PIConGPU: Designing An Application

On the shoulders of alpaka, openPMD and beyond...

CASUS · HZDR · Julian Lenz · j.lenz@hzdr.de · www.casus.science

INSTITUTE OF

HZDR
HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF

PARTICIPATING INSTITUTIONS

UFZ HELMHOLTZ Zentrum für Umweltforschung

CBG Max Planck Institute of Molecular Cell Biology and Genetics

TECHNISCHE UNIVERSITÄT DRESDEN

Uniwersytet Wrocławski

FUNDED BY

Bundesministerium für Bildung und Forschung

SACHSEN

Diese Maßnahme wird mitfinanziert mit Steuermitteln auf Grundlage des vom Sächsischen Landtag beschlossenen Haushaltes.

Highlights of PIConGPU's adaption of alpaka, openPMD

and more...

Execution and lockstep programming

- Tailored kernel execution interface
- Abstraction for iterations

Memory Management

- Static: HostDeviceBuffer
- Dynamic: MallocMC

I/O and In-Situ Processing

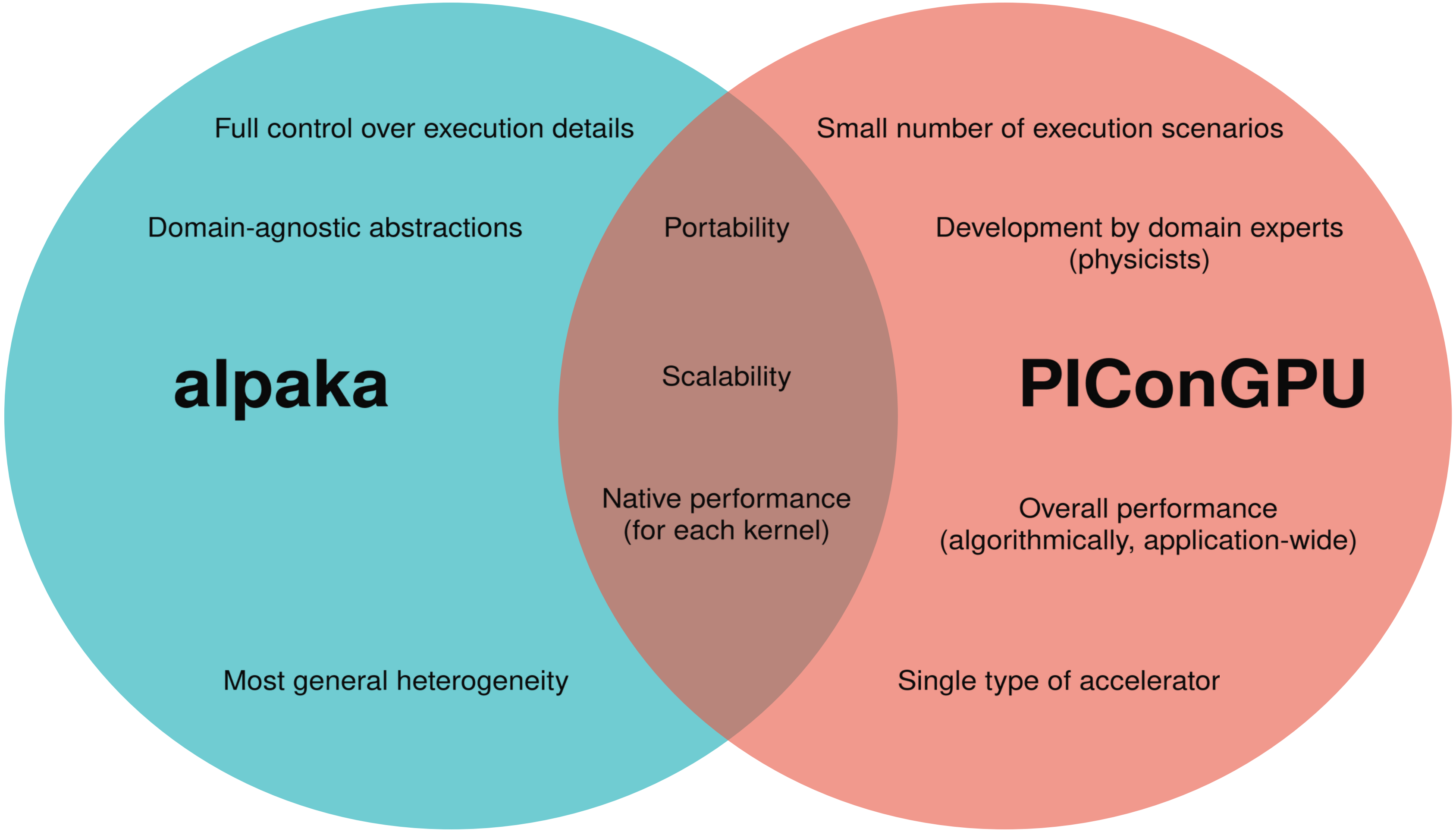
- OpenPMD as a backend
- Plugin system

Other Learnings

- Floating point precision and normalisation
- Performance tuning
- Professional software development
- …

