HEP-CCE PPS Weekly Meeting – 29 January 2021

Introduction to alpaka

Jan Stephan <j.stephan@hzdr.de>

Introduction to alpaka

Introduction to alpaka

alpaka – <u>A</u>bstraction <u>L</u>ibrary for <u>Pa</u>rallel <u>K</u>ernel <u>A</u>cceleration

Alpaka is...

- ... a parallel programming library: Accelerate your code by exploiting your hardware's parallelism!

- ... an abstraction library: Create portable code that runs on CPUs, GPUs and FPGAs!

- ... free & open-source software

Introduction to alpaka

Programming with alpaka

- C++ only!

- Header-only library: No additional runtime dependency introduced

- Modern library: alpaka is written entirely in C++14 (transitioning to C++17 soon)

- Supports a wide range of modern C++ compilers (g++, clang++, Apple clang, MS Visual Studio)

- Portable across operating systems: Linux, macOS, Windows are supported

Introduction to alpaka

Basic concepts

- Full control for the user, everything is explicit

  → no hidden allocations, copies, ...

- Everything is a concept / trait (through C++ templates)

  → high degree of customization possible

- Abstractions are resolved at compile time

  → longer compilation time, but very close to native code performance

- CUDA inheritance: work division defined through grids, blocks, warps & threads

- Additional work division layer: data elements per thread (used for auto-vectorization)

Introduction to alpaka

AXPY

$$\vec{y} \leftarrow a \cdot \vec{x} + \vec{y}$$

```cpp
using namespace alpaka;

struct AxpyKernel
{
    template <typename TAcc>
    ALPAKA_FN_ACC void operator()(TAcc const& acc,
      std::size_t numElements, int a, int const* X, int* Y) const
    {
        auto gridThreadIdx = getIdx<Grid, Threads>(acc)[0u];
        auto threadElems = getWorkDiv<Thread, Elems>(acc)[0u];
        auto first = gridThreadIdx * threadElems;

        if(first < numElements)
        {
            auto last = first + threadElems;
            for(auto i = first; i < last; ++i)
                Y[i] = a * X[i] + Y[i];
        }
    }
};
```

Introduction to alpaka

AXPY

$$\vec{y} \leftarrow \alpha \cdot \vec{x} + \vec{y}$$

```cpp
using namespace alpaka;
using Dim = DimInt<1u>;
using Idx = std::size_t;
using Acc = AccGpuCudaRt<Dim, Idx>;
```

Introduction to alpaka

AXPY

$$\vec{y} \leftarrow a \cdot \vec{x} + \vec{y}$$

```cpp
using namespace alpaka;
using Dim = DimInt<1u>;
using Idx = std::size_t;
using Acc = AccGpuCudaRt<Dim, Idx>;

auto const host = getDevByIdx<DevCpu>(0u);
auto const dev = getDevByIdx<Acc>(0u);
using myQueue = Queue<Acc, property::Blocking>;
auto queue = myQueue{dev};
```

Introduction to alpaka

AXPY

$$\vec{y} \leftarrow \alpha \cdot \vec{x} + \vec{y}$$

```cpp
using namespace alpaka;
using Dim = DimInt<1u>;
using Idx = std::size_t;
using Acc = AccGpuCudaRt<Dim, Idx>;

auto const host = getDevByIdx<DevCpu>(0u);
auto const dev = getDevByIdx<Acc>(0u);
using myQueue = Queue<Acc, property::Blocking>;
auto queue = myQueue{dev};
```

```cpp
auto const ext = Vec<Dim, Idx>{1024};
auto hostBufX = allocBuf<int, Idx>(host, ext);
/* Initialize … */
auto devBufX = allocBuf<int, Idx>(dev, ext);
```

Introduction to alpaka

AXPY

$$\vec{y} \leftarrow a \cdot \vec{x} + \vec{y}$$

```cpp
using namespace alpaka;
using Dim = DimInt<1u>;
using Idx = std::size_t;
using Acc = AccGpuCudaRt<Dim, Idx>;

auto const host = getDevByIdx<DevCpu>(0u);
auto const dev = getDevByIdx<Acc>(0u);
using myQueue = Queue<Acc, property::Blocking>;
auto queue = myQueue{dev};

auto const ext = Vec<Dim, Idx>{1024};
auto hostBufX = allocBuf<int, Idx>(host, ext);
/* Initialize … */
auto devBufX = allocBuf<int, Idx>(dev, ext);

memcpy(queue, devBufX, hostBufX, ext); // namespace alpaka
```

