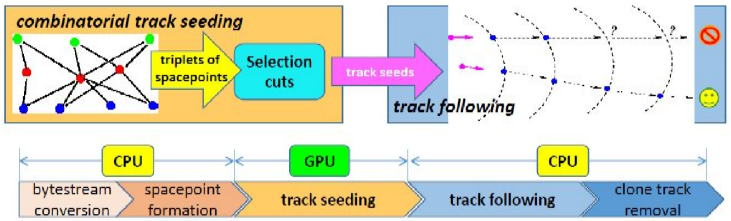
http://home.cern

Backup

Previous Studies (1)

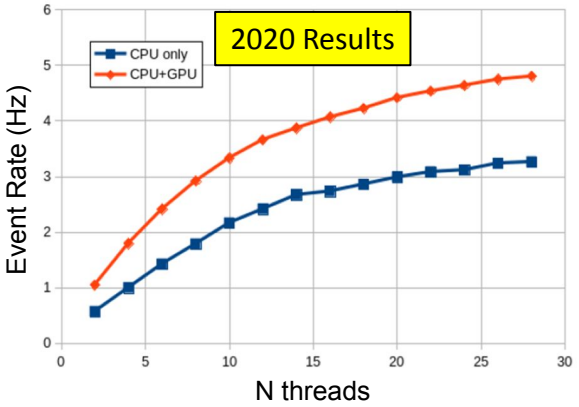
- 2012: ID Trigger prototype (ATL-DAQ-PROC-2012-006)
  - **Complete Level2 ID Trigger** on GPU (ByteStream to tracks)
  - GPU (Tesla C2050) gave **x12 speedup*** c.f. 1 CPU core
- 2015: Trigger GPU Demonstrator (ATL-COM-DAQ-2019-059)
  - **Athena integration** using client-server technology (APE)
  - **Calo topo-clustering** & cluster splitting: **x3.6 speedup*** on Kepler K80 GPU
  - **Pix & SCT clustering + ID seed-maker: x28 speed-up*** on Pascal GTX1080 GPU
  - Overall **trigger server throughput x1.4** throughput with GPU c.f. Cpu-only
- 2019: GPU ID pattern-matching prototype (ATL-COM-DAQ-2019-173)
  - FTK-like pattern matching on GPU

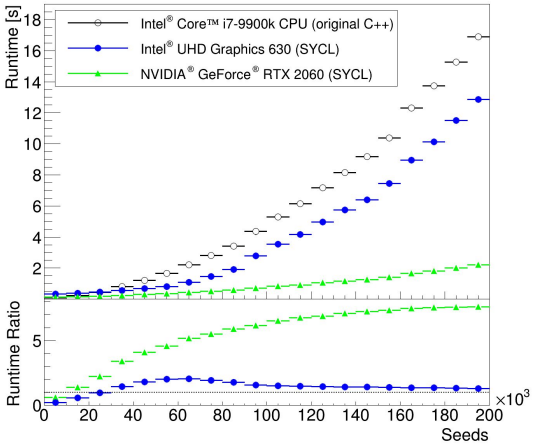




*speedup = time on 1 CPU core / time on GPU

Previous Studies (2)





- 2020: GPU trigger algorithm integration in AthenaMT
  - AthenaMT integration using acceleration service
  - ID seed-maker algorithm implemented on GPU
  - Calorimeter reconstruction under development
- Acts
  - Seed finding implemented using both CUDA and SYCL
    - https://github.com/acts-project/acts/tree/master/Plugins/Cuda
    - https://github.com/acts-project/acts/tree/master/Plugins/Sycl
  - Kalman filter demonstrator
- FCS: Parametrized Calorimeter Simulation
  - First developed in CUDA, but then used as a software portability testbed
  - ATL-COM-SOFT-2020-069
  - oneMKL cuRAND Support Development (GitHub Code)
- Studies with GNNs for tracking (presentation)