Performance Portability for Heterogeneous Computing

Felice Pantaleo (EP-CMG-DS)

For the Patatrack Team

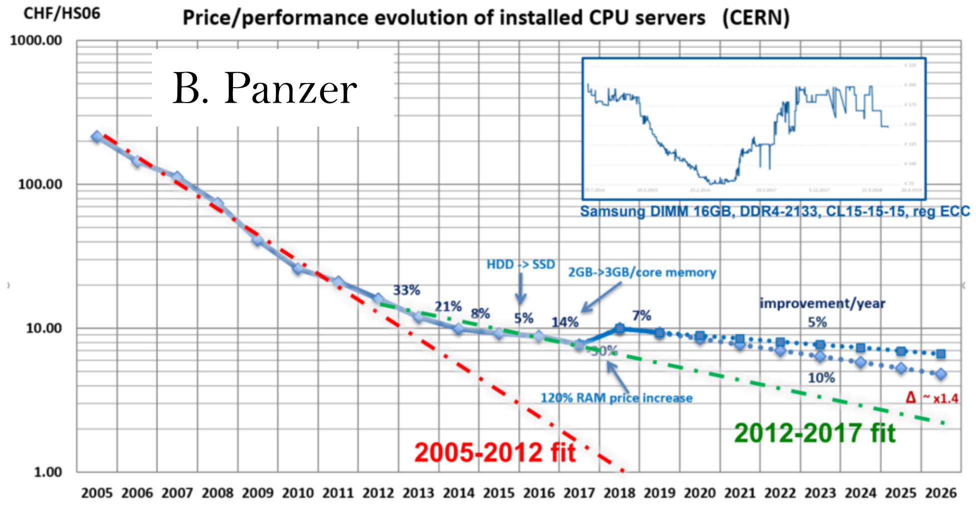
felice@cern.ch

Why are we caring?

- The Patatrack team has demonstrated a complete CMS Pixel reconstruction running on GPU:
  - on a NVIDIA T4 can achieve 50% higher performance than a full Skylake Gold node
  - NVIDIA T4 costs approx. 1/5 of a node
  - It is fully integrated in CMSSW and supports standard validation
  - It is written in CUDA for the GPU part, C++ for the CPU part
- Maintaining and testing two codebases might not be the most sustainable solution in the medium/long term
  - Not a showstopper at the moment, but will become one when we will transfer ownership of the code to the collaboration
- In the long term other accelerators might appear

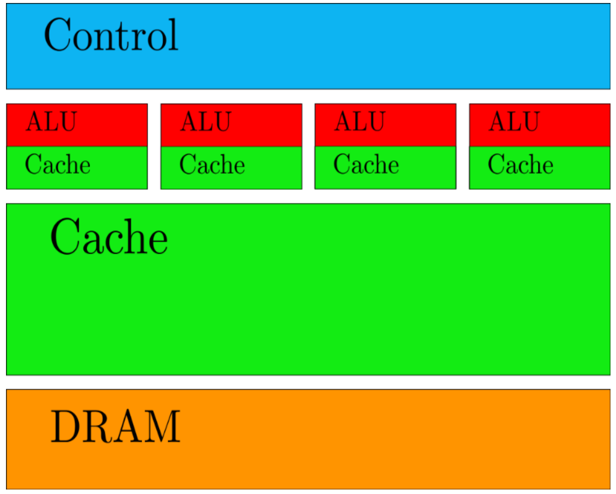
Why should our community care?

- Accelerators are becoming ubiquitous
  - Driven by more complex and deeper neural networks
  - Details hidden to the user by the FW
- Better Time-to-Solution, Energy-to-Solution, Cost-to-Solution
- Experiments are encouraged to run their software on Supercomputers
  - We are not using their GPUs
  - Summit: 190PFLOPS out of 200PFLOPS come from GPUs
- Training neural networks for production workflows is a negligible part
- Redesigning our algorithms and data structures to be well digested by a GPU can speed it up also when running on CPUs

