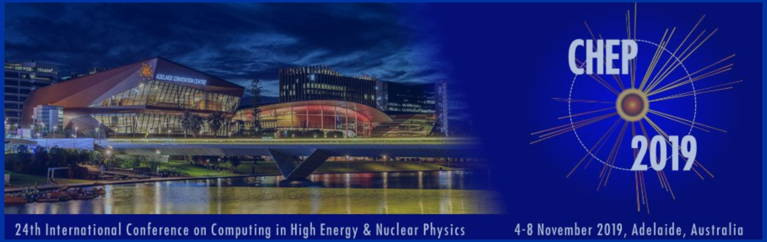
GPU Usage in ATLAS Reconstruction and Analysis

Attila Krasznahorkay
on behalf of the ATLAS Collaboration

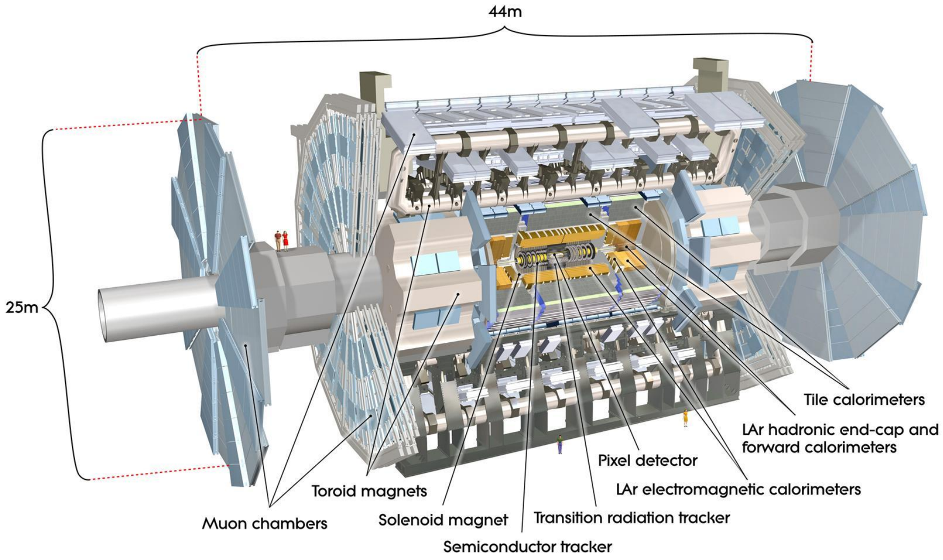


Overview

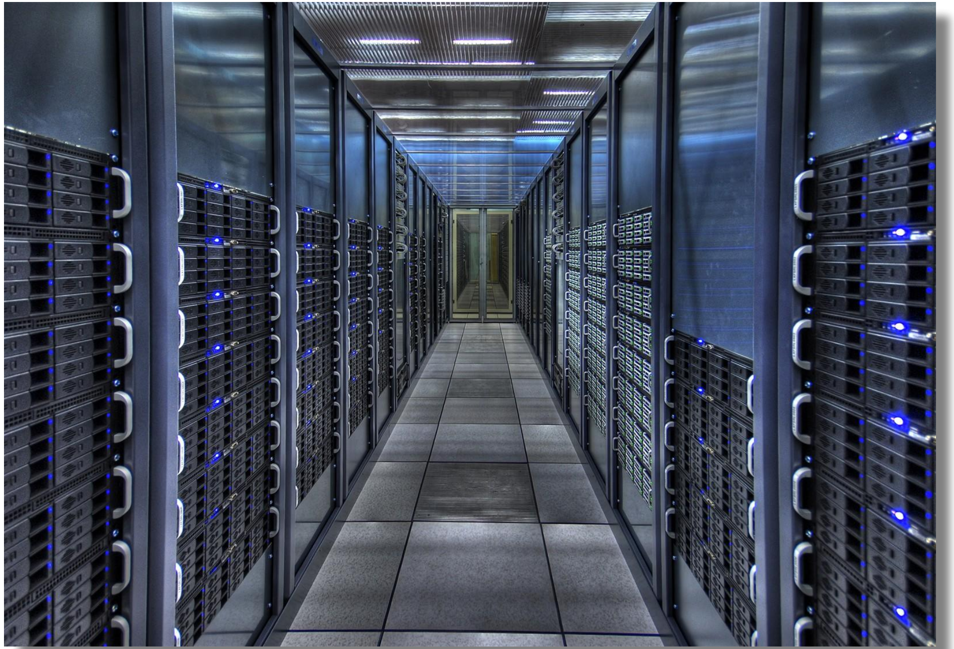
- Short introduction about ATLAS and its software / computing infrastructure
- Some details about multi-threading in Gaudi / AthenaMT
- Overview of current GPU programming possibilities
- Tests with offloaded calculations in the Gaudi / AthenaMT environment
- Future outlook

The ATLAS Experiment

- Since I'm the first person to talk about ATLAS in this session… 😛
- ATLAS is:
    - One of the general purpose experiments at the Large Hadron Collider
    - Collecting ~1.6 MB proton-proton (and sometimes Pb-Pb, Pb-p) data events with O(1) kHz rate, which our offline software has to process/analyse
    - Using hundreds of thousands of CPUs all over the world to process O(100) PB of data 24/7
    - Undergoing major hardware and software updates for LHC's Run-3/4