alpaka is not (yet!) tailored to your needs

Design goals, requirements and constraints



Execution & Lockstep

Tailored kernel definition and execution

The Idea: An Interface Fitting Your Task

Lockstep Programming

The Physicist's Task

Conceptually most of our code is this:

```
void processAllCells() {
    for (auto& cell : cells)
    {
        process(cell);
    }
}
```

Physicist: Write the `process(cell)` function.

The Software Engineer's Tasks

- Keeping track of parallelisation parameters
- Handling indices
- Bounds checks
- Distributing work amongst threads
- Handling special cases
- ...

Provide a suitable interface!

- The details of your application of alpaka are **not generic** to all applications.
- But within your application you should provide **tailored abstractions and interfaces**!

The ForEach API: Abstracting iteration

Lockstep Programming

PIConGPU's ForEach User Code

```
auto processAllCells = [] ALPAKA_FN_ACC (auto const& worker) {

    auto forEachCell = makeForEach<numberOfCells>(worker);

    forEachCell(
        [](uint32_t const idx) {
            process(cells[idx]);
        }
    );
};
```

What does ForEach do? (see include/pmacc/lockstep/ForEach.hpp)

- Keep track of parallelisation parameters (152-154)
- Iteration with bounds checks, grid striding, etc. (192-232)
- Handle special cases like single-threading (235-243)

Tailored Execution: Exposing What You Need

Kernel execution

The PMACC_LOCKSTEP_KERNEL Macro

```
constexpr uint32_t blockSize = 32U;
uint32_t const numBlocks = 16U;

PMACC_LOCKSTEP_KERNEL(processAllCells)
    .config<blockSize>(numBlocks)
    (/*... we could pass arguments here... */);
```

The Call Stack (see include/pmacc/lockstep/ForEach.hpp and include/pmacc/exec/KernelLauncher.hpp)

- `PMACC_LOCKSTEP_KERNEL` (ForEach.hpp#295 → 281) → `KernelPreparationWrapper` (70)
  - debugging information, user kernel, launcher factory
- `.config<...>(...)` (100) → `KernelLauncher` (KernelLauncher.hpp#49)
  - execution details (in PIConGPU data structures), application context (event system, etc.)
- `.operator()` (87)
  - translate details to alpaka data structures, call alpaka::enqueue(...), inform event system

The Worker: Accelerator++

Execution & Lockstep

```
template<typename T_Acc, typename T_BlockCfg>
class Worker {
    T_Acc const& m_acc;
    uint32_t const m_workerIdx;

    static constexpr uint32_t numWorkers(){
        return T_BlockCfg::numWorkers();
    }
    static constexpr uint32_t blockDomSize() {/*...*/}
    static constexpr auto blockDomSizeND() {/*...*/}
    // ...

    void sync() const {
        alpaka::syncBlockThreads(m_acc);
    }
    // ...
};
```

A (Trivial) Real-World Example: Generating an ID

Execution & Lockstep

```
class IdProvider { // include/pmacc/IdProvider.hpp
    // ...
    uint64_t getNewIdHost() {
        HostDeviceBuffer<uint64_t, 1> newIdBuf(DataSpace<1>(1));

        auto kernel = [] ALPAKA_FN_ACC
                (auto const& acc, auto idGenerator, uint64_t* nextId)
            -> void {
                *nextId = idGenerator.fetchInc(acc);
            };

        PMACC_LOCKSTEP_KERNEL(kernel)
            .config<1>(1)
            (getDeviceGenerator(), newIdBuf.getDeviceBuffer().data());
        newIdBuf.deviceToHost();
        return *newIdBuf.getHostBuffer().data();
    }
    // ...
};
```

A (Non-Trivial) Real-World Example: Transition Rates

Lockstep Programming

What does it do? **(see /include/picongpu/particles/atomicPhysics/kernel/FillLocalRateCache_BoundFree.kernel#L193)**