Introduction to alpaka

AXPY

$$\vec{y} \leftarrow \alpha \cdot \vec{x} + \vec{y}$$

```cpp
using namespace alpaka;
using Dim = DimInt<1u>;
using Idx = std::size_t;
using Acc = AccGpuCudaRt<Dim, Idx>;

auto const host = getDevByIdx<DevCpu>(0u);
auto const dev = getDevByIdx<Acc>(0u);
using myQueue = Queue<Acc, property::Blocking>;
auto queue = myQueue{dev};

auto const ext = Vec<Dim, Idx>{1024};
auto hostBufX = allocBuf<int, Idx>(host, ext);
/* Initialize … */
auto devBufX = allocBuf<int, Idx>(dev, ext);

memcpy(queue, devBufX, hostBufX, ext);
```

```cpp
auto workDiv = getValidWorkDiv<Acc>(dev, ext, Idx{1u});
auto taskKernel = createTaskKernel<Acc>(
                  workDiv, AxpyKernel{}, /* params … */);
enqueue(queue, taskKernel);
```

Introduction to alpaka

AXPY

$$\vec{y} \leftarrow \alpha \cdot \vec{x} + \vec{y}$$

```cpp
using namespace alpaka;
using Dim = DimInt<1u>;
using Idx = std::size_t;
using Acc = AccGpuCudaRt<Dim, Idx>;

auto const host = getDevByIdx<DevCpu>(0u);
auto const dev = getDevByIdx<Acc>(0u);
using myQueue = Queue<Acc, property::Blocking>;
auto queue = myQueue{dev};

auto const ext = Vec<Dim, Idx>{1024};
auto hostBufX = allocBuf<int, Idx>(host, ext);
/* Initialize … */
auto devBufX = allocBuf<int, Idx>(dev, ext);

memcpy(queue, devBufX, hostBufX, ext);

auto workDiv = getValidWorkDiv<Acc>(dev, ext, Idx{1u});
auto taskKernel = createTaskKernel<Acc>(
                    workDiv, AxpyKernel{}, /* params … */);
enqueue(queue, taskKernel);

memcpy(queue, hostBufX, devBufX, ext);
```
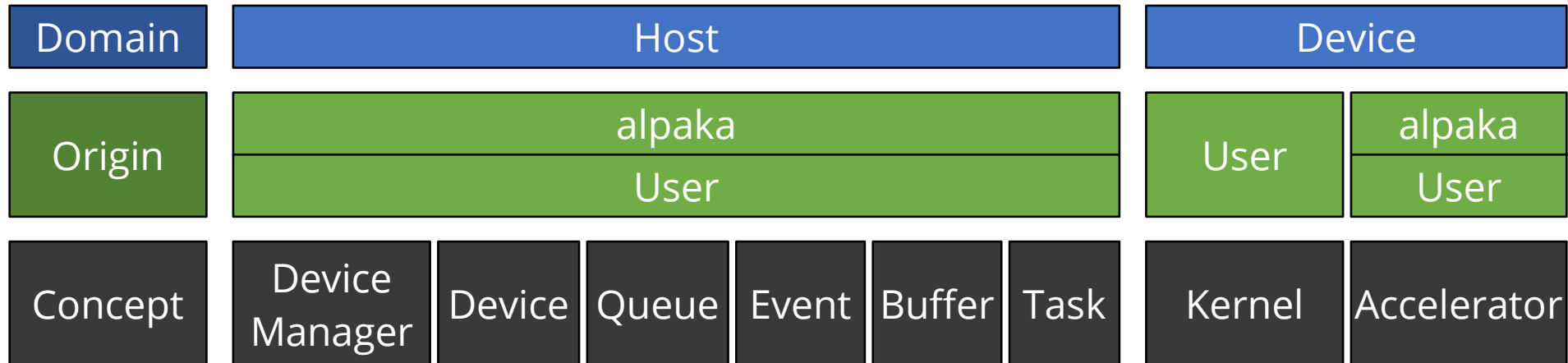
Introduction to alpaka

alpaka's structure

Domain	Host						Device	
Origin	alpaka						User	alpaka
	User							User
Concept	Device Manager	Device	Queue	Event	Buffer	Task	Kernel	Accelerator