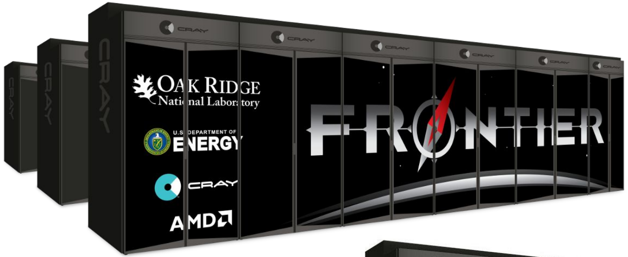


The (Current) Computing Landscape



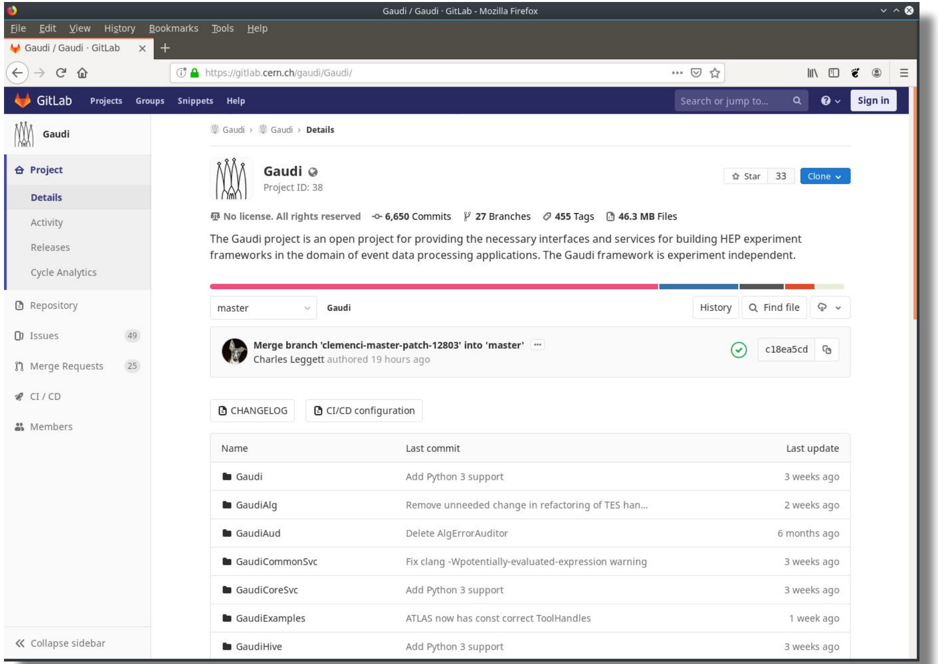
- Up until the very last steps of a physics analysis all our data can be processed in an embarrassingly parallel way
  - Every collision event recorded by the detector can be processed individually
- Up until now we do this by splitting the processing of events across many single-threaded x86 processes
  - We use a large infrastructure for this, which is being discussed in tracks 3, 4, 7 and (partly) 9
    - I.e. most of the tracks/sessions

The (Evolving) Computing Landscape



- Is a complicated one…
  - We are clearly moving towards a very heterogeneous environment for the foreseeable future
- Many different accelerators are on the market
  - NVidia GPUs are the most readily available in general, and also used in Summit and Perlmutter
  - AMD GPUs are not used too widely in comparison, but will be in Frontier
  - Intel GPUs are used even less at the moment, but will get center stage in Aurora
  - FPGAs are getting more and more attention, but they come with even more questionmarks…

