Porting to GPU, experience from Experimental-HEP

Dr. Tony Di Pilato
tony.dipilato@cern.ch
13th October 2022

The heterogeneous computing scenario

- CPUs development ***can no longer provide*** the highest computing performance that complex applications require
- GPUs have been largely considered as the suitable ***accelerators*** to exploit for fast and parallel operations, due to their optimized architecture
  - Parallel-thinking approach is key for the design of new algorithms
  - Complex deep neural networks can be trained in a few hours rather than a few days

A problematic variety of choices

- Several vendors…

A problematic variety of choices

- Several vendors…

… with different programming languages

NVIDIA CUDA

AMD ROCm

OpenCL

A problematic variety of choices

- Several vendors…

Code duplication, hard to maintain!

… with different programming languages

NVIDIA CUDA

AMD ROCm

OpenCL™

Performance portability

- Performance portability solutions have become an interesting solution
  - ➢ Write code once, compile for different backends at the same time, execute on target platform
  - ➢ Not all the technologies provide close-to-native backend performance
- Portable code can be maintain easily and support new accelerators
- R&D with the Patatrack team @ CERN mainly focuses on 3 APIs
  - ➢ Alpaka
  - ➢ Kokkos
  - ➢ SYCL OneAPI

Performance portability

- Performance portability solutions have become an interesting solution
  - ➢ Write code once, compile for different backends at the same time, execute on target platform
  - ➢ Not all the technologies provide close-to-native backend performance
- Portable code can be maintain easily and support new accelerators
- R&D with the Patatrack team @ CERN mainly focuses on 3 APIs
  - ➢ Alpaka
  - ➢ Kokkos
  - ➢ SYCL OneAPI

Chosen by the CMS experiment as the performance portability technology for Run 3

Performance portability

- Performance portability solutions have become an interesting solution
  - ➢ Write code once, compile for different backends at the same time, execute on target platform
  - ➢ Not all the technologies provide close-to-native backend performance
- Portable code can be maintain easily and support new accelerators
- R&D with the Patatrack team @ CERN mainly focuses on 3 APIs
  - ➢ Alpaka
  - ➢ Kokkos
  - ➢ SYCL OneAPI

Discarded by CMS for multiple reasons (lack of performance wrt alpaka, …)

Performance portability

- Performance portability solutions have become an interesting solution
  - Write code once, compile for different backends at the same time, execute on target platform
  - Not all the technologies provide close-to-native backend performance
- Portable code can be maintain easily and support new accelerators
- R&D with the Patatrack team @ CERN mainly focuses on 3 APIs
  - Alpaka
  - Kokkos
  - SYCL/OneAPI

Promising candidate on longer term, but still with several missing features

The alpaka portability library

- **A**bstraction **L**ibrary for **Pa**rallel **K**ernel **A**cceleration
  - Developed and maintained at **HZDR** (Helmholtz-Zentrum-Dresden-Rossendorf) and **CASUS** (Center for Advanced Systems Understanding)
- C++ header-only library
  - No need for installation
  - Currently on C++17 standard
- Supports a wide range of compilers (g++, clang, …)
- Several backends supported
  - CPU serial and parallel execution (std::thread or TBB)
  - NVIDIA GPU (CUDA)
  - AMD GPU (HIP/ROCm)
  - Intel GPU and FPGAs (SYCL) *under development*

https://github.com/alpaka-group/alpaka

Alpaka: what to know

- Programming strategy *inspired by* CUDA
  - Easy porting CUDA-to-alpaka
  - Same way of organizing the work division – ***Grids-Blocks-Threads*** + additional abstraction layer ***Elements*** that can be exploited for vectorization

Alpaka: what to know

- Programming strategy *inspired by* CUDA
  - Easy porting CUDA-to-alpaka
  - Same way of organizing the work division – ***Grids-Blocks-Threads*** + additional abstraction layer ***Elements*** that can be exploited for vectorization
- Performance is close to the native backend
  - No overhead wrt to native CUDA or HIP
  - The serial backend showed sometimes slightly better performance wrt to native CPU code (but might be because of alpaka's optimization of the code)

