Using (GPU) Accelerators in HEP Software

Attila Krasznahorkay

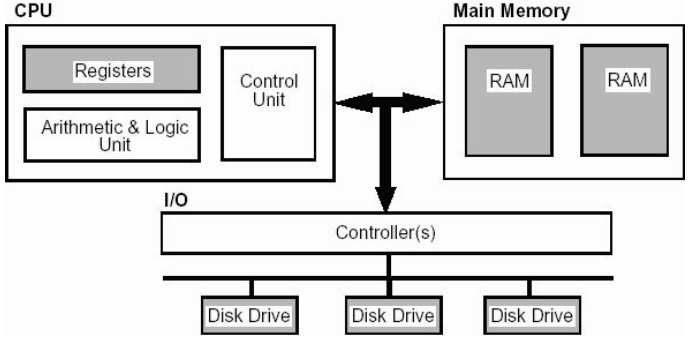
Outline

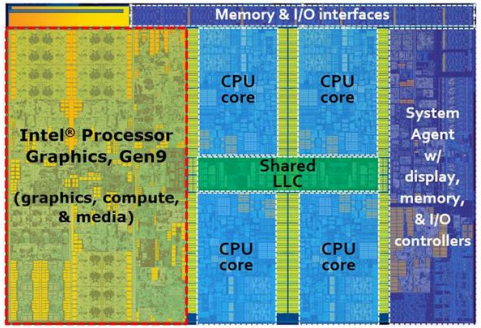
- An overview of computing and accelerators
  - And why HEP is interested in them
- Programming heterogeneous/GPU hardware
  - Including some amount of technicalities

(High Performance) Computing in 2021

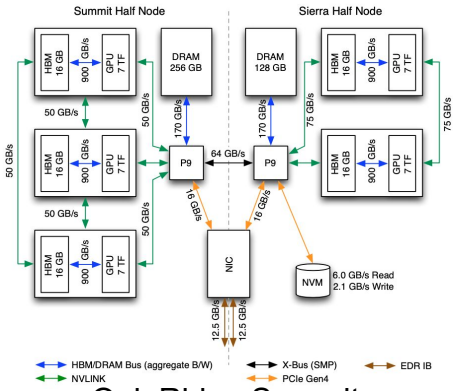
- Computing has been getting more and more complicated in the last decades
  - A modern CPU has a very complicated design, mainly to make sure that (our!) imperfect programs would execute fast on it
- Complexity shows up both “inside of single computers”, but also in the structure of computing clusters
  - 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™

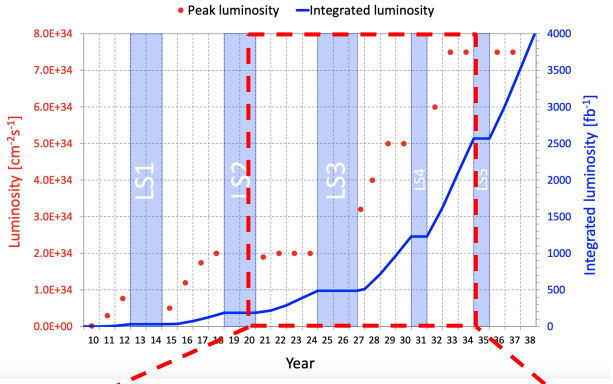


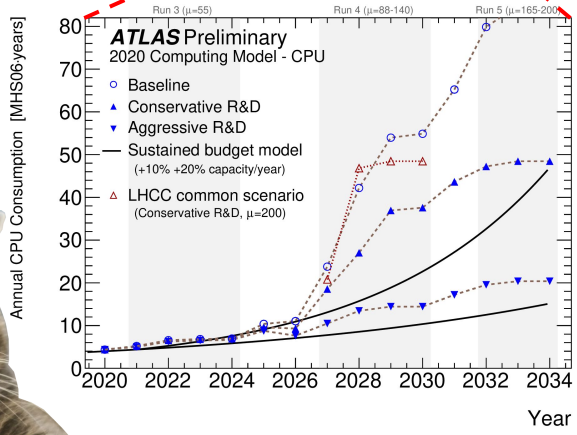
Oak Ridge Summit

(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 Should HEP Care?