- Request and initialise shared memory (214-228)
  - Only one single thread in use
  - `PMACC_SMEM` forwards to `alpaka::declareSharedVar`
  - `makeMaster` forwards to `makeForEach<1, 1>`

- Min/Max reduction over cells (230-258)
  - One thread handles one cell
  - Use atomics on shared memory

- Reduction of ~N^2 transition rates to one sum per state (260-303)
  - One thread handles transitions from one state
  - Thread-local reduction doesn't need atomics

Summary: Trade-Offs

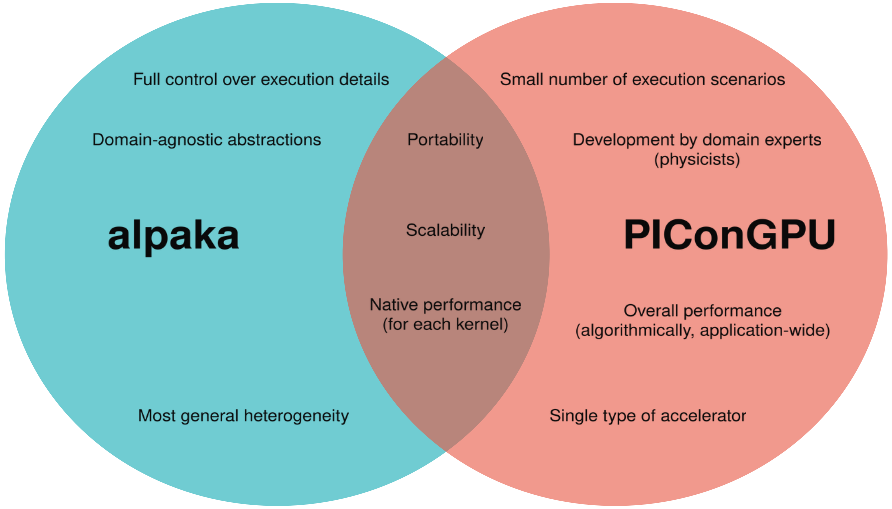
Lockstep Programming

Advantages

- Clean interface & reduction of boilerplate
- Versatile iteration schemes
- Compile-time known memory access
- Library agnostic

Restrictions of generality

- Block sizes must be compile-time const
- Execution details are inaccessible to user (device, queue, accelerator)



Outlook: That’s just a tiny glimpse

Lockstep Programming

Features we didn’t cover

- Another layer of abstraction to **iterate through our most common data structures**
- ContextVariables allow **carrying state between ForEachs**
- **Self-aware functors can express constraints** on their execution parameters
- **Might be coming to alpaka!**

Lessons learnt

Adoption of alpaka is best performed via…

an **application-specific interface layer**...

concretising (and hiding) application-specific choices.

Memory Management

Domain-specific memory handling

Integration into application context: Static == Fields

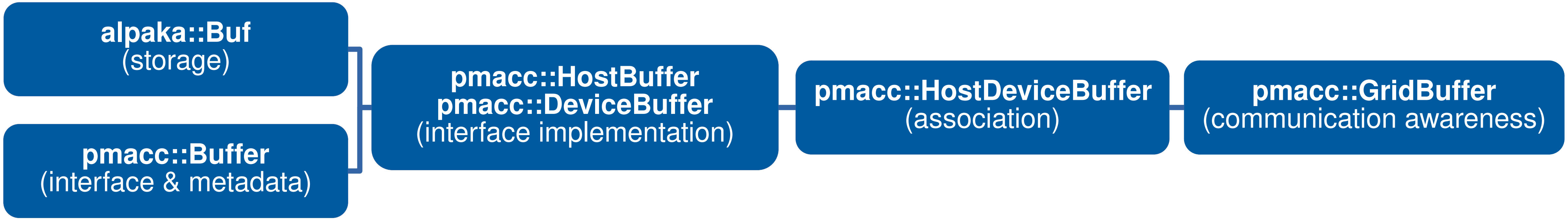
Memory Management

alpaka::Buf

- Basics: malloc/free
- Metadata: type, extent and pitch (host only)
- Explicit interface and fully flexible:
  - Device
  - Queues

PMacc's requirements

- Convenient interface (STL-like, ...)
- Metadata: on device!
- Concrete environment:
  - Fixed device
  - Integration into event system
- Association of host and device buffers
- Communication awareness (MPI)