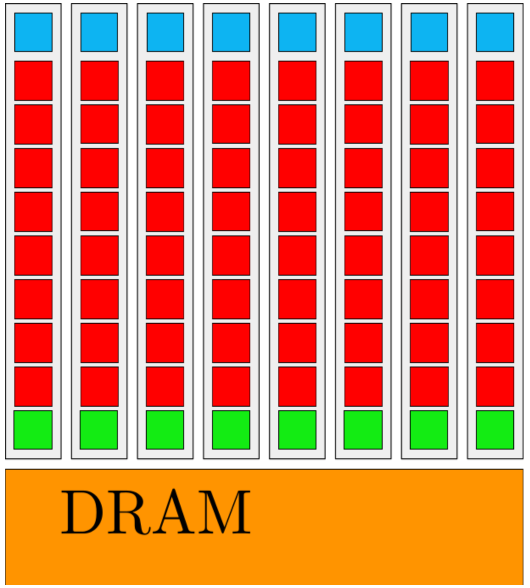


Architectures

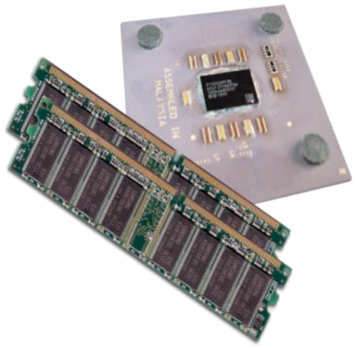
Control
ALU
Cache
DRAM



CPU



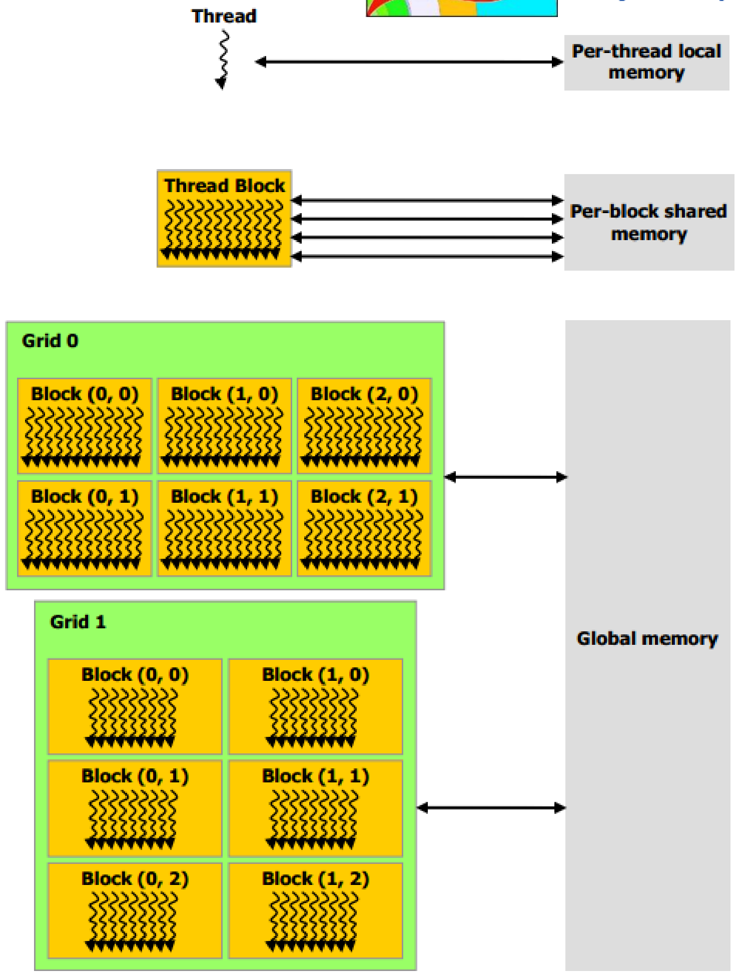
GPU



CUDA Programming model

A parallel kernel is launched on a grid of threads, grouped in blocks.

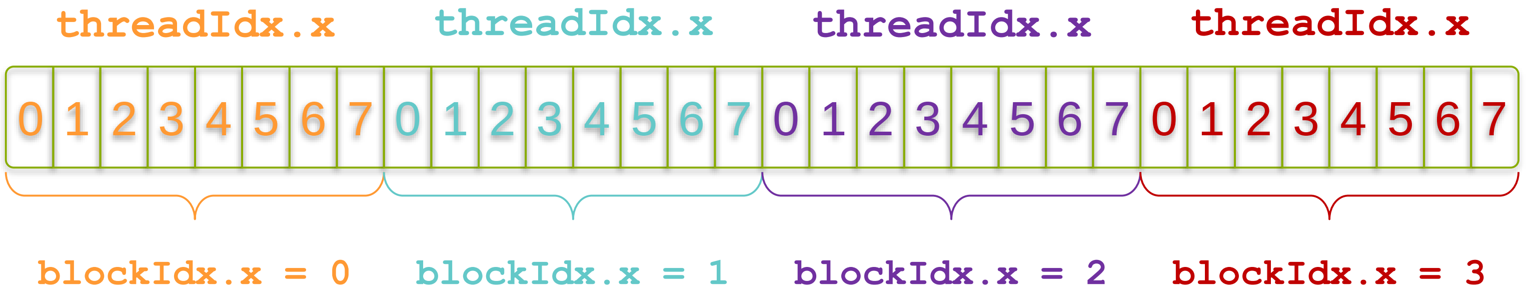
- All threads in the same block:
  - run on the same SM, in warps (SIMD)
  - can communicate
  - can synchronize



CUDA Kernels

Assign each thread a unique identifier and unroll the for loop.