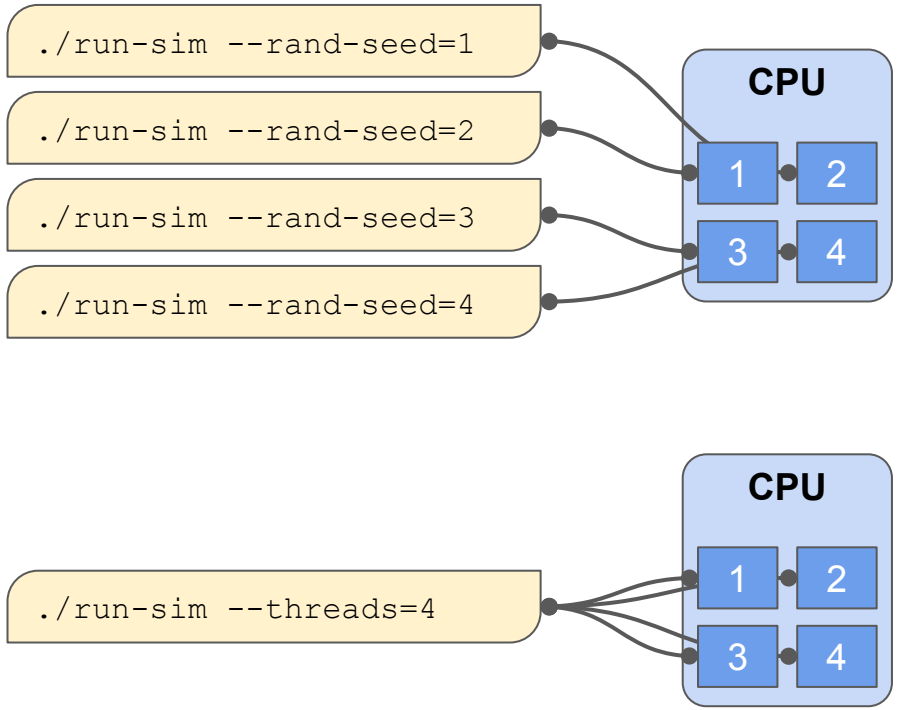




- As described in CERN-LHCC-2020-015, being able to process the data collected in LHC Run 4 (and beyond) in ATLAS requires major software developments
  - In order to fit into our “CPU budget”, we need to consider new approaches in our data processing
- One of these areas is to look at non-CPU resources

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.



(CPU vs. GPU) Multithreading

- Multithreading can be done in a lot of different ways. It all depends on what your code is doing exactly.
- But in general we can categorise them as:
  - Parallelising similar / the same calculations on multiple data
    - Similar to SIMD (SIMT). Relatively easily portable to GPUs.
    - Can be expressed using either in-language constructs (for instance in C++) or “pragmas” (for instance in Fortran)

```
float a[ size ] = …;
tbb::parallel_for(
   tbb::blocked_range<std::size_t>(0, size),
   [&a](...){...} );
```

```
   float a[ size ] = …;
#pragma omp for
   for( std::size_t i = 0; i < size; ++i )
      do_something( a[ i ] );
```

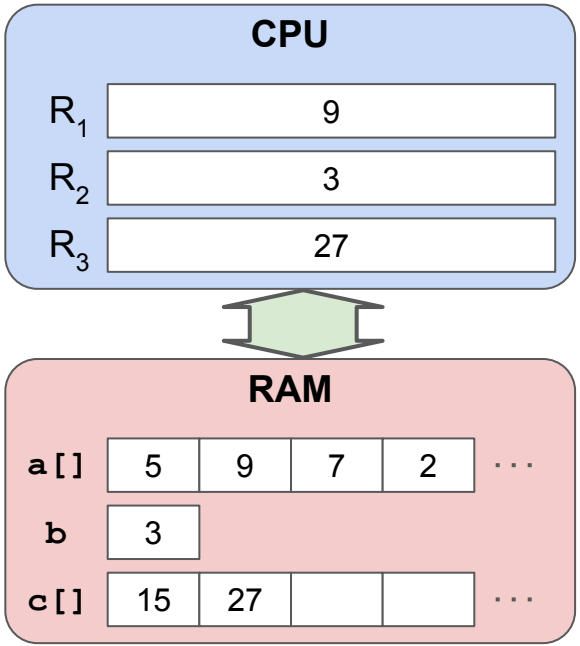
  - Running independent calculations in parallel
    - This is mostly called “task based multi-threading”. Much more difficult to port to GPUs.