Alpaka: what to know

- Programming strategy *inspired by* CUDA
  - Easy porting CUDA-to-alpaka
  - Same way of organizing the work division – ***Grids-Blocks-Threads*** + additional abstraction layer ***Elements*** that can be exploited for vectorization
- Performance is close to the native backend
  - No overhead wrt to native CUDA or HIP
  - The serial backend showed sometimes slightly better performance wrt to native CPU code (but might be because of alpaka's optimization of the code)
- Alpaka objects behave like shared_ptrs → must be passed by value or const reference

Alpaka: what to know

- Programming strategy *inspired by* CUDA
  - Easy porting CUDA-to-alpaka
  - Same way of organizing the work division – ***Grids-Blocks-Threads*** + additional abstraction layer ***Elements*** that can be exploited for vectorization
- Performance is close to the native backend
  - No overhead wrt to native CUDA or HIP
  - The serial backend showed sometimes slightly better performance wrt to native CPU code (but might be because of alpaka's optimization of the code)
- Alpaka objects behave like shared_ptrs → must be passed by value or const reference
- native buffers (vectors, arrays, …) must be ported to alpaka buffers, which **don't have a default constructor**

Current developments with alpaka

- The standalone version of the CMS pixel track and vertices reconstruction ***has been fully ported to alpaka***
  - CMSSW-like framework
  - Optimized memory management through the caching allocator which reuses memory
  - Code is compiled once, and can be run on multiple backends while splitting the jobs between them
- The CLUE clustering algorithm for the future CMS-HGCAL detector has been integrated in the same type of framework and ported to alpaka
  - 2D version (published in 2020) has been ported from native cuda to alpaka
  - The new 3D version has been ported from native CMSSW serial implementation to alpaka → **Performance will be presented at ACAT late this month**

Current developments with alpaka

- alpaka is being easily integrated in CMSSW!
  - some refinements are still ongoing
  - we should be able to use it in production after the Christmas break

- A simple guide to port CUDA code to alpaka in CMSSW (and not only) can be found here
  - Some helper functions are user-defined in the testbed framework and not natively available (you can find them here, i.e. in ***alpakaMemory.h*** and ***alpakaWorkDiv.h***)

Sample code comparison (CPU-alpaka)

```cpp
class CLUEAlgoSerial {
public:
  // constructor
  CLUEAlgoSerial() = delete;
  explicit CLUEAlgoSerial(float const &dc,
                          float const &rhoc,
                          float const &outlierDeltaFactor,
                          uint32_t const &numberOfPoints)
      : dc_{dc}, rhoc_{rhoc}, outlierDeltaFactor_{outlierDeltaFactor} {}

  ~CLUEAlgoSerial() = default;

  void makeClusters(PointsCloud const &host_pc);

  PointsCloudSerial d_points;

  std::array<LayerTilesSerial, NLAYERS> hist_;

private:
  float dc_;
  float rhoc_;
  float outlierDeltaFactor_;

  void setup(PointsCloud const &host_pc);
};
```

```cpp
namespace ALPAKA_ACCELERATOR_NAMESPACE {

  class CLUEAlgoAlpaka {
  public:
    // constructor
    CLUEAlgoAlpaka() = delete;
    explicit CLUEAlgoAlpaka(float const &dc,
                            float const &rhoc,
                            float const &outlierDeltaFactor,
                            Queue stream,
                            uint32_t const &numberOfPoints)
        : d_points{stream, numberOfPoints},
          queue_{std::move(stream)},
          dc_{dc},
          rhoc_{rhoc},
          outlierDeltaFactor_{outlierDeltaFactor} {
      init_device();
    }

    ~CLUEAlgoAlpaka() = default;

    void makeClusters(PointsCloud const &host_pc);

    PointsCloudAlpaka d_points;

    LayerTilesAlpaka<Acc1D> *hist_;
    cms::alpakatools::VecArray<int, maxNSeeds> *seeds_;
    cms::alpakatools::VecArray<int, maxNFollowers> *followers_;

  private:
    Queue queue_;
    float dc_;
    float rhoc_;
    float outlierDeltaFactor_;

    std::optional<cms::alpakatools::device_buffer<Device, LayerTilesAlpaka<Acc1D>[]>> d_hist;
    std::optional<cms::alpakatools::device_buffer<Device, cms::alpakatools::VecArray<int, maxNSeeds>>> d_seeds;
    std::optional<cms::alpakatools::device_buffer<Device, cms::alpakatools::VecArray<int, maxNFollowers>[]>> d_followers;

    // private methods
    void init_device();

    void setup(PointsCloud const &host_pc);
  };
}  // namespace ALPAKA_ACCELERATOR_NAMESPACE
```