Introduction to alpaka

Vector addition: alpaka CUDA PTX

```
mov.u32      %r3, %ctaid.x;
mov.u32      %r4, %ntid.x;
mov.u32      %r5, %tid.x;
mad.lo.s32  %r1, %r4, %r3, %r5;
setp.ge.s32 %p1, %r1, %r2;
@%p1 bra   BB6_2;

cvta.to.global.u64 %rd3, %rd2;
cvta.to.global.u64 %rd4, %rd1;
mul.wide.s32       %rd5, %r1, 8;
add.s64            %rd6, %rd4, %rd5;
ld.global.f64      %fd2, [%rd6];
add.s64            %rd7, %rd3, %rd5;
ld.global.f64      %fd3, [%rd7];
fma.rn.f64         %fd4, %fd2, %fd1, %fd3;
st.global.f64      [%rd7], %fd4;
```

Vector addition: native CUDA PTX

```
mov.u32      %r3, %ctaid.x;
mov.u32      %r4, %ntid.x;
mov.u32      %r5, %tid.x;
mad.lo.s32  %r1, %r4, %r3, %r5;
setp.ge.s32 %p1, %r1, %r2;
@%p1 bra   BB6_2;

cvta.to.global.u64 %rd3, %rd2;
cvta.to.global.u64 %rd4, %rd1;
mul.wide.s32       %rd5, %r1, 8;
add.s64            %rd6, %rd4, %rd5;
ld.global.nc.f64   %fd2, [%rd6];
add.s64            %rd7, %rd3, %rd5;
ld.global.f64      %fd3, [%rd7];
fma.rn.f64         %fd4, %fd2, %fd1, %fd3;
st.global.f64      [%rd7], %fd4;
```

Introduction to alpaka

Supported platform	Supported hardware
NVIDIA CUDA	NVIDIA GPUs
AMD HIP	AMD GPUs
	NVIDIA GPUs
OpenMP	OpenMP >= 2.0: x86, OpenPOWER, ARM
	OpenMP >= 5.0: Offloading targets
OpenACC >= 2.0	Offloading targets
SYCL (experimental)	oneAPI hardware (CPUs, GPUs, FPGAs)
	Xilinx FPGAs
Additional CPU backends	CPUs

Introduction to alpaka

alpaka is targeting advanced users!

- Template metaprogramming & specialization result in high degree of flexibility

- alpaka has a steep learning curve!

- Good knowledge of C++ templates required

- Explicitness leads to verbosity

- Example: Use alpaka kernels to speed up performance-critical sections of smaller programs

- Example: HPC experts provide an alpaka-based library of common algorithms to scientists

Introduction to alpaka

alpaka is free software (MPL 2.0). Find us on GitHub!

Our GitHub organization: https://www.github.com/alpaka-group

- Contains all alpaka-related projects, documentation, samples, …

- New contributors welcome!

The library: https://www.github.com/alpaka-group/alpaka

- Full source code, issue tracker, installation instructions, examples, …

Detailed YouTube video series: https://bit.ly/33MYZSQ

- Slides: **https://www.github.com/alpaka-group/alpaka-workshop-slides/tree/200629_cern**