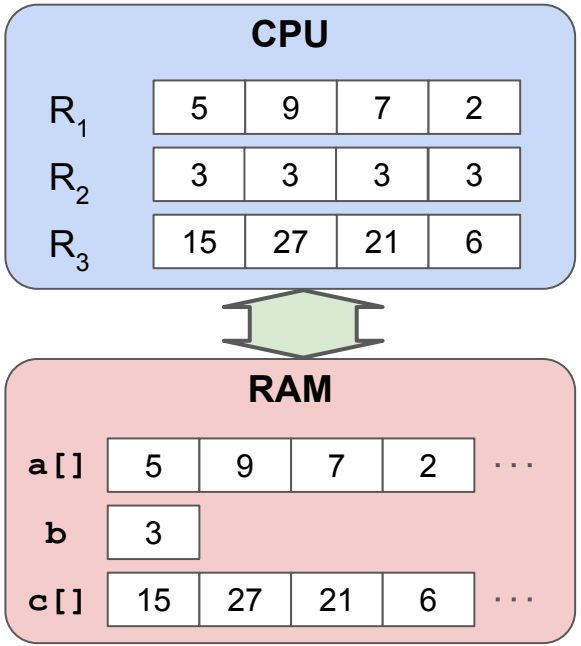
```
   tbb::task_group tg;
   tg.run( [...](...){...} );
   tg.run( [...](...){...} );
   tg.wait();
```

  - In HEP we overwhelmingly use task based multithreading…

SIMD, SIMT

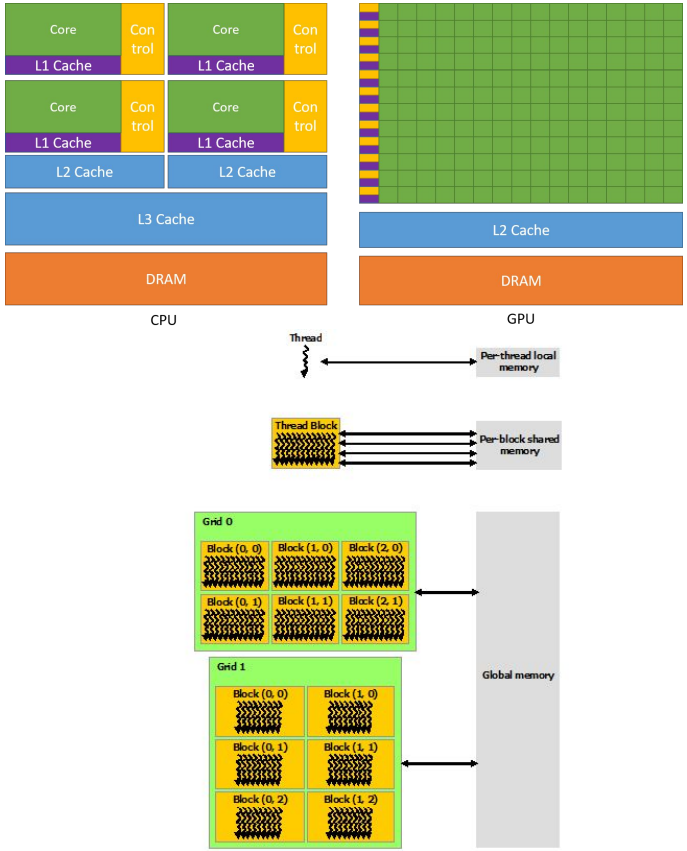
```
for(int i = 0; i < N; ++i)
    c[i] = a[i] * b
```





Accelerators / GPGPUs

- General Purpose GPUs (GPGPUs) 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