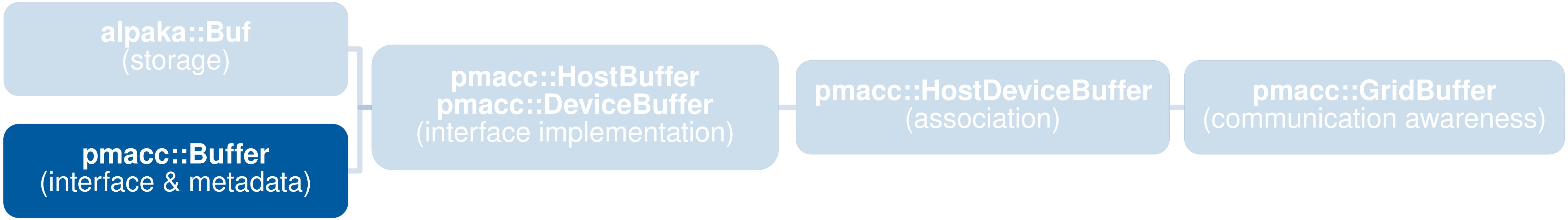
See /include/pmacc/memory/buffers/<class name>.hpp

Single Buffers

Memory Management (static)

pmacc::Buffer

- Interface & default implementations
- Stores own size in alpaka::Buf
- Models memory owning



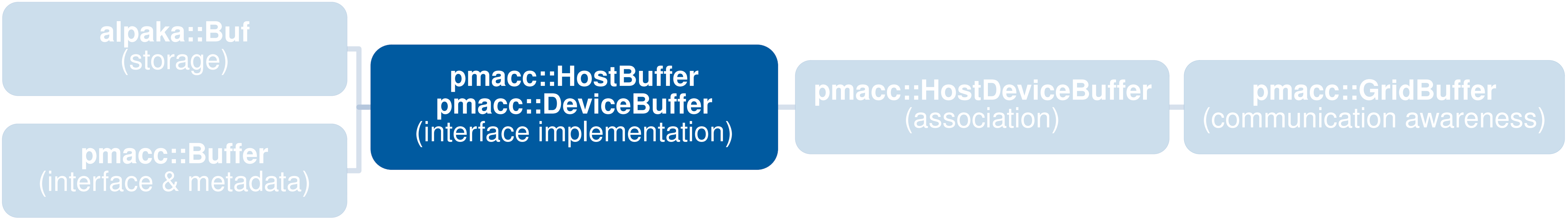
See /include/pmacc/memory/buffers/<class name>.hpp

Single Buffers

Memory Management (static)

pmacc::HostBuffer & pmacc::DeviceBuffer

- Keep memory in alpaka::Buf
- Stores size on device
- Operations interact with event system
- Provides copy operations



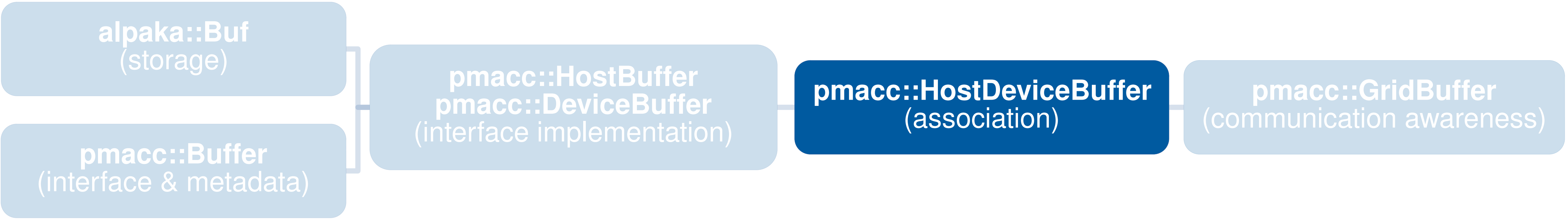
See /include/pmacc/memory/buffers/<class name>.hpp

Double Buffers

Memory Management (static)

pmacc::HostDeviceBuffer

- Associates one host and one device buffer
- Adds sync operations for convenience