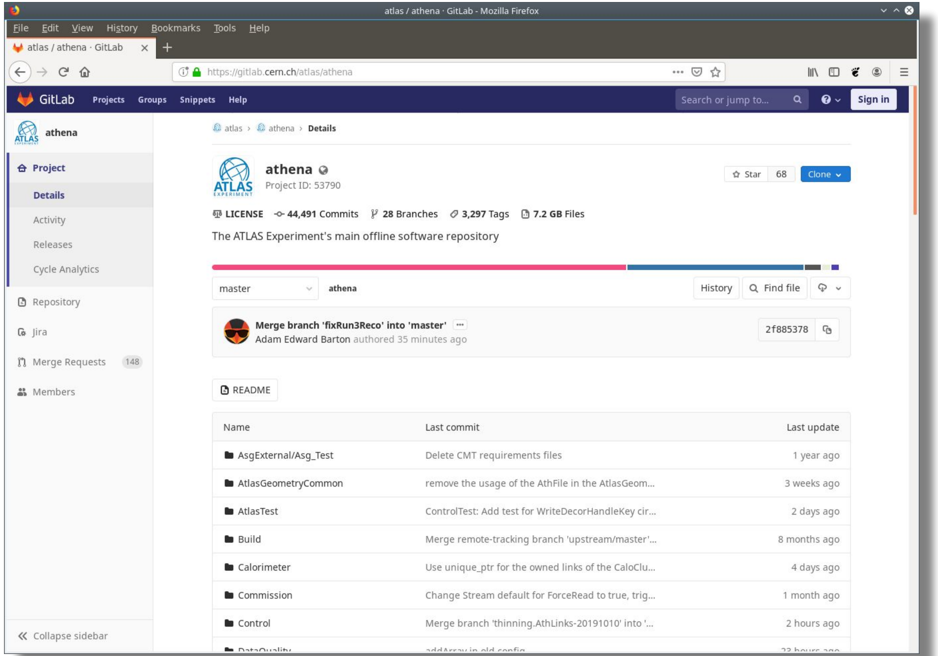
Gaudi / Athena



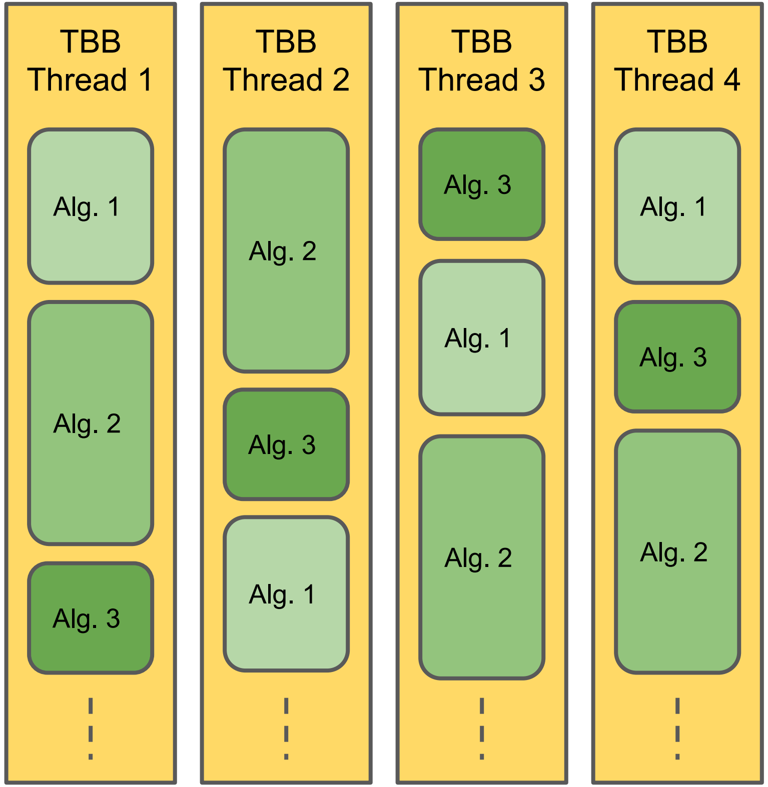
- ATLAS and LHCb share Gaudi as the basis of their software frameworks
  - ATLAS calls its own framework, built on top of Gaudi, Athena
- The framework defines “algorithms” as the base unit of execution
  - Classes that have an `execute(...)` function, which performs some data processing with the help of various “services” and “tools”

ATLAS’s Offline and Analysis Software

- All of ATLAS’s central offline software is kept in https://gitlab.cern.ch/atlas/athena
  - Some pieces, mainly those shared with other experiments, do sit in separate places though
- This allows us to build a number of different software projects from the same repository
  - The different projects build different selections of the code included in the repository
  - Providing us with (small) projects aimed at event generation, simulation and analysis beside our big reconstruction (Athena) project



Task Scheduling in AthenaMT



- Athena (Gaudi) uses TBB to execute algorithms on multiple CPU threads in parallel
  - The framework's scheduler takes care of creating TBB tasks that execute algorithms, at the "right times"
- The goal, of course, is to fully utilise all CPU cores assigned to the job, but not to use more
  - So any offloading needs to thoughtfully integrate into this infrastructure

Previous ATLAS GPU Efforts

- The idea of using accelerators in offline/trigger software is not new of course
- ATLAS presented some of its earlier efforts in:
  - https://iopscience.iop.org/article/10.1088/1742-6596/898/3/032003
  - https://twiki.cern.ch/twiki/bin/view/AtlasPublic/TriggerSoftwareUpgradePublicResults
- Previously the conclusion was not to pursue the usage of GPGPUs
  - The overall benefit was not worth the cost at that point

CHEP IOP Publishing
IOP Conf. Series: Journal of Physics: Conf. Series **898** (2017) 032003 doi:10.1088/1742-6596/898/3/032003

Multi-Threaded Algorithms for GPGPU in the ATLAS High Level Trigger

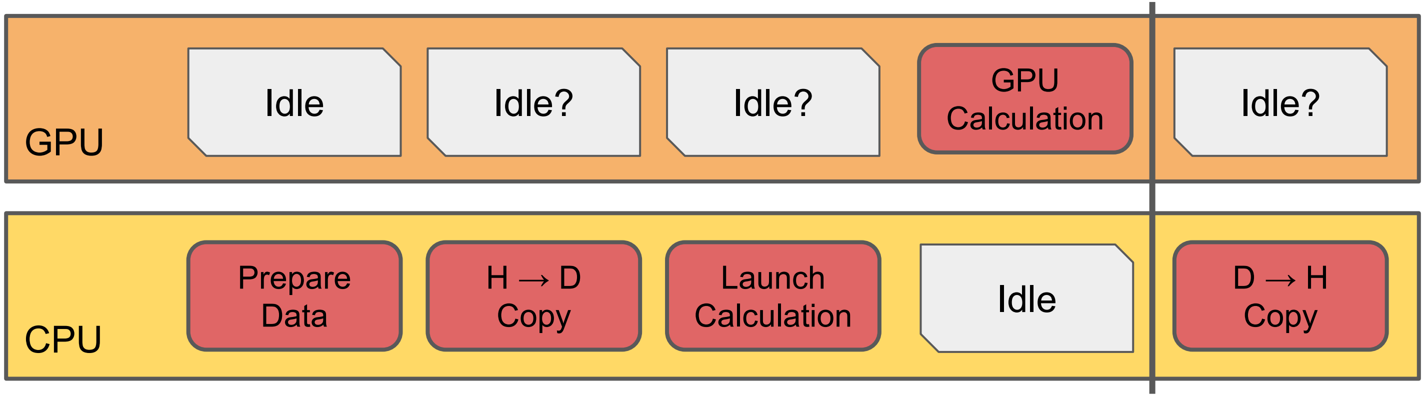
P. Conde Muíño on behalf of the ATLAS Collaboration
LIP (Laboratório de Instrumentação e Física Experimental de Partículas),
Elias Garcia 14, 1000-149 Lisbon, PT
Departamento de Física, Faculdade de Ciências, Universidade de Lisboa, PT

Abstract. General purpose Graphics Processor Units (GPGPU) are being evaluated for possible future inclusion in an upgraded ATLAS High Level Trigger farm. We have developed a demonstrator including GPGPU implementations of Inner Detector and Muon tracking and Calorimeter clustering within the ATLAS software framework. ATLAS is a general purpose particle physics experiment located on the LHC collider at CERN. The ATLAS Trigger system consists of two levels, with Level-1 implemented in hardware and the High Level Trigger implemented in software running on a farm of commodity CPU.