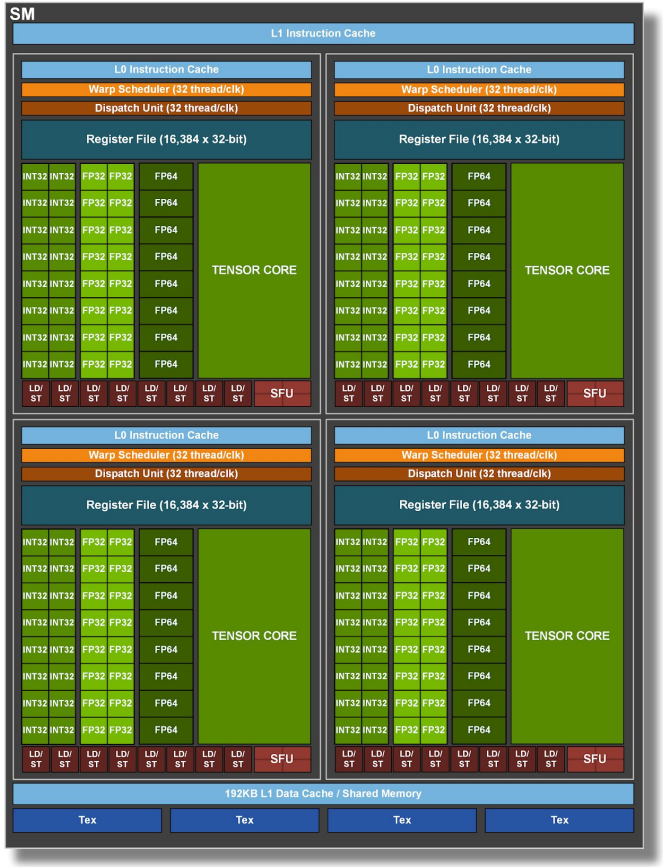


HEP Software

- 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

Heterogeneous Hardware

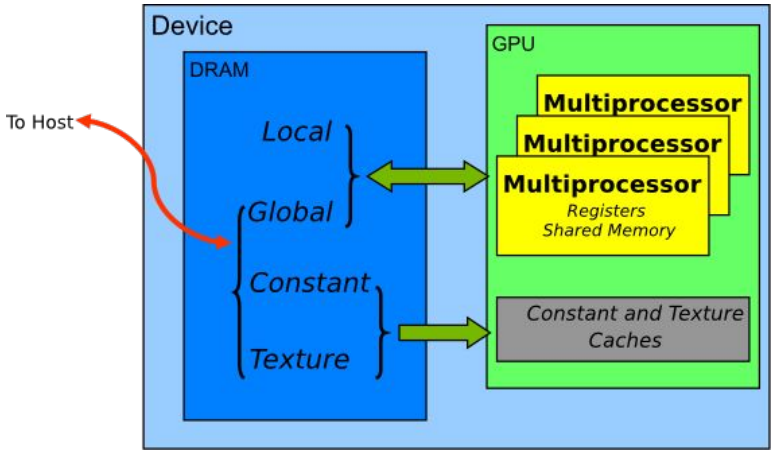
Modern GPUs

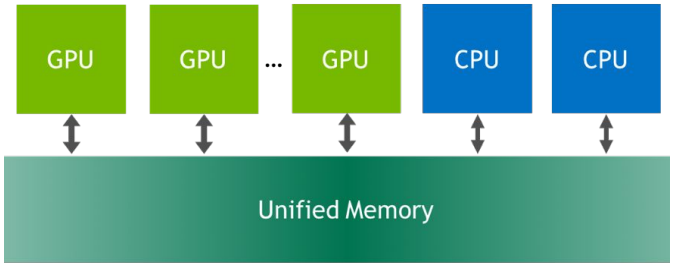


- Just like modern CPUs, modern GPUs have been getting more and more complicated
- Similar to advanced CPU instructions, they have purpose-built units for performing specific tasks
  - High-throughput 16/32/64-bit integer/floating point calculations
  - Multi-dimensional matrix (tensor) operations
  - Ray-tracing operations
- Unfortunately just as how we struggle to use SSE/AVX instructions in our CPU code, we will likely struggle using Tensor/RT cores ☹

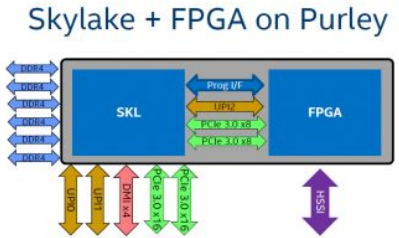
Memory Management

- 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, no implicit/automatic caching is happening on GPUs
- In some cases however you can get away with not knowing everything
  - For a performance penalty...





The Future of CPUs/GPUs (?)



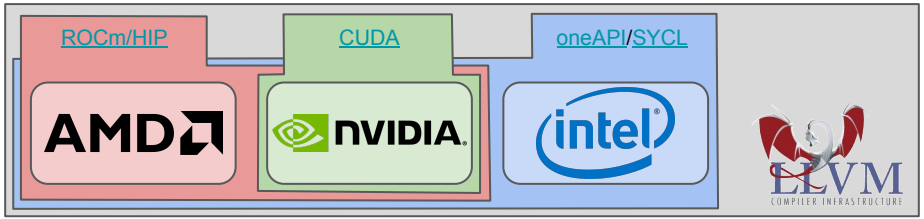
Cores	Up to 28C with Intel® HT Technology	
FPGA	Altera® Arria 10 GX 1150	
Socket TDP	Shared socket TDP Up to 165W SKL & Up to 90W FPGA	
Socket	Socket P	
Scalability	Up to 2S – with SKL-SP or SKL + FPGA SKUs	
PCH	Lewisburg: DMI3 – 4 lanes; 14xUSB2 ports Up to: 10xUSB3; 14xSATA3, 20xPCIe*3 Newc Innovation 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…

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 different capabilities



- Multiple projects are underway for hiding this complexity from the programmers (Kokkos, Alpaka, Thrust, Parallel STL, etc.)
  - In the US HEP-CCE is looking at this, but mostly as a bystander…
  - Eventually the goal is to make heterogeneous programming part of the ISO C++ standard, but that won’t realistically happen before the 2030s