Introduction to alpaka

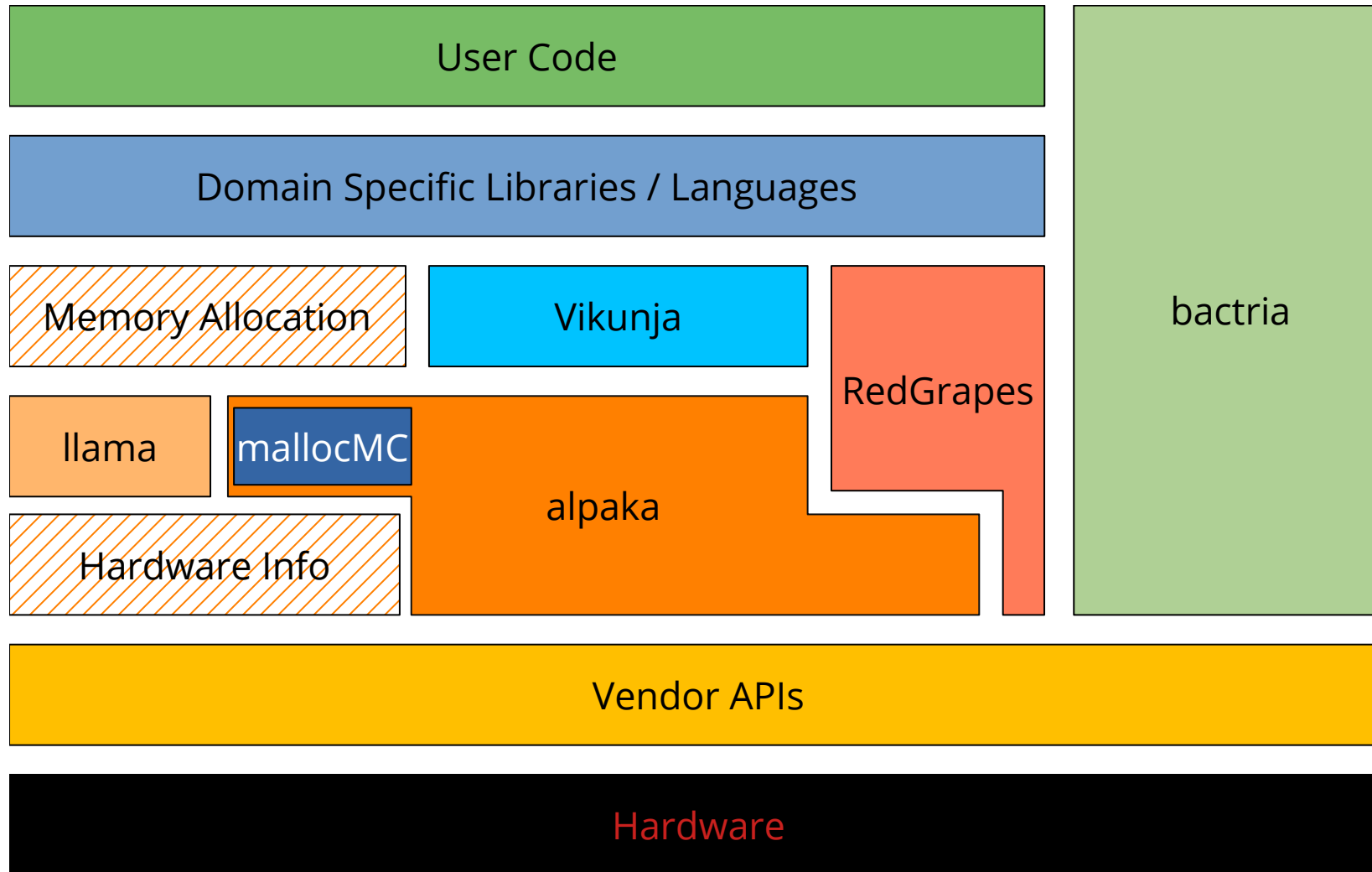**If you use alpaka for your research, please cite one of the following publications:**

Matthes A., Widera R., Zenker E., Worpitz B., Huebl A., Bussmann M. (2017): Tuning and Optimization for a Variety of Many-Core Architectures Without Changing a Single Line of Implementation Code Using the Alpaka Library. In: Kunkel J., Yokota R., Taufer M., Shalf J. (eds) High Performance Computing. ISC High Performance 2017. Lecture Notes in Computer Science, vol 10524. Springer, Cham, DOI: `10.1007/978-3-319-67630-2_36`.

E. Zenker et al., "Alpaka – An Abstraction Library for Parallel Kernel Acceleration", 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Chicago, IL, 2016, pp. 631 – 640, DOI: `10.1109/IPDPSW.2016.50`.

Worpitz, B. (2015, September 28). Investigating performance portability of a highly scalable particle-in-cell simulation code on various multi-core architectures. Zenodo. DOI: `10.5281/zenodo.49768`.

alpaka's Ecosystem

The alpaka ecosystem

cupla – The CUDA portability layer

```
#include <cuda_runtime.h>

/* CUDA API calls */

kernel<<<blocks, threads>>>(/* params */);
```

```
#include <cuda_to_cupla.h>

/* CUDA API calls */

CUPLA_KERNEL(kernel)(blocks, threads)
                               (/* params */);
```

- Use cupla (C++ User interface for the Platform independent Library Alpaka) to make existing CUDA codes portable!

- cupla maps CUDA API calls to the corresponding alpaka API calls

cupla – The CUDA portability layer

```cpp
__global__ void kernel(/* params */)
{
    /* … */
}
```

```cpp
struct kernel
{
    template <typename TAcc>
    ALPAKA_FN_ACC
    void operator()(TAcc const& acc,
                    /* params */)
    {
        /* … */
    }
};
```

- CUDA kernels need to be transformed into alpaka kernels

- Porting guide: **https://github.com/alpaka-group/cupla/blob/master/doc/PortingGuide.md**

- Tuning guide: **https://github.com/alpaka-group/cupla/blob/master/doc/TuningGuide.md**

cupla – The CUDA portability layer

cupla is free software (LGPL 3.0). Find us on GitHub!