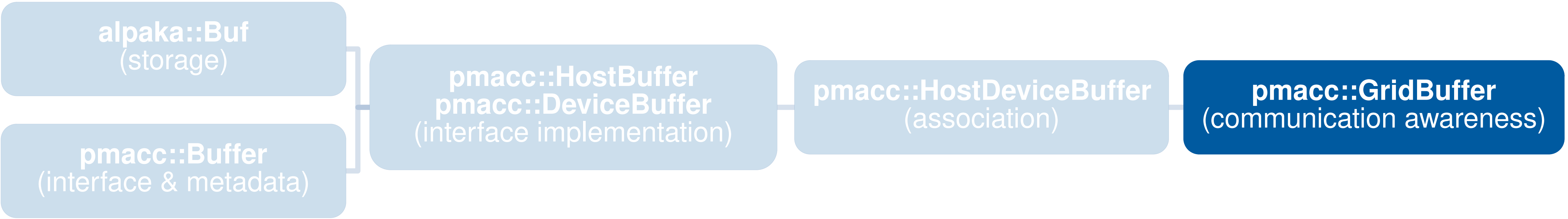
See /include/pmacc/memory/buffers/<class name>.hpp

Double Buffers

Memory Management (static)

pmacc::GridBuffer

- MPI communication interface
  - send
  - receive
  - sync
- Awareness of memory distribution



See /include/pmacc/memory/buffers/<class name>.hpp

Integration into application context: Static == Fields

Memory Management

alpaka::Buf	PMacc's requirements

• Basics: malloc/free
• Metadata: type, extent and pitch (host only)
• Explicit interface and fully flexible:
  ◦ Device
  ◦ Queues
	• Convenient interface (STL-like, ...)
	• Metadata: on device!
	• Concrete environment:
	  ◦ Fixed device
	  ◦ Integration into event system
	• Association of host and device buffers
	• Communication awareness (MPI)

alpaka::Buf
(storage)

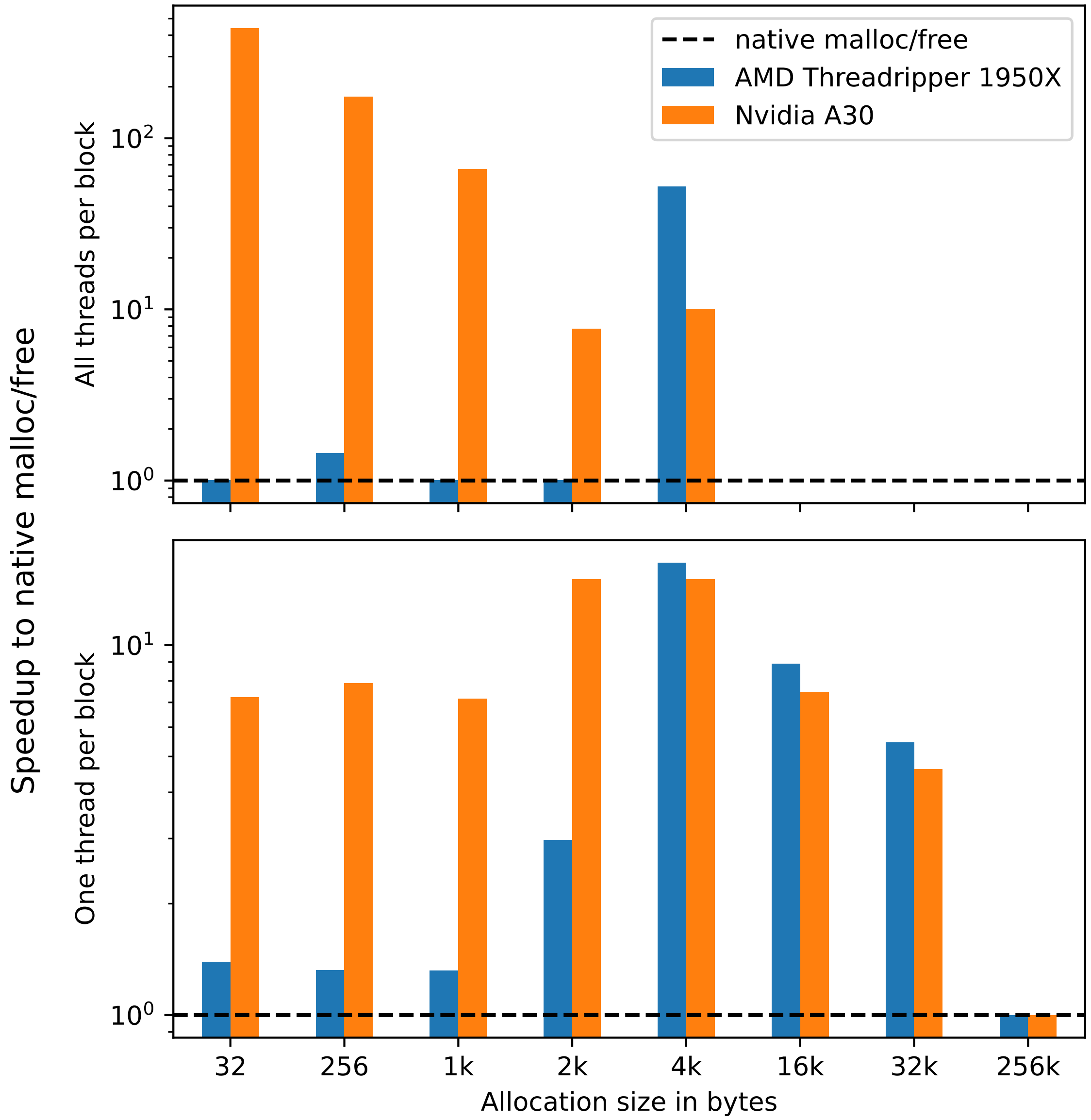
pmacc::Buffer
(interface & metadata)