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

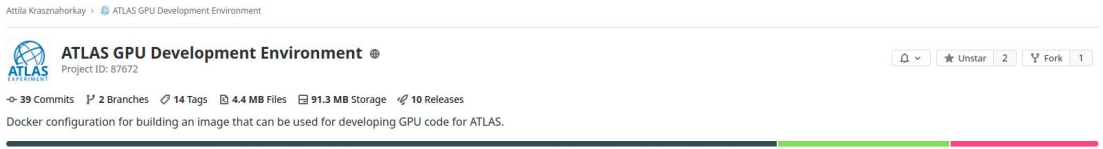
- Intel's answer to the programming language question
- Unlike CUDA, does not require an extension of the C++ language
  - Which means that it's possible to provide support for SYCL code using "a library" with any compiler
    - As long as GPU support is not required
- Very strong design-wise, built 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 at least questionable…

oneAPI

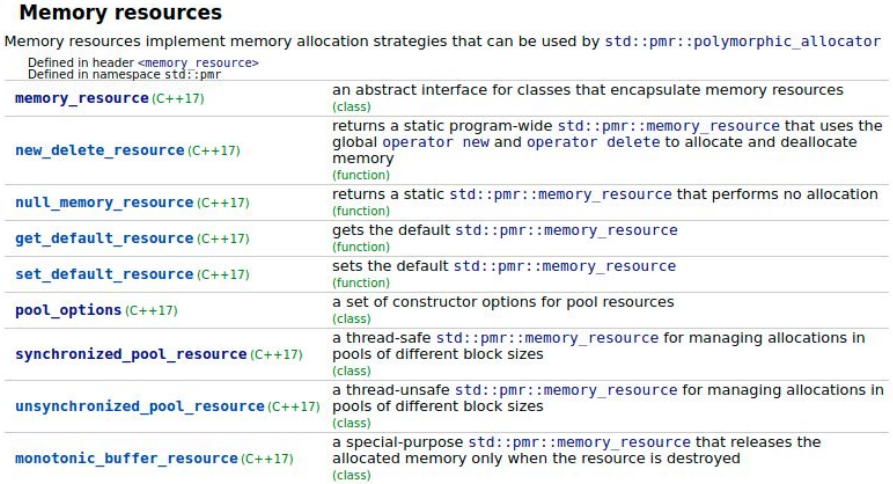
```
      // Create a vector array that would be manipulated.
      std::vector< float > dummyArray;
      static const std::size_t ARRAY_SIZE = 10000;
      dummyArray.reserve( ARRAY_SIZE );
      static const float ARRAY_ELEMENT = 3.141592f;
      for( std::size_t i = 0; i < ARRAY_SIZE; ++i ) {
  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

Software Installation

- The pecking order is much the same here as in all other areas
- Beside macOS (which is a longer story itself…) CUDA can be used practically anywhere
- oneAPI, being a much newer project, has been focusing on a smaller number of Linux distributions and on Windows
  - macOS support is almost for sure to come eventually
- ROCm/HIP is only supported on Linux
  - And is in general the most finicky to install correctly
- Since setting them up correctly can be a chore, I spent the time a while ago to create a Docker image that holds all of them side by side
  - https://gitlab.cern.ch/akraszna/atlas-gpu-devel-env



C++20 and Beyond



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)

- Both Intel and NVidia are hard at work to extend the ISO C++ standard according to their own taste
  - In practice so far the CUDA and SYCL languages/concepts are moving closer to each other 🥳
    - SYCL adopted the same (simple) memory management style used by CUDA
    - CUDA is looking towards declaring device code in-situ, much like SYCL does
- With C++17/20 one can already make use of some advanced memory handling features

Writing Code

Disclaimer

- From here on out I will be using CUDA in my examples
  - Anybody starting to write code for GPUs should just look at CUDA at first. As it gives the widest range of programming options at the moment.
- All described concepts are available in ROCm/HIP and oneAPI/SYCL as well, just with slightly different incantations

Host Code ↔ Device Code

- On first order when you compile x86_64 code on one machine, that will run on another x86_64 machine as well
  - However this is mostly because we tend not to use advanced (SSE, AVX) instructions in HEP code
  - If you do, managing your code can become a whole lot more complicated.
    - See for instance: https://gcc.gnu.org/onlinedocs/gcc/Function-Multiversioning.html
- Since GPU hardware evolved much more rapidly recently than CPU hardware, most of the time you don't ship binary "device code" with your application
  - Instead the application would hold some sort of "intermediate representation" of your code, which could be assembled into machine code for the GPU(s) at runtime
  - But even with this, you still need to specify which "compute capability" you want to support as a minimum by your code
    - This is a bit specific to NVidia hardware at the moment. AMD and Intel don't have a long enough hardware history yet for this to be an issue…