The High Level Trigger reduces the trigger rate from the 100 kHz Level-1 acceptance rate to 1.5 kHz for recording, requiring an average per-event processing time of ~ 250 ms for this task. The selection in the high level trigger is based on reconstructing tracks in the Inner Detector and Muon Spectrometer and clusters of energy deposited in the Calorimeter. Performing this reconstruction within the available farm resources presents a significant challenge that will increase significantly with future LHC upgrades. During the LHC data taking period starting in 2021, luminosity will reach up to three times the original design value. Luminosity will increase further to 7.5 times the design value in 2026 following LHC and ATLAS upgrades. Corresponding improvements in the speed of the reconstruction code will be needed to provide the required trigger selection power within affordable computing resources.

Key factors determining the potential benefit of including GPGPU as part of the HLT processor farm are: the relative speed of the CPU and GPGPU algorithm implementations; the relative execution times of the GPGPU algorithms and serial code remaining on the CPU; the number of GPGPU required, and the relative financial cost of the selected GPGPU. We give a brief overview of the algorithms implemented and present new measurements that compare the performance of various configurations exploiting GPGPU cards.

GPU / Accelerator Programming



- The general way of offloading calculations to an accelerator is fairly simple
  - Copy all information necessary for the calculation onto the accelerator, run the calculation, and copy the results back into the host's memory
- The naive way of doing this leaves the CPU/GPU idle for long periods
  - For calculations that can be done on a GPU much faster than on a CPU, this is acceptable
  - Unfortunately calculations used in reconstruction/analysis tend not to be like that ☹
- Our goal was to see how to efficiently offload calculations from our TBB based framework

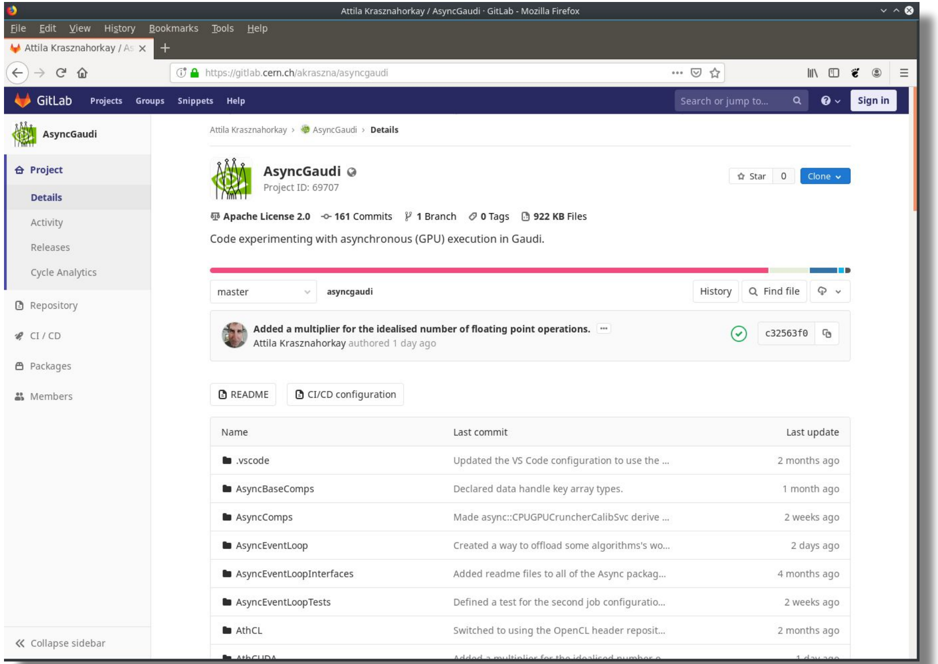
Programming Languages/Methods

- There is absolutely no general agreement currently on how to write accelerated calculations
  - Each hardware manufacturer has its own methods, which overlap very little
- OpenCL
  - Had the most promise as a standard
  - Initially supported by most manufacturers, but the support disappeared by now
    - The best features of the standard were never implemented by NVidia or AMD
- CUDA
  - Is the clear market leader at the moment, and the most well developed programming environment for GPUs
  - Runs only on NVidia GPUs
- HIP/ROCm
  - AMD's version of a combination of CUDA's and OpenCL's best features
  - Can in principle target both AMD and NVidia GPUs, but support for NVidia GPUs in the future is anything but certain
- OpenMP/OpenACC
  - Not appropriate for integrating with out TBB based framework, even though these are the most widely supported ways of writing accelerated code at the moment ☹️

Programming Languages/Methods

- SYCL
  - A very promising standard from the same group that oversees (oversaw) OpenCL as well
  - Very actively supported by Intel as the programming interface for their (future) accelerators
- Single source portability
  - Our big issue is that not only don't we want to implement calculations separately for all different accelerators, we don't want to implement them separately for (classical) CPUs and accelerators either
    - Our codebase is much too large to reliably maintain (and validate) multiple implementations of the same components
  - Only OpenMP, OpenACC and SYCL provide single source portability like this out of the box
    - And as said previously, OpenMP/OpenACC are inappropriate for us for other reasons
  - CUDA/HIP allow us to write our own layer on top of them that provides this sort of single source feature
    - But these layers are in all cases quite intimately tied to the underlying accelerator programming interface

AsyncGaudi



- I collected all of my “Athena test code” into https://gitlab.cern.ch/akraszna/asyncgaudi
  - The code combines parts of atlas/atlasexternals, atlas/athena and gaudi/Gaudi with accelerator test code built on top of these, into a single software project
    - In order to make it possible to compile/use it on multiple different platforms
  - Contains some code using OpenCL, CUDA and SYCL
    - With the CUDA code being the most developed/tested

(A)synchronous Execution

- Based on discussions with CMS software developers, I wrote a Gaudi algorithm scheduler that can handle "asynchronous algorithms"
  - Algorithms that have a "main" and a post-execute step, and have to notify the scheduler when they are ready for their post-execute step
- CUDA provides asynchronous execution through its Stream API
  - OpenCL and HIP provide similar features, but SYCL at the moment does not support this sort of execution natively