user-defined namespace that contain all the needed symbols (Platform , Device, Queue, BufferType)

Pointers to device memory passed to kernels

Executes the task (similar to cudaStream)

Missing default constructor

Sample code comparison (CUDA-alpaka)

```cpp
class CLUEAlgoCUDA {
public:
  // constructor
  CLUEAlgoCUDA() = delete;
  explicit CLUEAlgoCUDA(float const &dc, float const &rhoc, float const &outlierDeltaFactor, cudaStream_t stream)
      : d_points{stream}, dc_{dc}, rhoc_{rhoc}, outlierDeltaFactor_{outlierDeltaFactor}, stream_{stream} {
    init_device();
  }

  ~CLUEAlgoCUDA() = default;

  void makeClusters(PointsCloud const &host_pc);

  PointsCloudCUDA d_points;

  LayerTilesCUDA *hist_;
  cms::cuda::VecArray<int, maxNSeeds> *seeds_;
  cms::cuda::VecArray<int, maxNFollowers> *followers_;

private:
  float dc_;
  float rhoc_;
  float outlierDeltaFactor_;
  cudaStream_t stream_ = nullptr;
  cms::cuda::device::unique_ptr<LayerTilesCUDA[]> d_hist;
  cms::cuda::device::unique_ptr<cms::cuda::VecArray<int, maxNSeeds>> d_seeds;
  cms::cuda::device::unique_ptr<cms::cuda::VecArray<int, maxNFollowers>[]> d_followers;

  // private methods
  void init_device();

  void setup(PointsCloud const &host_pc);
};
```

```cpp
namespace ALPAKA_ACCELERATOR_NAMESPACE {

  class CLUEAlgoAlpaka {
  public:
    // constructor
    CLUEAlgoAlpaka() = delete;
    explicit CLUEAlgoAlpaka(float const &dc,
                            float const &rhoc,
                            float const &outlierDeltaFactor,
                            Queue stream,
                            uint32_t const &numberOfPoints)
        : d_points{stream, numberOfPoints},
          queue_{std::move(stream)},
          dc_{dc},
          rhoc_{rhoc},
          outlierDeltaFactor_{outlierDeltaFactor} {
      init_device();
    }

    ~CLUEAlgoAlpaka() = default;

    void makeClusters(PointsCloud const &host_pc);

    PointsCloudAlpaka d_points;

    LayerTilesAlpaka<Acc1D> *hist_;
    cms::alpakatools::VecArray<int, maxNSeeds> *seeds_;
    cms::alpakatools::VecArray<int, maxNFollowers> *followers_;

  private:
    Queue queue_;
    float dc_;
    float rhoc_;
    float outlierDeltaFactor_;

    std::optional<cms::alpakatools::device_buffer<Device, LayerTilesAlpaka<Acc1D>[]>> d_hist;
    std::optional<cms::alpakatools::device_buffer<Device, cms::alpakatools::VecArray<int, maxNSeeds>>> d_seeds;
    std::optional<cms::alpakatools::device_buffer<Device, cms::alpakatools::VecArray<int, maxNFollowers>[]>> d_followers;

    // private methods
    void init_device();

    void setup(PointsCloud const &host_pc);
  };
}  // namespace ALPAKA_ACCELERATOR_NAMESPACE
```

Sample code comparison (kernels)

```cpp
void KernelComputeHistogram(std::array<LayerTilesSerial, NLAYERS> &d_hist, PointsCloudSerial &points) {
  for (unsigned int i = 0; i < points.n; i++) {
    // push index of points into tiles
    d_hist[points.layer[i]].fill(points.x[i], points.y[i], i);
  }
};
```