```
nvcc … -arch sm_50 …
```

```
set( CMAKE_CUDA_ARCHITECTURES "sm_50" )
```

A Trivial Example

```
__global__
void myKernel( std::size_t size, const float* input, float* result, float a, int b ) {
   const std::size_t index = blockIdx.x * blockDim.x + threadIdx.x;
   if( index >= size ) {
      return;
   }
   result[ index ] = input[ index ] * a + b;
}

int main() {
   static const std::size_t ARRAY_SIZE = 1e6;
   float *input = nullptr, *output = nullptr;
   CUDA_ERROR_CHECK( cudaMallocManaged( &input, ARRAY_SIZE * sizeof( float ) ) );
   CUDA_ERROR_CHECK( cudaMallocManaged( &output, ARRAY_SIZE * sizeof( float ) ) );
   fillWithData( input );

   static const int threadsPerBlock = 1024;
   const int nBlocks = ( ( ARRAY_SIZE + threadsPerBlock - 1 ) / threadsPerBlock );
   myKernel<<< nBlocks, threadsPerBlock >>>( ARRAY_SIZE, input, output, 1.23f, 34 );
   CUDA_ERROR_CHECK( cudaGetLastError() );
   CUDA_ERROR_CHECK( cudaDeviceSynchronize() );
   ...

   return 0;
}
```

(Explicit) Memory Management

- In your code you always need to explicitly differentiate between “device” and “host” memory
  - While integrated GPUs may use “host” memory directly, your code should never assume this. The runtime can skip explicit memory copies if they are not necessary.
- All of this is nothing magic, all of the memory management happens in the same way as in ISO C
  - The only tricky thing is that you always get pointers for “device memory” that can never be valid in host code, and vice versa.

```
// Allocate memory on the host.
static const std::size_t ARRAY_SIZE = 100;
int* hostArray = new int[ ARRAY_SIZE ];

// Allocate memory on the device.
int* deviceArray = nullptr;
cudaMalloc( &deviceArray,
            ARRAY_SIZE * sizeof( int ) );

// Copy the array from the host to the
// device.
cudaMemcpy( deviceArray, hostArray,
            ARRAY_SIZE * sizeof( int ),
            cudaMemcpyHostToDevice );

// Clean up.
delete[] hostArray;
cudaFree( deviceArray );
```

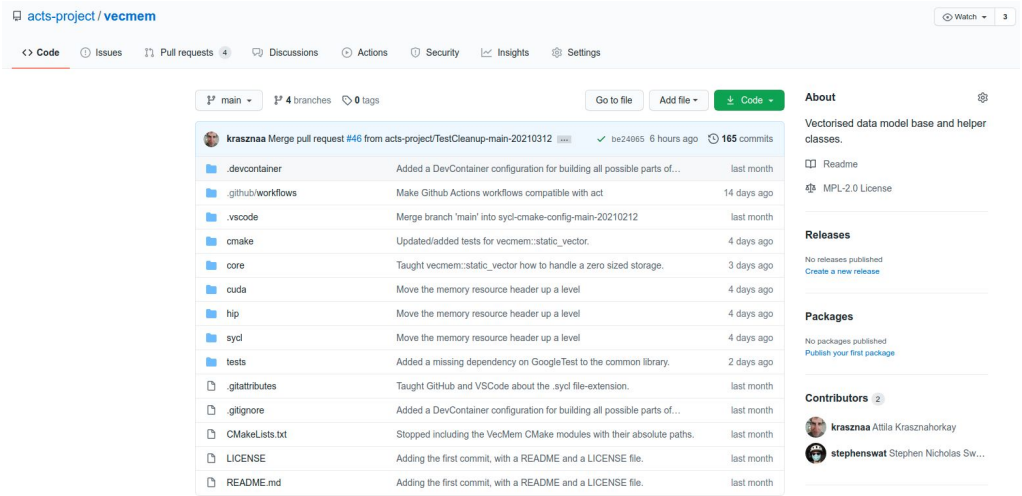
(Automatic) Memory Management

```
// Allocate managed memory.
static const std::size_t ARRAY_SIZE = 100;
int* array = nullptr;
cudaMallocManaged(
   &array, ARRAY_SIZE * sizeof( int ) );

// Use the memory from the host.
array[ 21 ] = 1.23f;

// Use the array in device code.
gpuCode<<< ARRAY_SIZE, 1 >>>(
   ARRAY_SIZE, array );