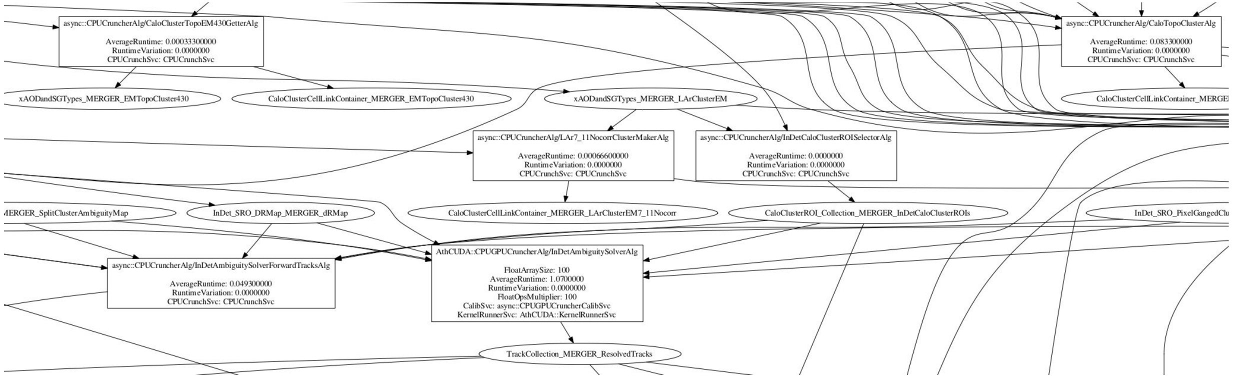
```
/// @name Custom execute interface for asynchronous calculations
///
/// This "execution interface" is used on the algorithm when
/// @c async::Algorithm::isAsynchronous() returns @c true for the
/// algorithm object.
///
/// @{

/// Execute the algorithm on the host or an asynchronous device
///
/// The algorithm is provided with a pre-prepared task, which, when
/// executed, calls the @c postExecute function of the algorithm.
/// The function must take care of either enqueing the task (using
/// @c async::Algorithm::enqueue), or passing the task to some service
/// that will enqueue the task once the asynchronous calculation set up
/// by this function has finished.
///
/// @param ctx The current event's context
/// @param postExecTask Task executing the @c postExecute function
///                     of this algorithm
/// @return The usual @c StatusCode values
///
virtual StatusCode mainExecute( const EventContext& ctx,
                                AlgTaskPtr_t postExecTask ) const;

/// Run a post-execution step on the algorithm
///
/// This step is meant mainly to collect data from an asynchronous
/// calculation, and save it into the event store.
///
/// @param ctx The current event's context
/// @return The usual @c StatusCode values
///
virtual StatusCode postExecute( const EventContext& ctx ) const;

/// @}
```

Reconstruction Emulation



- During the development of GaudiHive snapshots were taken of the behaviour of ATLAS reconstruction jobs
  - Recording how algorithms depended on each others' data products, and how long each of them took to run on a reference host
  - The data is still kept in GaudiHive/data/atlas in GraphML + JSON files
- This information was used extensively in the development of the algorithm scheduling code of Gaudi not that long ago
  - And now I taught my project how to construct asynchronous test jobs using it

CPU / GPU Crunching

```
StatusCode CPUGPUCruncherAlg::mainExecute( const EventContext&,
                                            async::AlgTaskPtr_t pet ) const {

   // Create a vector of floats with the configured size. This will be the
   // array to run the calculation on.
   auto array =
      std::make_unique< std::vector< float > >( m_floatArraySize.value(),
                                                 0 );

   // Set up a calculation task on this array.
   auto task = make_CPUGPUCruncherTask( m_fpOps, *array,
                                        std::move( pet ) );
   ATH_CHECK( m_kernelRunnerSvc->execute( std::move( task ) ) );

   // Record it into the event store.
   ATH_CHECK( evtStore()->record( std::move( array ),
                                  m_outputKeys.value()[ 0 ].objKey() ) );

   // Return gracefully.
   return StatusCode::SUCCESS;
}

StatusCode CPUGPUCruncherAlg::postExecute( const EventContext& ) const {

   // Just print a debug message.
   ATH_MSG_DEBUG( "CPU / GPU crunching task finished" );

   // Return gracefully.
   return StatusCode::SUCCESS;
}
```

- The tests were not using any “real” ATLAS reconstruction code
- To emulate the behaviour of “CPU-only” algorithms, I used the same CPUCrunchSvc that was developed for the GaudiHive tests originally
- For the GPU emulation I did something different…
  - The test jobs measure during initialisation how many FPOPS the CPU can do in a single thread in a unit of time
  - With this information I associate FPOPS values to the time values stored in the GaudiHive data files
  - The GPU tasks then execute this number of FPOPS on small arrays, with some configurable multipliers applied

Reconstruction Emulation Results

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

- Did a number of tests…
  - As reference ran jobs with only using the sort of CPU crunching that was developed previously
  - As a validation I exchanged some of the algorithms to run my CPU/GPU crunching code, but running only on the CPU
    - Checking that I’d get the same results as in the first case
  - Finally configured 3 of the CPU intensive reconstruction algorithms to run on the GPU instead
    - Applying also an additional multiplier to the number of FPOPS that they’d have to execute on the GPU

Reconstruction Emulation Results

Setup	Time [s]
50 events, 8 thread… algorithms	
50 events, 8 thread… "critical-path" CPU/… run only on CPUs	
50 events, 8 thread… "critical-path" algori… with ideal FPOPS	
50 events, 8 thread… path" algorithms of… FPOPS	
50 event, 8 threads… non-critical-path" al… offloaded with ideal…	
50 events, 8 threads, 4 "heavy non-critical-path" algorithms offloaded with 3x FPOPS	70.3 ± 10.0

Some takeaways:

- One has to be very careful with offloading algorithms that many other algorithms depend on
  - Making these too slow can cause big issues for the job
- Algorithms off of the "critical path" can handle being executed less efficiently on an accelerator, but not by much
- My ASync::SchedulerSvc code is clearly not scheduling asynchronous algorithms as efficiently as it should at the moment
  - As it turns out, that is **very** important to do, otherwise the job is not able to fill its CPU/GPU resources efficiently.