For example:



```
__global__ void add(const int *a, const int *b,
                    int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

P != PP

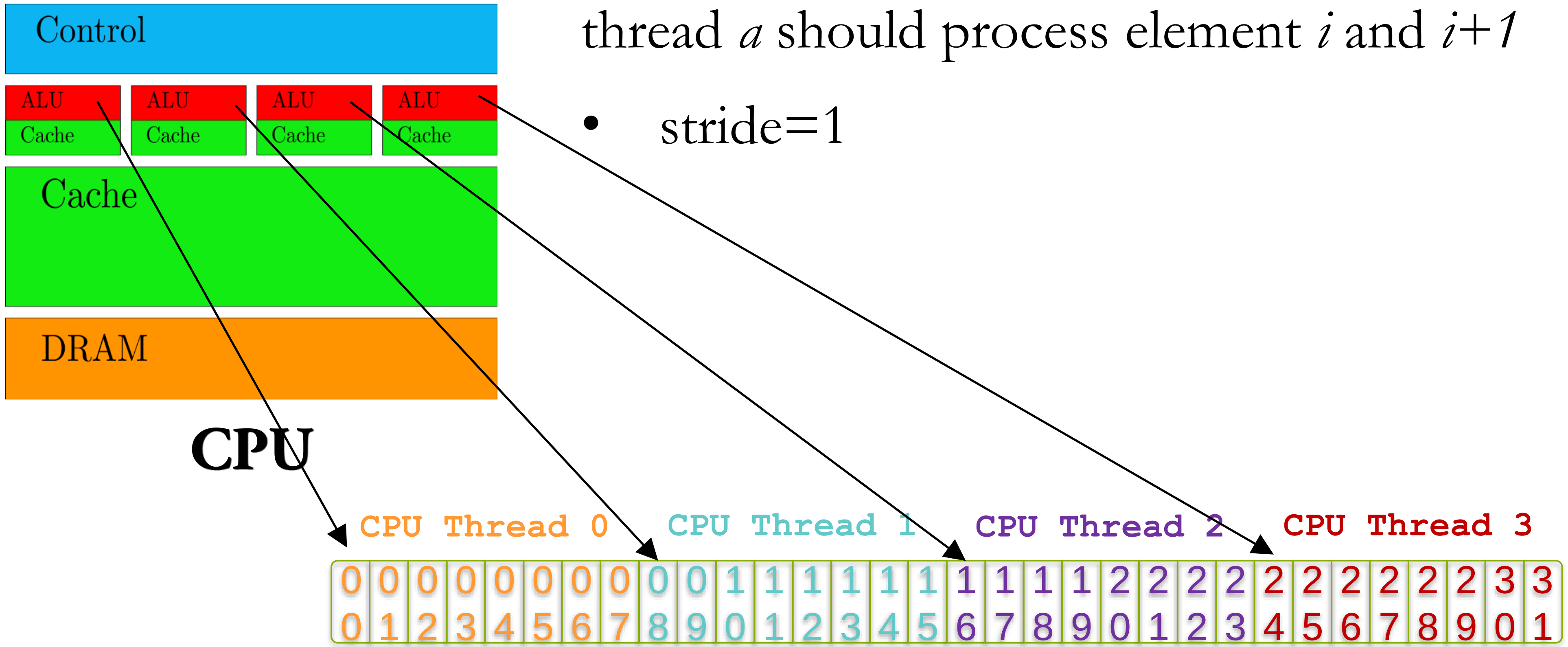
Portability could be achieved by blindly translating CUDA threads to, e.g., CPU threads or viceversa (plus some synchronization mechanism)

- You would not need to learn how a GPU works

Unfortunately, this is a terrible idea and will almost certainly lead you to poor performance

Portability does not imply Performance Portability

Memory access patterns: cached

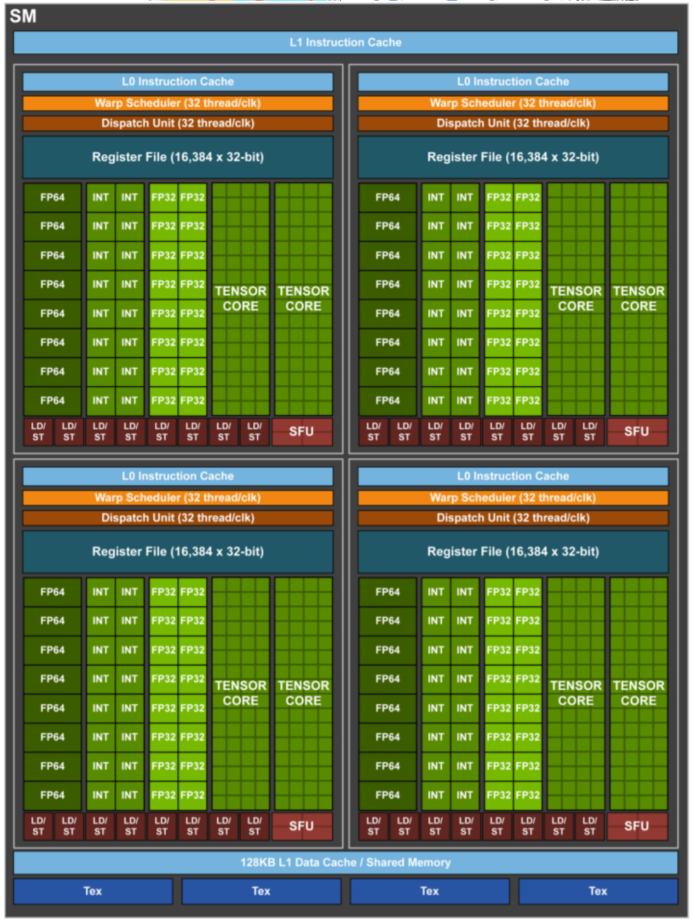


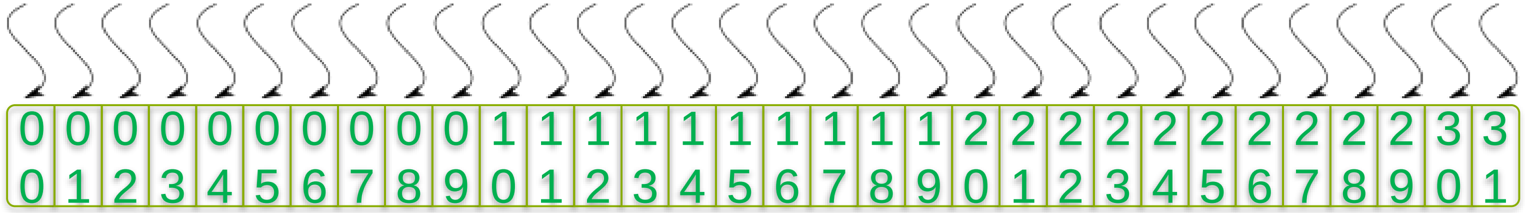
For optimal CPU cache utilization, the thread *a* should process element *i* and *i+1*

- stride=1

Inside a GPU SM: coalesced

- L1 data cache shared among ALUs
- ALUs work in SIMD mode in groups of 32 (warps)
- If a *load* is issued by each thread, they have to wait for all the loads in the same warp to complete before the next instruction can execute
- Coalesced memory access pattern optimal for GPUs: thread *a* should process element *i*, thread *a+1* the element and *i+1*
  - Lose an order of magnitude in performance if cached access pattern used on GPU