pmacc::HostBuffer
pmacc::DeviceBuffer
(interface implementation)

pmacc::HostDeviceBuffer
(association)

pmacc::GridBuffer
(communication awareness)

See /include/pmacc/memory/buffers/<class name>.hpp



mallocMC: Dynamic == Particles

Memory Management

Particles move around,…

... so we need **dynamic, in-kernel allocations**!

Memory usage pattern:

- Allocate static memory on host and device.
- Reserve remaining device memory.
- Dynamically allocate particles...
- … on host (standard malloc/free) and
- … on device within reservation (mallocMC).
- Quantise allocations in „particle frames“.

I/O And Processing

Build only what you’re an expert in

I/O should be a thin layer in your application

I/O And Processing

You are...

… most likely not an expert in (all) distributed file systems and file formats.

… probably not paid for becoming one.

… hopefully an expert in how your numbers in memory shall related to numbers on disk.

Conclusion

You should employ a capable backend and only code the domain specifics on top.

Success Stories

I/O And Processing

Migrating to Frontier

- World's largest supercomputer
- Problem: I/O ran out of CPU memory
- Solution: New ADIOS2 engine
- Diff: +0 -0

Enabling streaming

- Avoid filesystem limitations
- Solution: openPMD streaming API
- Diff: +72 -19

Writing Metadata

- Metadata is backend independent.
- Problem: Different APIs for file formats
- Solution: openPMD's tailored API
- Diff: +162 -395

Open Source: Advance the community

- Allow aspects to mature in implementations.
- Contribute back:
  - ED-PIC extension
  - Dataset-specific compression

Further Learnings

Everything else we could possibly think off...

Precision and Normalisation

Know your limits and go beyond them!

Floating-point precision matters!

- Smaller type = larger memory throughput
- Specialisation of hardware (not always!)
- Rounding errors accumulate
- Domain's values might span large ranges

Example: Laser-Plasma Physics

- Speed ~ speed of light ~ 10^{8} m/s
- Time ~ plasma freq ~ PIC time step ~ 10^{-16} s
- Mass ~ macro-particle electron ~ 10^{25-31} kg
- Length ~ Time * Speed ~ 10^{-8} m

Normalisation and the unit system (see /include/picongpu/unitless/simulation.unitless)

- In **appropriate units** all occuring numbers will be close to one. → **Low precision required.**
- **Configurable** precision allows easier adaption to new hardware.
- Implementing a **unit system** allows convenient **interfacing with domain experts**.
- **Not all operations allow reduced precision!** (Example: sqrt)

Performance Tuning and Optimisation

Some Tips and Tricks

Understand your code!

- Everything happens for a reason!
- Compilers and tools are always right (except for when they are wrong)!
- Benchmark, trace and profile (potentially roofline models for bottlenecks)
- Feedback from production (ask users, check logs, check cluster statistics, ...)

alpaka: Tips and Tricks

- alpaka compiles to native code. → **Use vendor tools!**
- Single-header alpaka for **compiler explorer**
- Check **register counts, spills and occupancies:** `-Dalpaka_CUDA_SHOW_REGISTERS`
- Use `-Dalpaka_DEBUG=2` to check workDivs

HZDR CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Professional Software Development

The Basics

Automated testing and continuous integration

→ Vary architectures, compilers and libraries, too!

Manage contributions

→ Pull requests, issues, code reviews, …

Communicate about your code

→ Ask questions, pair programming, code reviews, …

Less is more!

→ If your team is small, keep your code small! (If your team is large, still keep your code small!)

Thanks for the attention!