- Did a number of tests
  …nly using the
  …as developed

  …some of the
  …PU crunching
  …e CPU
  …e same results

  …PU intensive
  …run on the

  …onal multiplier
  …S that they'd
  have to execute on the GPU

Summary

- ATLAS (and HEP in general) has to take computations on heterogeneous hardware very seriously
  - Our computing pattern is quite distinct from other fields, requiring different methods for using accelerators efficiently
- We are currently evaluating different programming methods for the ATLAS offline software in close collaboration with Intel and NVidia
  - Whatever programming model we choose to migrate some of our code to, has to stay viable for a "reasonable" period of time
  - Will only start large scale migrations after further evaluations
- Asynchronous scheduling of calculations in our TBB based software framework show promising results so far
  - Although it **is** a concern how inefficient algorithms can get before we lose any advantage from running them asynchronously
  - Efficient parallel execution of GPU kernels using TBB is **very** important for us!

Backup

AthCUDA::AuxStore

- The ATLAS (analysis) Event Data Model uses the concept of “auxiliary stores” to store event data
  - The whole idea with that setup was to abstract the storage of data from the way that we interact with it
- For my tests with CUDA I implemented AthCUDA::AuxStore
  - It manages arrays of primitive types in unpinned host memory through the SG::IAuxStore interface
  - It provides functions setting up the H→D and D→H copies of those arrays asynchronously
  - Finally it provides a non-virtual interface to the arrays for CUDA device code

```
class AuxStore : public SG::IAuxStore {

public:
   /// Default constructor
   ATHCUDA_HOST
   AuxStore();
   /// Constructor from existing data
   ATHCUDA_HOST_AND_DEVICE
   AuxStore( std::size_t size, std::size_t nVars, void** vars );
   /// Destructor
   ATHCUDA_HOST_AND_DEVICE
   ~AuxStore();

   /// @name Interface for using the data on a (GPU) device
   /// @{

   /// Get the size of the variable arrays
   ATHCUDA_HOST_AND_DEVICE
   std::size_t arraySize() const;

   /// Function for accessing a variable array (non-const)
   template< typename T >
   ATHCUDA_HOST_AND_DEVICE
   T* array( SG::auxid_t auxid );

   /// Function for accessing a variable array (const)
   template< typename T >
   ATHCUDA_HOST_AND_DEVICE
   const T* array( SG::auxid_t auxid ) const;

   /// Type of the variable returned by the @c attachTo function
   typedef std::pair< std::size_t, void** > ExposedVars_t;

   /// Get the array of variables to be used on a CUDA device
   ATHCUDA_HOST
   ExposedVars_t attachTo( cudaStream_t stream );

   /// Retrieve the variables from a CUDA device
   ATHCUDA_HOST
   void retrieveFrom( cudaStream_t stream );

   /// @}
```

AthCUDA::IKernelRunnerSvc

```
namespace AthCUDA {

   /// Interface for executing @c AthCUDA::IKernelTask tasks
   ///
   /// The implementation of this service is supposed to be used for executing
   /// "GPU calculations" in a balanced way between the CPU and the GPU.
   ///
   /// @author Attila Krasznahorkay <Attila.Krasznahorkay@cern.ch>
   ///
   class IKernelRunnerSvc : public virtual IService {

   public:
      /// Declare the interface ID
      DeclareInterfaceID( AthCUDA::IKernelRunnerSvc, 1, 0 );

      /// Execute a user specified kernel task
      ///
      /// If a GPU is available at runtime, and it is not doing other things
      /// at the moment, this function offloads the calculation to the GPU,
      /// and returns right away. The user is expected to return control in the
      /// calling thread to the framework, as the kernel task will notify the
      /// framework when the task gets finished.
      ///
      /// If a GPU is not available for any reason, the function just executes
      /// the task on the CPU in the caller thread, and returns only once the
      /// task is finished.
      ///
      /// @param task The task to be executed on the CPU or GPU
      /// @return The usual @c StatusCode values
      ///
      virtual StatusCode execute( std::unique_ptr< IKernelTask > task ) = 0;

   }; // class IKernelRunnerSvc

} // namespace AthCUDA
```

- Since “memory operations” and kernel offloads with CUDA need to happen one at a time, I introduced a service for serialising these tasks
    - The tasks are still executed as TBB tasks, the service just uses a custom task arena to make sure that these tasks are executed one at a time
    - The service also takes care of scheduling the post-execute task for asynchronous algorithms