CPU serial: loops over all the points

```cpp
__global__ void kernel_compute_histogram(LayerTilesCUDA* d_hist, pointsView* d_points, int numberOfPoints) {
  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);
  }
}  // kernel
```

GPU CUDA: each thread execute the same instruction with a different point

```cpp
struct KernelComputeHistogram {
  template <typename TAcc>
  ALPAKA_FN_ACC void operator()(const TAcc &acc,
                                LayerTilesAlpaka<Acc1D> *d_hist,
                                pointsView *d_points,
                                uint32_t const &numberOfPoints) const {
    // push index of points into tiles
    cms::alpakatools::for_each_element_in_grid(
        acc, numberOfPoints, [&](uint32_t i) { d_hist[d_points->layer[i]].fill(d_points->x[i], d_points->y[i], i); });
  }
};
```

Call KernelComputeHistogram() instead of using the <<< … >>> syntax

CPU/GPU alpaka: same as CUDA, with a user-defined helper function that accounts for the additional ***elements*** abstraction layer

The alpaka portability library

- The latest alpaka documentation can be found here
  - Cheatsheet available
  - Deeper explanation of the abstraction layers
  - Various details
- alpaka has been tested on
  - CPU (serial and TBB backends)
  - NVIDIA GPU (Tesla T4, V100, A10)
  - AMD GPU (preliminary tests, as AMD GPU are not supported yet in CMSSW)

and showed ***equivalent performance*** to the native backends

SYCL/oneAPI

- Abstraction layer for heterogeneous computing
  - Maintained by Khronos group
- Intel developed ***DPC++***, an open source project to introduce SYCL for LLVM and oneAPI
- Several implementations to support different backends
  - Intel CPUs, Intel GPUs, Intel FPGAs
  - NVIDIA GPU (experimental - tested)
  - AMD GPU (experimental)
- As for now, SYCL is **still in active development and not considered for complex applications**

SYCL: what to know

- Primary goal is to achieve closer convergence with ISO C++
  - In theory, all the host code can be compiled with a standard C++ compiler

SYCL: what to know

- Primary goal is to achieve closer convergence with ISO C++
  - In theory, all the host code can be compiled with a standard C++ compiler
- Performance can be higher than the native backend on CPU
  - SYCL optimizes CPU execution through TBB and vectorization

SYCL: what to know

- Primary goal is to achieve closer convergence with ISO C++
  - In theory, all the host code can be compiled with a standard C++ compiler
- Performance can be higher than the native backend on CPU
  - SYCL optimizes CPU execution through TBB and vectorization
- Official tool to support parallel programming on future (yet unreleased) Intel GPUs

SYCL: what to know

- Primary goal is to achieve closer convergence with ISO C++
  - In theory, all the host code can be compiled with a standard C++ compiler
- Performance can be higher than the native backend on CPU
  - SYCL optimizes CPU execution through TBB and vectorization
- Official tool to support parallel programming on future (yet unreleased) Intel GPUs
- Some C++ features are unavailable due to portability reasons (i.e. usage of function pointers or call virtual functions inside kernels)

SYCL: what to know

- Primary goal is to achieve closer convergence with ISO C++
  - In theory, all the host code can be compiled with a standard C++ compiler
- Performance can be higher than the native backend on CPU
  - SYCL optimizes CPU execution through TBB and vectorization
- Official tool to support parallel programming on future (yet unreleased) Intel GPUs
- Some C++ features are unavailable due to portability reasons (i.e. usage of function pointers or call virtual functions inside kernels)
- CUDA and HIP backends are still experimental

SYCL: what to know

- Primary goal is to achieve closer convergence with ISO C++
  - In theory, all the host code can be compiled with a standard C++ compiler
- Performance can be higher than the native backend on CPU
  - SYCL optimizes CPU execution through TBB and vectorization
- Official tool to support parallel programming on future (yet unreleased) Intel GPUs
- Some C++ features are unavailable due to portability reasons (i.e. usage of function pointers or call virtual functions inside kernels)
- CUDA and HIP backends are still experimental
- Programming strategy slightly different from CUDA and porting requires generally more efforts