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex



Portability frameworks

OpenMP and OpenACC

- Portability programming models based on compiler directives
- Sensitive to compiler support and maturity
- Difficult coexistence with a tbb-based framework-scheduler

OpenCL -> SYCL -> OneAPI

- Initially The promise for portability, then became framework for portability between GPUs from different vendors, now supporting FPGAs
- While OpenCL did not support the combination of C++ host code and accelerator code in a single source file, SYCL does
    - This is a precondition for templated kernels which are required for policy based generic programming
- SYCL enables the usage of a single C++ template function for host and device code
- At the moment, OneAPI is SYCL

For all the above, if you need portable performance you have to manage memory and its layout yourself

Performance Portability frameworks

In the context of Patatrack R&D we have been recently looking into:

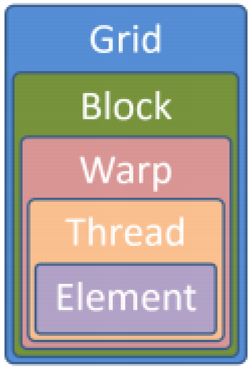
- Alpaka/Cupla: https://github.com/ComputationalRadiationPhysics/alpaka
    - Developed by Helmholtz-Zentrum Dresden – Rossendorf
        - Applications in Material science, XFEL, HPC
- Kokkos: https://github.com/kokkos/kokkos
    - Developed by Sandia National Lab, U.S. National Nuclear Security Administration

They provide an interface that hides the back-end implementation.

In the following, the assumption is that you already have a data-parallel code.

Alpaka abstraction hierarchy

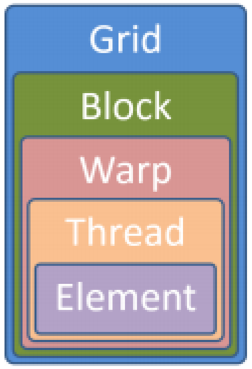
- multiple elements are processed per thread
- multiple threads are executed in lock-step within a warp
- multiple warps form independent blocks



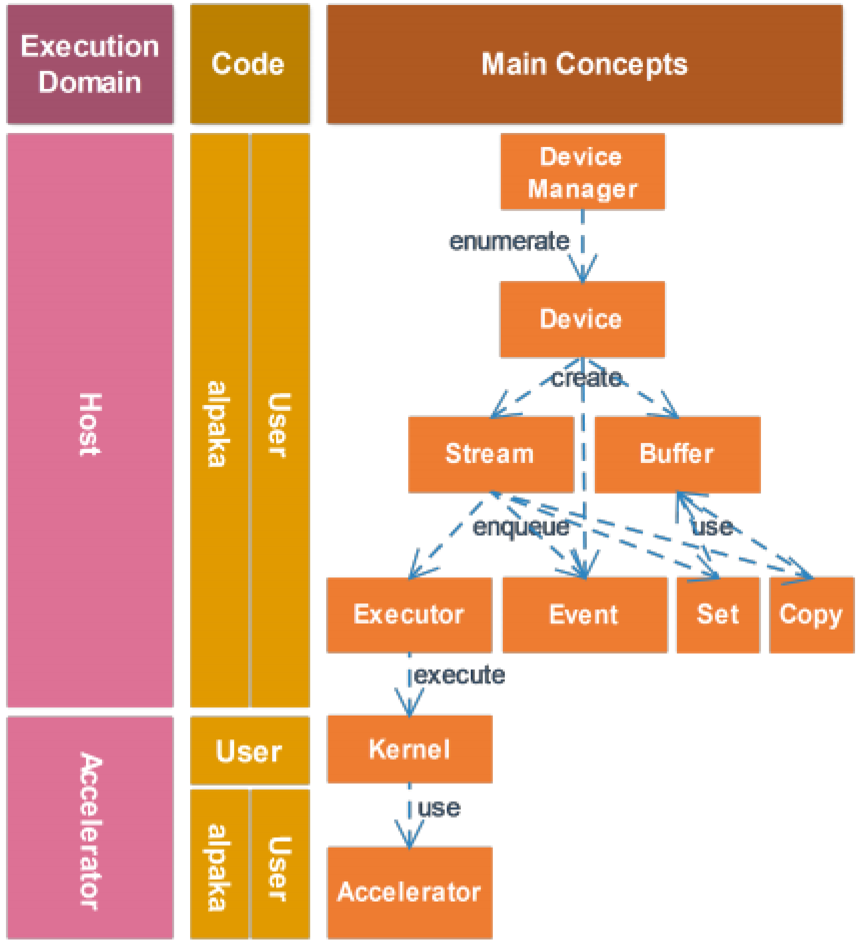
- Cupla was created because mapping the Alpaka's abstraction to CUDA is straightforward as the hierarchy levels are identical up to the element level.

Alpaka abstraction hierarchy to CPU

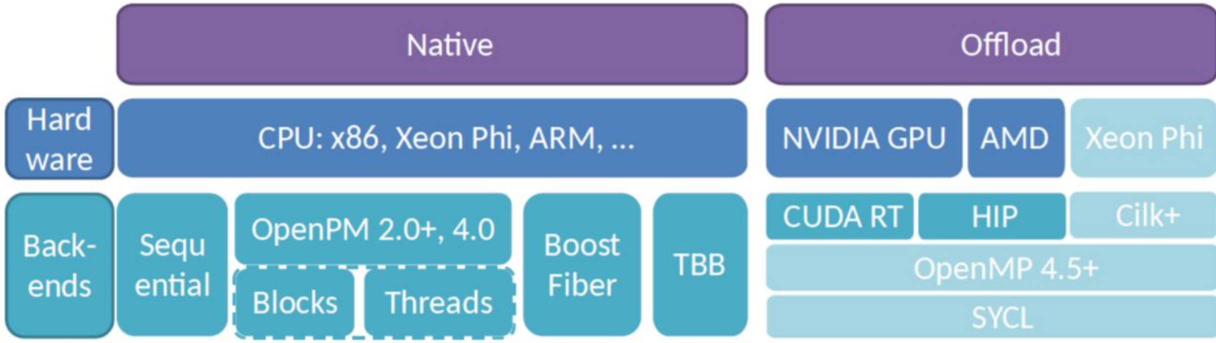
- On GPU, warps can handle branches with divergent control flows of multiple threads
    - There is no component on the CPU capable of this
    - 1to1 mapping of threads to warps
- Blocks cannot be mapped to the node nor socket
    - too much cache, memory, bus traffic
    - They are mapped to the cores
- Elements can be used to map CPU vector units



Alpaka/Cupla