// Clean up.
cudaFree( array );
```

- All languages also support managing your memory for you
- In this setup the same amount of memory is allocated in “host” and “device” memory
  - At runtime memory copies are initiated through “page faults” when the CUDA runtime detects that the code is about to access memory that is not in sync “with the other side”
- Setting up your code like this during development is an excellent choice
  - Can cut down a lot on coding, when you’re mainly interested in developing your algorithm.
- But it provides much worse performance than explicit memory management in most cases!

Aided Memory Management

- Once you write a slightly larger piece of code, you should think of using code designed to help with memory management
  - Solutions exist in Kokkos, in Alpaka, and in many other places
- Things usually become complicated once you need jagged arrays in your code
  - Which happens to be a thing that I myself am currently involved in writing code for 😛



Device / Kernel Code

- Once you moved some data to your GPU, you want to do something with it
- You need to provide a function that would be started in all GPU threads
  - The function can ask for the identifiers of the thread that it is executing in, and perform its task accordingly
- Threads may be started for “invalid” identifiers as well!
  - Your code **must** check whether the ID that it has should be done anything for

```
// “Kernel” function
__global__
void gpuCode( std::size_t size,
              float* array ) {

   // Get the ID of the thread.
   const int index =
      blockIdx.x * blockDim.x + threadIdx.x;
   if( index >= size ) {
      return;
   }

   // Perform a task.
   array[ index ] *= 2;
}


   …
   gpuCode<<< ARRAY_SIZE, 1 >>>(
      ARRAY_SIZE, array );
   …
```

Error Checking

- Any CUDA function call can fail!
  - When they do, they tend to fail silently
- You must rigorously check the return codes of CUDA function calls!
  - Most conveniently by setting up a helper macro for it. Which can be as simple as:

```
/// Simple macro to run CUDA commands with
#define CUDA_CHECK( EXP )                                                    \
   do {                                                                      \
      const cudaError_t ce = EXP;                                            \
      if( ce != cudaSuccess ) {                                              \
         std::cerr << "Failed to execute: " << #EXP << std::endl;             \
         std::cerr << "Reason: " << cudaGetErrorString( ce ) << std::endl;    \
         return;                                                             \
      }                                                                      \
   } while( false )
```

- One big exception is a kernel launch, which returns nothing
  - You must use cudaGetLastError() or cudaPeekLastError() to detect any errors from a kernel launch

Asynchronous Execution

```
// Create a CUDA stream.
cudaStream_t stream = nullptr;
cudaStreamCreate( &stream );

// Allocate memory on the host.
static const std::size_t ARRAY_SIZE = 100;
int* hostArray = nullptr;
cudaMallocHost( &hostArray,
                ARRAY_SIZE * sizeof( int ));

// Allocate memory on the device.
int* deviceArray = nullptr;
cudaMalloc( &deviceArray,
            ARRAY_SIZE * sizeof( int ) );

// Copy the memory asynchronously.
cudaMemcpyAsync( deviceArray, hostArray,
                 cudaMemcpyHostToDevice,
                 stream );