SYCL: what to know

- Primary goal is to achieve closer convergence with ISO C++
  - In theory, all the host code can be compiled with a standard C++ compiler
- Performance can be higher than the native backend on CPU
  - SYCL optimizes CPU execution through TBB and vectorization
- Official tool to support parallel programming on future (yet unreleased) Intel GPUs
- Some C++ features are unavailable due to portability reasons (i.e. usage of function pointers or call virtual functions inside kernels)
- CUDA and HIP backends are still experimental
- Programming strategy slightly different from CUDA and porting requires generally more efforts
- Several hidden compiler flags that performs optimizations
  - In our experience, this sometimes leads to ***different and/or unexpected results*** wrt to native application

Current developments with SYCL

- The standalone version of the CMS pixel track and vertices reconstruction ***is being ported to SYCL***
  - Several issues to be solved and missing features
  - The CUDA-compatible compiler for SYCL (experimental) cannot be used in a complex framework (still investigating)
- The CLUE clustering algorithm has been integrated in the same type of framework and ported to SYCL
  - Only 2D version (published in 2020) has been ported from native cuda to SYCL for now
- A simple and experimental guide to port CUDA code to SYCL can be found here
- Tests have been made on Intel CPUs, unreleased Intel GPU and the same NVIDIA GPUs mentioned earlier in this talk

Sample code (kernels)

```
void kernel_compute_histogram(LayerTilesSYCL *d_hist, view *d_points, int const &numberOfPoints, sycl::nd_item<1> item) {
  int i = item.get_group(0) * item.get_local_range().get(0) + item.get_local_id(0);
  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);
  }
}
```

SYCL: Work division is similar to CUDA, but different syntax

Sample code comparison (kernel launch)

```
kernel_compute_histogram<<<gridSize, blockSize, 0, stream_>>>(d_hist.get(), d_points.view(), host_pc.x.size());
```

CUDA baseline

```
auto WorkDiv1D = cms::alpakatools::make_workdiv<Acc1D>(gridSize, blockSize);
alpaka::enqueue(
    queue_,
    alpaka::createTaskKernel<Acc1D>(WorkDiv1D, KernelComputeHistogram(), hist_, d_points.view(), d_points.n));
```

alpaka: kernels are enqueued in task objects

```
(*queue_).submit([&](sycl::handler &cgh) {
  //SYCL kernels cannot capture by reference - need to reassign pointers inside the submit to pass by value
  auto d_hist_kernel = d_hist;
  auto num_points_kernel = d_points.n;
  cgh.parallel_for(sycl::nd_range<1>(gridSize * blockSize, blockSize), [=](sycl::nd_item<1> item) {
    kernel_compute_histogram(d_hist_kernel, d_points_view, num_points_kernel, item);
  });
});
```

SYCL: kernel launched as a lambda

Final considerations

- Performance portability libraries are a ***must-seeked solution*** for large HPC centers
  - Nowadays, it is impossible to get the very same high-performance device on each machine of a cluster due to many reasons
  - Intel is coming into the "GPU-game" in the near future

- The impression that we got in the last year(s) of testing is that the best option (if NVIDIA GPUs are being used) is to currently use alpaka for several reasons:
  1. Can also exploit AMD GPUs in heterogeneous data centers
  2. Its syntax is easy to understand if CUDA is known, and porting is relatively easy
  3. SYCL backend is also under development, so alpaka ***is expected to support also*** Intel FPGAs and GPUs in future

Final considerations

- Is it worth considering SYCL into the game? **YES**
- The main reason is that the team maintaining SYCL is consolidated and has a stable financial support for the future (Intel is somewhere there, behind the curtain)
- The alpaka team relies on a few people (although funds are secured until 2027 at HZDR and 2038 at CASUS) and sometimes struggles to support users' requests
  - In HEP (in particular CMS @ CERN), **Dr. Andrea Bocci** is the ***most qualified person*** to give additional information about alpaka and its usage in CMSSW. He often open requests/issues as a user and have active discussions with the alpaka team (and sometimes implements the needed features himself).