```
struct kernel_compute_histogram {
  template <typename T_Acc>
  ALPAKA_FN_ACC
  void operator()(T_Acc const &acc, LayerTilesCupla<T_Acc> *d_hist,
      PointsPtr d_points, int numberOfPoints) const {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < numberOfPoints) {
      // push index of points into tiles
      d_hist[d_points.layer[i]].fill(d_points.x[i], d_points.y[i], i);
    }
  }
};
```



Kokkos

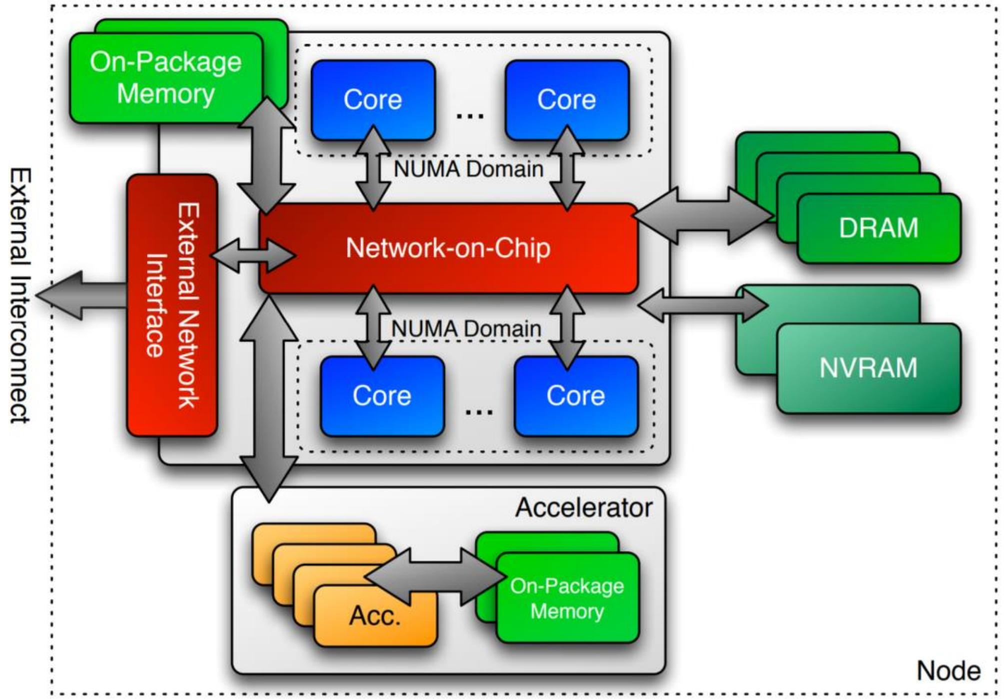
- Provides an abstract interface for portable, performant shared-memory programming

Supported backends:

- std::threads, OpenMP, Intel tbb
- CUDA, ROCm
- Offers `parallel_for`, `parallel_reduce`, `parallel_scan`, `task` to describe the pattern of the parallel tasks
- Multidimensional arrays with a neutral indexing and an architecture dependent layout are available
- Thread-safety issues: the most portable approach is for only one (non-Kokkos) thread of execution to control Kokkos

Kokkos Machine Model

- Kokkos assumes an *abstract machine model* , in which multiple processing devices can coexist and might share memory space



Kokkos Execution Policy

An execution policy determines how the threads are executed:

- sizes of blocks of threads
- static, dynamic scheduling

Range Policy: execute an operation once for each element in a range

Team Policy: *teams* of threads form a *league*

- *sync and shared memory* in same team
- Different teams can run different execution patterns (parallel_for, scan etc)
- Policies can be nested

You decide where to run the parallel kernel by specifying an Execution Space