// Launch a kernel asynchronously.
gpuCode<<< blocks, threadsPerBlock, 0,
           stream >>>( ARRAY_SIZE, array );
```

- Only mentioning it here to make you interested…
  - This goes a bit beyond what fits into this talk ☹
- In most cases you want your CPU and GPU to work in parallel
  - Executing “heavily branching” code on the CPU, and SIMT code on the GPU
- This is possible by launching memory copies and kernels asynchronously
- Generalising how a multi-threaded software framework can do this efficiently is one of the challenges in the LHC experiments…

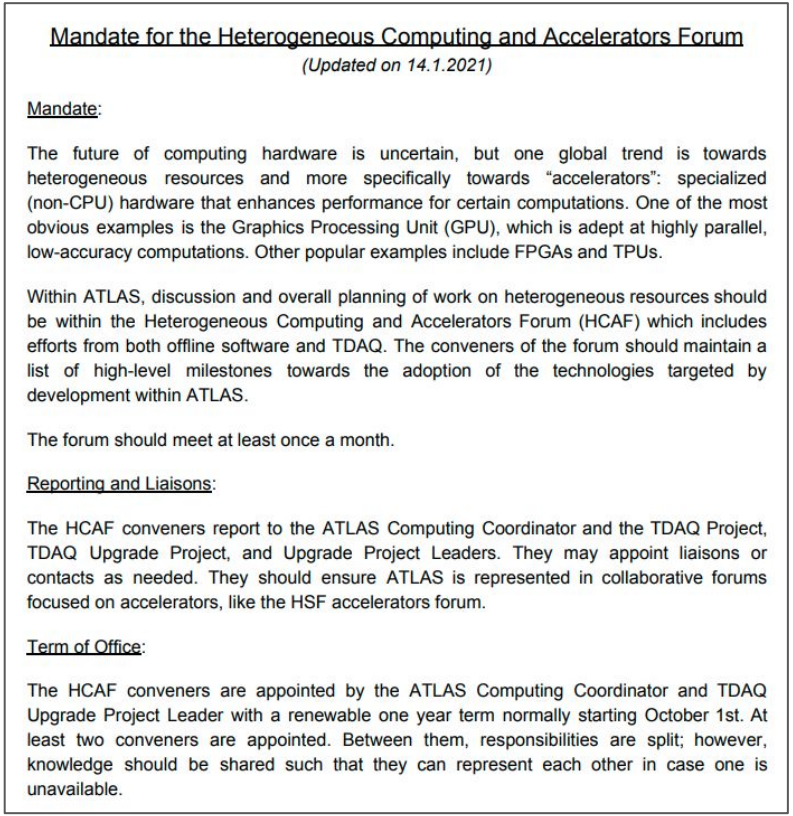
Code Management/Building

- Compiling a small program is easy enough in any language
  - However once you want to compile a large project with GPU support, things become a lot more complicated…
- In ATLAS -- and in HEP in general -- we use CMake to build our projects
  - It has excellent built-in support for CUDA. If you write your code in that, your life will be very easy.
    - You can provide CMake's add_library(...) / add_executable(...) / etc. functions with `.cu` files, and it will compile/link them ~correctly out of the box
  - If your project is "simple enough", you can just tell CMake to build all of your source files with hipcc / dpcpp for ROCm/HIP or oneAPI/SYCL projects
    - However in most cases this is not appropriate. In those cases, for now, you have to tell CMake very explicitly how it should build your source files.

ATLAS, LHC, HEP...

GPUs in ATLAS

- Previous organisational elements were recently merged into HCAF
  - We try to oversee all GPU/FPGA/etc. developments in the offline code in this forum
- Development is happening in a few different areas:
  - TDAQ is overseeing tracking and calo clustering developments
  - On the offline side a lot of effort is going into Acts
  - The Machine Learning forum is also becoming more and more active!



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.

GPUs at the LHC

- Every major LHC experiment is actively working on making use of GPUs
  - ALICE is ahead of everyone else by having used GPUs in production during LHC Run-2
  - CMS and LHCb will use GPUs during LHC Run-3 to varying degrees
  - ATLAS will keep GPUs as R&D platforms during LHC Run-3, possibly using them in production in HL-LHC
- CERN IT is actively working on making CERN hosted GPUs available for interactive and batch access
  - With the eventual goal being to be able to log into interactive nodes as easily as logging into lxplus for developing GPU code

GPUs in HEP

- As you may know yourself, detector simulation and event reconstruction will not be the only problems for HL-LHC
  - A big fraction of ATLAS's CPU budget is aimed at event generation. Developments in making use of accelerators in those is very important to all of HEP!
- Upcoming neutrino experiments may use GPUs very efficiently in their event reconstruction
  - That by itself is a very interesting area, but is happening mostly outside of CERN
- Discussions about all of these are taking place in various meetings of the HEP Software Foundation and the Compute Accelerator Forum

Additional Resources

- As said already, your best bet is to have a GPU “of your own” to develop code
  - But some resources do exist if you don’t have one
  - Intel DevCloud: Allows you to develop / run your code on Intel’s public cluster
  - I believed that the NVidia Developer Program membership offered something similar, but it doesn’t do it (any longer) ☹
  - CERN can already provide GPUs to those who request it, and things should get even easier during this year
- We will be holding an ATLAS GPU Tutorial during 25-28 May
  - https://indico.cern.ch/e/ATLAS_GPU_TRAINING
  - Unfortunately places are all filled up by now. But we will for sure have other tutorials in 2021 as well.
    - Sign up to atlas-sw-accelerators@cern.ch to learn about these amongst the first

Summary

- High Performance Computing will be built on “accelerators” for the foreseeable future
  - Learning how to write HEP code for them is a necessity
- Both the hardware and the software is evolving very rapidly
  - For “smaller” projects this should not be too much of a problem. But in projects like ATLAS’s offline software, we need to be very careful which programming model we start using.
- NVidia is king both with its hardware and software at the moment
  - AMD is developing its hardware very well, it may compete with NVidia on that front soon
  - Intel is very active in its software developments. They will likely strongly affect the future of the ISO C++ standard.
- If you are in ATLAS, and are interested in becoming involved in these software developments, contact us on atlas-sw-accelerators@cern.ch! 🙂

http://home.cern