AthCUDA::AuxKernelTask

- The actual calculations happen in specialisations of the AthCUDA::IKernelTask interface
- AthCUDA::AuxKernelTask is a variadic template that can be used to run calculations on AthCUDA::AuxStore objects
  - It can wrap user provided functors, which would be executed either on a CUDA device, or on the host depending on the circumstances

```
template< class FUNCTOR, typename... ARGS >
class AuxKernelTask : public IKernelTask {

   // At least one argument has to be provided.
   static_assert( sizeof...( ARGS ) > 0,
                  "At least one functor argument must be provided" );

public:
   /// Constructor to use in a non-blocking execution
   AuxKernelTask( ASync::AlgTaskPtr_t postExecTask, AuxStore& aux,
                  ARGS... args );
   /// Constructor to use in a blocking execution
   AuxKernelTask( KernelStatus& status, AuxStore& aux, ARGS... args );

   /// @name Function(s) inherited from @c AthCUDA::IKernelTask
   /// @{

   /// Execute the kernel using a specific stream
   virtual StatusCode execute( StreamHolder& stream ) override;

   /// Function called when an asynchronous execution finishes
   virtual StatusCode finished( StatusCode code,
                                KernelExecMode mode ) override;

   /// @}

private:
   /// A possible task object to use for executing a post-execute step
   ASync::AlgTaskPtr_t m_postExecTask;
   /// A possible status object to notify about the task finishing
   KernelStatus* m_status;
   /// The auxiliary container to execute the calculation on
   AuxStore& m_aux;
   /// The arguments to pass to the functor
   std::tuple< ARGS... > m_args;

}; // class AuxKernelTask
```

AthCUDA::ArrayKernelTask

```
class ArrayKernelTask : public IKernelTask {

   // At least one argument has to be provided.
   static_assert( sizeof...( ARGS ) > 0,
                  "At least one functor argument must be provided" );

public:
   /// Constructor to use in a non-blocking execution
   ArrayKernelTask( ASync::AlgTaskPtr_t postExecTask,
                    std::size_t arraySizes, ARGS... args );
   /// Constructor to use in a blocking execution
   ArrayKernelTask( KernelStatus& status,
                    std::size_t arraySizes, ARGS... args );

   /// @name Function(s) inherited from @c AthCUDA::IKernelTask
   /// @{

   /// Execute the kernel using a specific stream
   virtual StatusCode execute( StreamHolder& stream ) override;

   /// Function called when an asynchronous execution finishes
   virtual StatusCode finished( StatusCode code,
                                KernelExecMode mode ) override;

   /// @}

private:
   /// A possible task object to use for executing a post-execute step
   ASync::AlgTaskPtr_t m_postExecTask;
   /// A possible status object to notify about the task finishing
   KernelStatus* m_status;
   /// The size of the arrays being processed
   std::size_t m_arraySizes;
   /// The arguments received by the constructor
   std::tuple< ARGS... > m_args;
   /// The received variables, copied into pinned host memory
   typename ::ArrayKernelTaskHostVariables< ARGS... >::type m_hostObjs;
   /// The received variables, in device memory
   typename ::ArrayKernelTaskDeviceVariables< ARGS... >::type m_deviceObjs;
   /// The arguments received by the constructor, in device memory
   std::tuple< ARGS... > m_deviceArgs;
   /// Status flag showing that the kernel was run on a device
   bool m_ranOnDevice;

}; // class ArrayKernelTask
```

- AthCUDA::ArrayKernelTask is a variadic template that can execute user functors that have a custom set of primitive and primitive array arguments
  - The code assumes that all pointer type variables point at arrays of equal sizes
- Is probably the least trivial part of the akraszna/asyncgaudi code…