```
parallel_for(
  RangePolicy< ExecutionSpace >(0,numberOfIntervals),
  [=] (const size_t i) {
    /* ... body ... */
  });
```

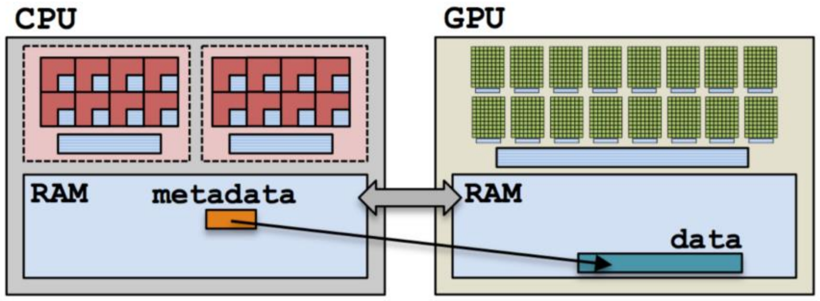
Kokkos Views

Multi-dimensional array of 0 or more dimensions, with sizes set at compile or run time

```
View<double ***, MemorySpace> data("label" , N0 , N1 , N2 ); 3 run, 0 compile

View<double **[N2], MemorySpace> data("label" , N0 , N1 ); 2 run, 1 compile

View<double *[N1][N2], MemorySpace> data("label" , N0 ); 1 run, 2 compile

View<double [N0][N1][N2], MemorySpace> data("label" ); 0 run, 3 compile
```

Specify MemorySpace to choose where to allocate the payload of the View

- HostSpace, CudaSpace, CudaUVMSpace…
- Mirroring/deep copy from one space to another possible
- Layout (row-/column-major) depends on the architecture for coalesced/cached memory access



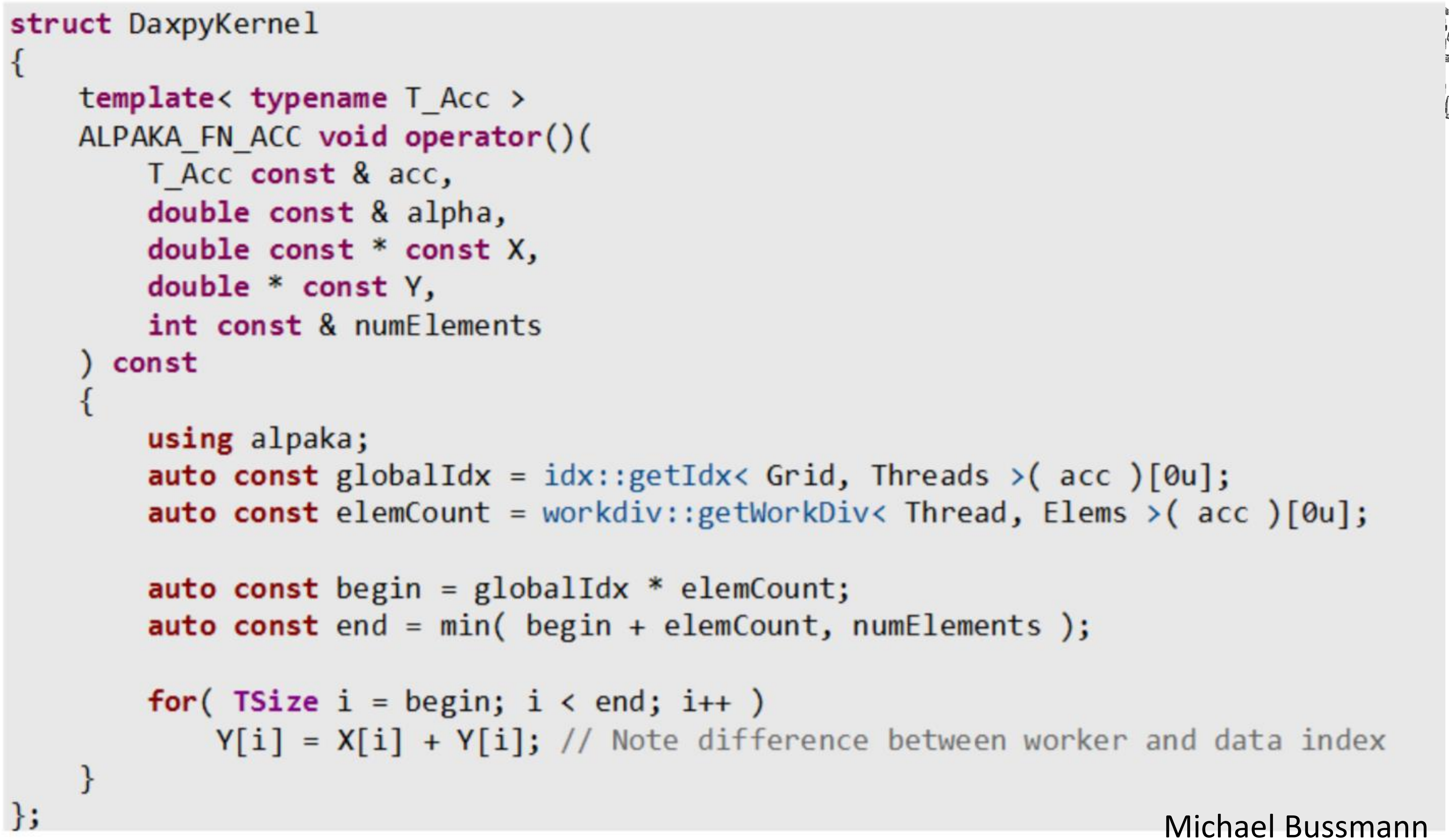
How Kokkos code looks like

```
Kokkos::View<Input, Kokkos::CudaSpace> input_d{"input_d"};
Kokkos::View<Input, Kokkos::CudaSpace>::HostMirror input_h = Kokkos::create_mirror_view(input_d);
std::memcpy(input_h.data(), &input, sizeof(Input));

Kokkos::View<Output, Kokkos::CudaSpace> output_d{"output_d"};
Kokkos::View<Output, Kokkos::CudaSpace>::HostMirror output_h = Kokkos::create_mirror_view(output_d);

auto start = std::chrono::high_resolution_clock::now();
Kokkos::deep_copy(input_d, input_h);

Kokkos::parallel_for(Kokkos::RangePolicy<Kokkos::Cuda>(0, input.wordCounter),
                     KOKKOS_LAMBDA (const size_t i) {
                       kokkos::rawtodigi(input_d, output_d, wordCounter,
                                         true, true, false, i);
  });
Kokkos::fence();
Kokkos::deep_copy(output_h, output_d);
Kokkos::fence();