The library: https://www.github.com/alpaka-group/cupla

- Full source code

- Issue tracker

- Installation instructions

- Small examples

LLAMA – Low Level Abstraction of Memory Access

Problem: representation of data is tied to physical layout

- Array of structs:

```
struct P { float x; float y; } p[8];
p[i].x = 3.14f;
func(p[i]);
```

- Struct of arrays:

```
struct P { float x[8]; float y[8]; } p;
p.x[i] = 3.14f;
func(p.x[i], p.y[i]);
```

- Changing the data layout requires changing the algorithm

- Different hardware needs different data layouts to perform well

LLAMA – Low Level Abstraction of Memory Access

Solution: Separate algorithmic data view and physical data layout

- LLAMA (Low Level Abstraction of Memory Access)

- Switch between memory layouts without touching the algorithm

- Header-only, portable C++17 library

- Supports CUDA >= 11

- Orthogonal to alpaka

LLAMA – Low Level Abstraction of Memory Access

```cpp
struct Vec
{
    float x;
    float y;
    float z;
};

struct Particle
{
    Vec pos;
    Vec vel;
    float mass;
};
```

- LLAMA builds a tree of abstract tags

- Leaf elements carry final type used for computation

```cpp
struct Pos{};
struct Vel{};
struct X{};
struct Y{};
struct Z{};
struct Mass{};

using Vec = llama:DS<
    llama::DE<X, float>,
    llama::DE<Y, float>,
    llama::DE<Z, float>>;

using Particle = llama::DS<
    llama::DE<Pos, Vec>,
    llama::DE<Vel, Vec>,
    llama::DE<Mass, float>>;
```

LLAMA – Low Level Abstraction of Memory Access

- Define problem size using `ArrayDomain`

- Compile-time dimensionality, runtime extents

- Choose desired memory mapping
  - LLAMA supports AoS, SoA and AoSoA mappings

- Allocate memory
  - LLAMA supports allocators based on std::vector (default), std::shared_ptr and stack memory

- Access data as needed
  - LLAMA provides a rich interface for accessing data

```cpp
// Define problem size
llama::ArrayDomain domain{dimX, dimY};

// decltype(domain) is:
llama::ArrayDomain<2>;

// Choose Array-of-structs mapping
llama::mapping::AoS map{domain, Particle{}};

// Allocate memory with default allocator
auto view = llama::allocView(mapping);

// Access first particle
auto p = view(pos);

// Access particle information
float& x = p(Pos{}, X{});
```

LLAMA – Low Level Abstraction of Memory Access

LLAMA is free software (LGPL 3.0). Find us on GitHub!

LLAMA is a research prototype (but still useful)!

The library: https://www.github.com/alpaka-group/llama

Read the Docs: https://llama-doc.rtfd.io

Doxygen: https://alpaka-group.github.io/llama/

Vikunja – Primitives for alpaka

Out now: Vikunja 0.1.0

- Vikunja provides high-level C++ primitives
  - `reduce`
  - `transform`
  - More on the way!

- Based on alpaka

- Get it here: **https://github.com/ComputationalRadiationPhysics/vikunja** (MPL 2.0)

RedGrapes – Task Scheduling Library

Resource-based, Declarative task-Graphs for Parallel, Event-driven Scheduling

- Lightweight, application-level C++14 framework for creating and scheduling task graphs

- RedGrapes generates a task graph based on high-level resource descriptions and order of the code

- Get it here: **https://github.com/ComputationalRadiationPhysics/redGrapes** (MPL 2.0)

- Read the Docs: **https://redgrapes.rtfd.io/**

Planning for the Future

Next alpaka Release

alpaka 0.7.0 – Expected later this year

- Improvement of kernel language
  - SIMD types
  - Abstraction of CUDA intrinsics

- Finalize SYCL back-end
  → mainline support for oneAPI hardware and Xilinx FPGAs

- More flexible memory abstraction

bactria – Profiling & Tracing Library

bactria – Broadly Applicable C++ Tracing and Instrumentation API

- Ongoing research project (still in early stages)

- Idea: Instrument portions of your code with a modern C++ API

- Choose desired profiling / tracing library at runtime and analyse output with your favourite tools!

- Monitor progress here: **https://github.com/alpaka-group/bactria** (EUPL 1.2)

Long Term Considerations

Will alpaka support future technologies?

- alpaka's internal structure makes new back-ends easy to integrate

- Recent examples: new back-ends for HIP, OpenACC, SYCL (soon)

Will alpaka's ecosystem still be supported in 5, 10, 15, ... years?

Always in motion is the future. - Yoda

- Development funded by HZDR

- Developers active since ~2008

- Continuously expanding HPC software portfolio

- We are developing for our own science projects, too!