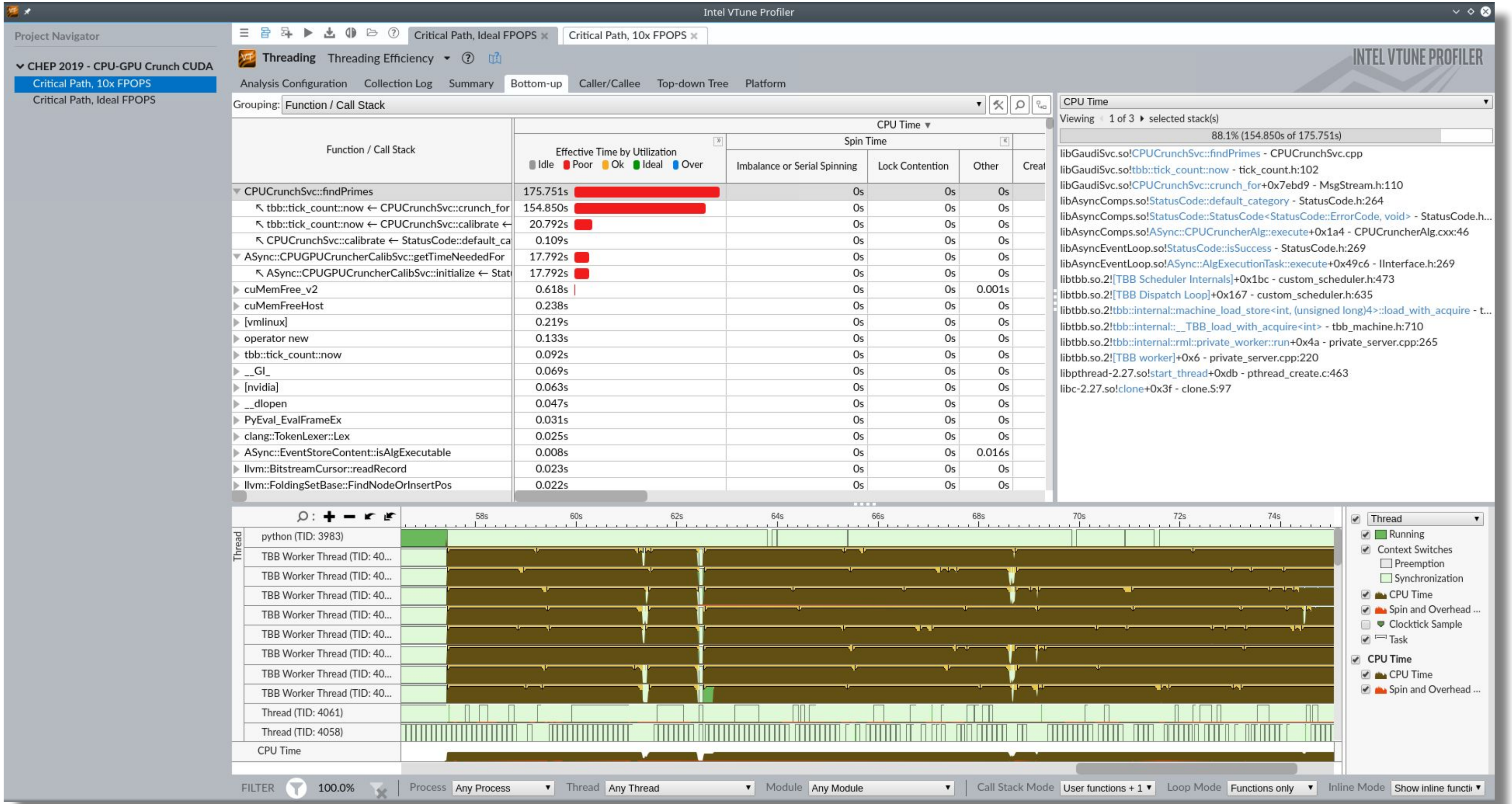
OpenCL Experiences

- Tried to use it in a few different ways
  - Directly, by getting/compiling OpenCL-Headers, OpenCL-ICD-Loader and POCL as part of our project
    - In order not to rely on OpenCL libraries/devices being present on the build/run host
  - Through tbb::flow::opencl_node
- If OpenCL 2.X would be widely supported in the industry, that would clearly be our choice for writing GPU code
  - Even with the inconvenience of keeping the OpenCL source files completely separately from the C++ ones
  - OpenCL 1.2 by itself does not fit our requirements
- But since nobody is expressing interest in it any longer, we have also given up on it…

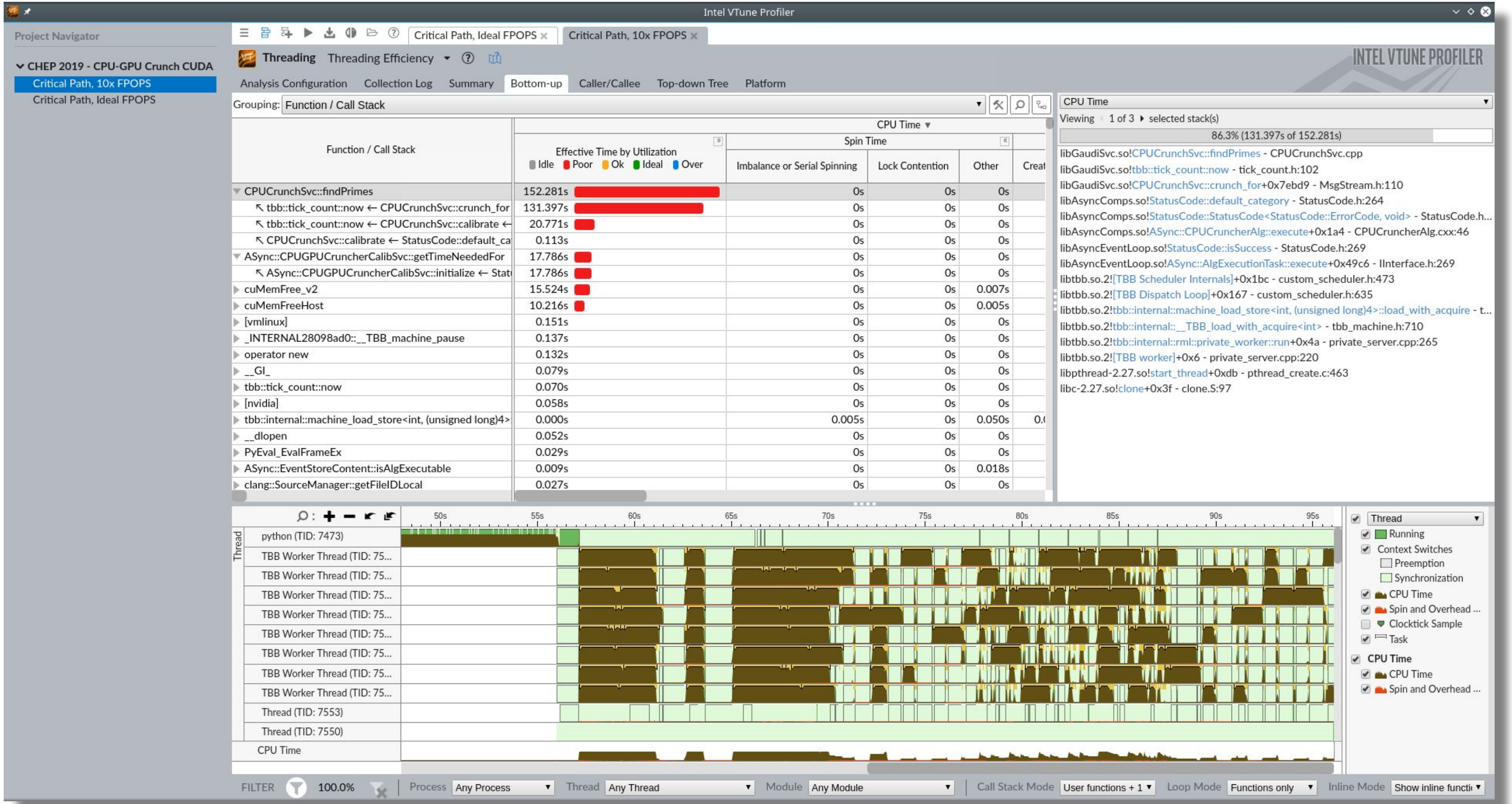
OpenACC Experiences

- Only did some very minimal testing with it so far
- Unfortunately, just as OpenMP, it does not fit our offline software
  - Our software does not have well identifiable hot spots, accelerated code will in all cases have to be fairly complex
- Support in GCC 8 is/was very shaky
  - Did not try with GCC 9 yet

Test Job Profiling



Test Job Profiling



CERN

http://home.cern