auto stop = std::chrono::high_resolution_clock::now();
```

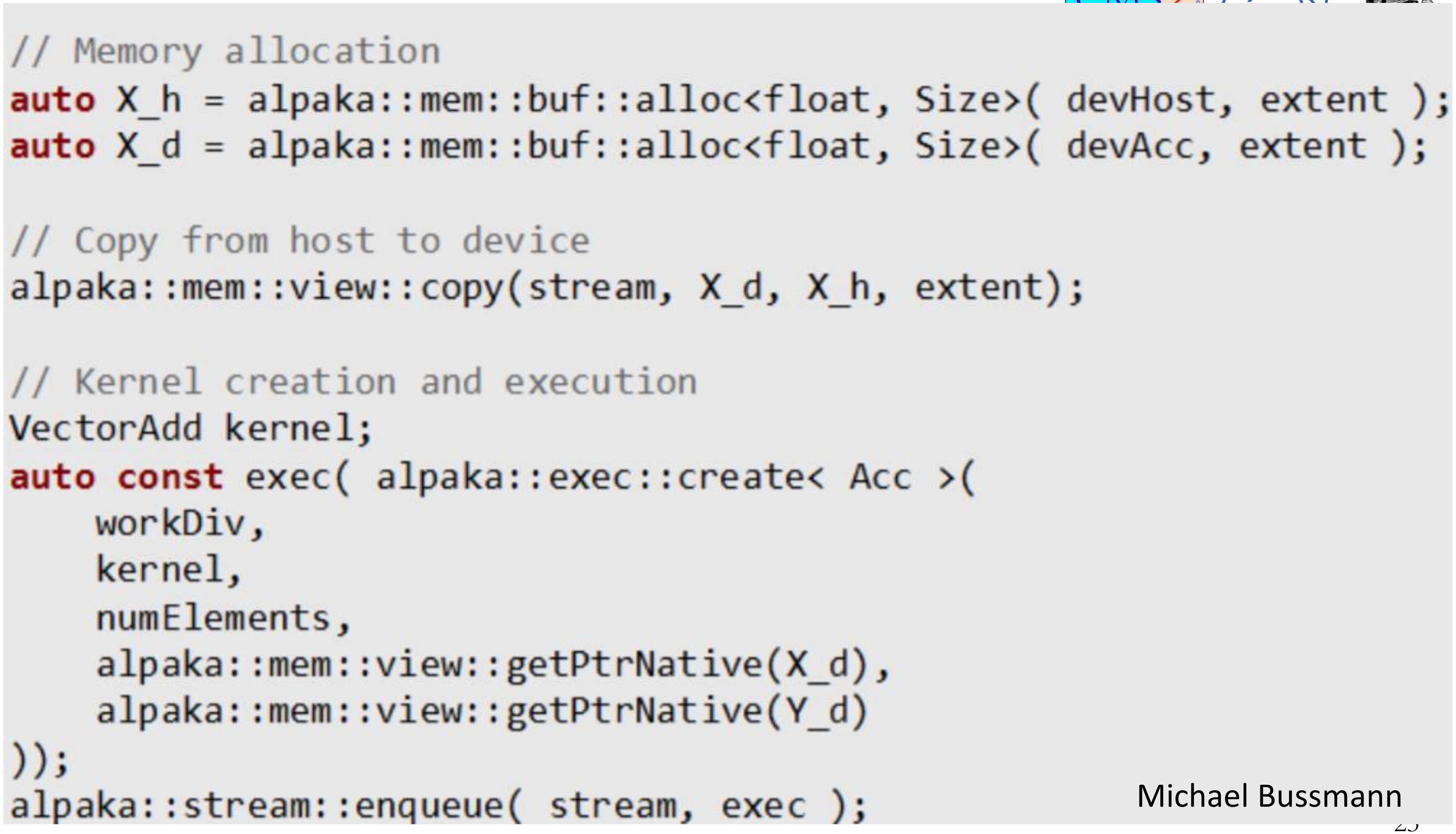
Conclusion

- Portable code is key for long-term maintainability, testability and support for new accelerator devices
- Many possible solutions, not so many viable ones, even less production ready
- Alpaka and Kokkos are very active teams and discussions/pull requests are ongoing
- Ongoing study and comparisons of solutions in Patatrack for CMS reconstruction
- Starting from a CUDA code makes life **much** easier

Backup



```
struct DaxpyKernel
{
    template< typename T_Acc >
    ALPAKA_FN_ACC void operator()(
        T_Acc const & acc,
        double const & alpha,
        double const * const X,
        double * const Y,
        int const & numElements
    ) const
    {
        using alpaka;
        auto const globalIdx = idx::getIdx< Grid, Threads >( acc )[0u];
        auto const elemCount = workdiv::getWorkDiv< Thread, Elems >( acc )[0u];

        auto const begin = globalIdx * elemCount;
        auto const end = min( begin + elemCount, numElements );

        for( TSize i = begin; i < end; i++ )
            Y[i] = X[i] + Y[i]; // Note difference between worker and data index
    }
};
```



```
// Memory allocation
auto X_h = alpaka::mem::buf::alloc<float, Size>( devHost, extent );
auto X_d = alpaka::mem::buf::alloc<float, Size>( devAcc, extent );

// Copy from host to device
alpaka::mem::view::copy(stream, X_d, X_h, extent);

// Kernel creation and execution
VectorAdd kernel;
auto const exec( alpaka::exec::create< Acc >(
    workDiv,
    kernel,
    numElements,
    alpaka::mem::view::getPtrNative(X_d),
    alpaka::mem::view::getPtrNative(Y_d)
));
alpaka::stream::enqueue( stream, exec );
```

Michael Bussmann