Bernhard Manfred Gruber

Introduction

- Alpaka = Abstraction Library for Parallel Kernel Acceleration
- From the docs:

  The alpaka library is a **header-only C++14** abstraction library for **accelerator development**. Its aim is to provide **performance portability** across accelerators through the abstraction (not hiding!) of the underlying levels of parallelism.

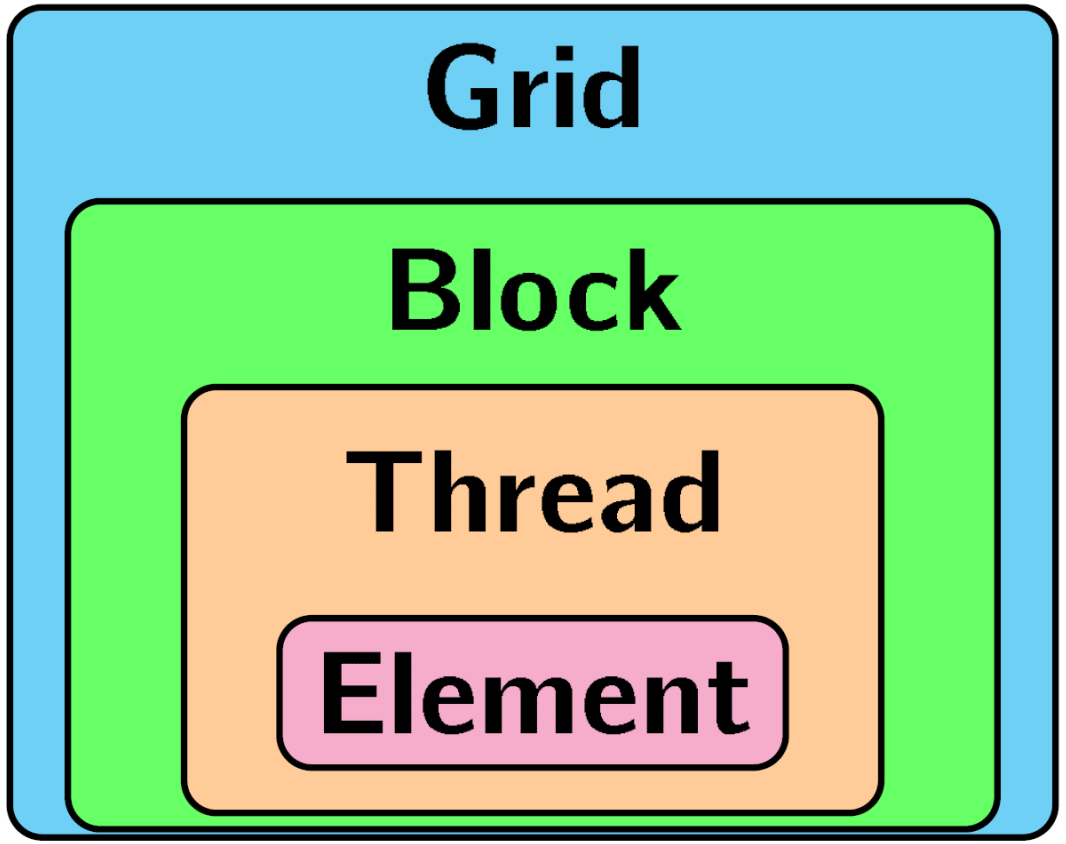
- https://github.com/alpaka-group/alpaka
- Mozilla Public License 2.0

Abstracts parallel technologies and platforms

- CUDA
- ROCm / HIP
- OpenMP
- OpenAcc
- SYCL
- TBB
- std::thread

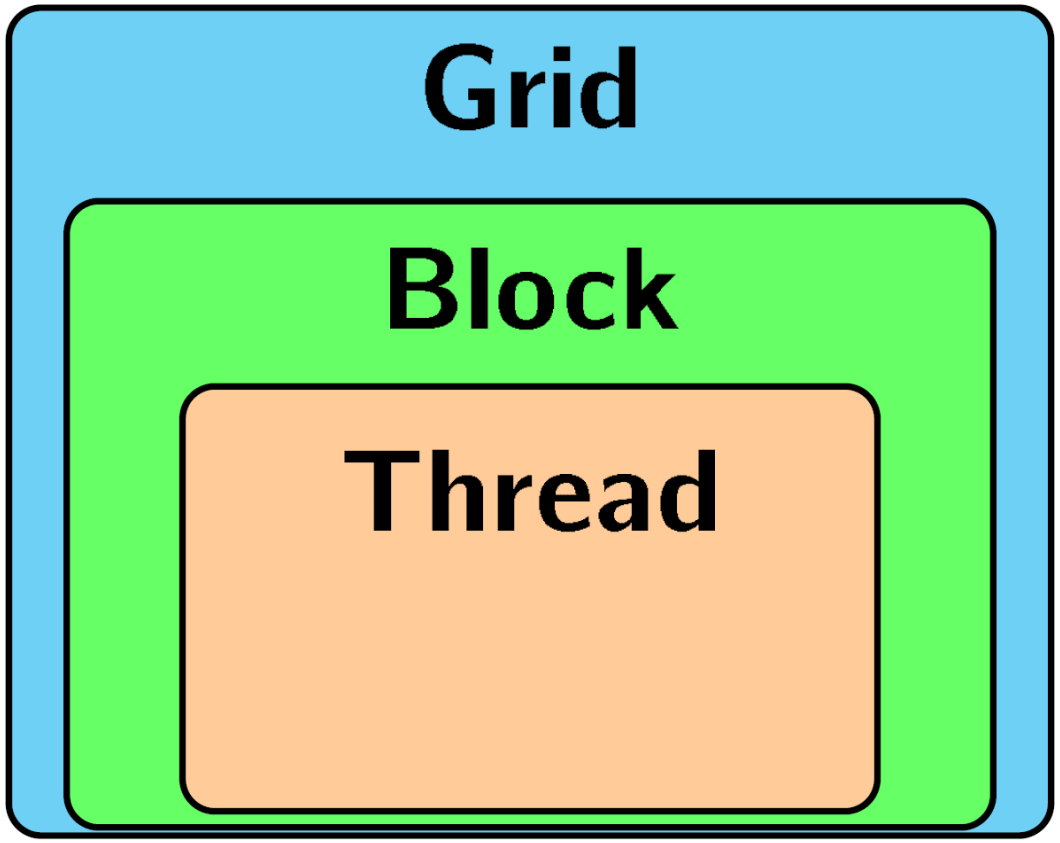
- Intel CPUs
- IBM OpenPower
- NVIDIA GPUs
- AMD CPUs + GPUs
- ARM
- FPGAs (via SYCL, future work)

Parallelism hierarchy



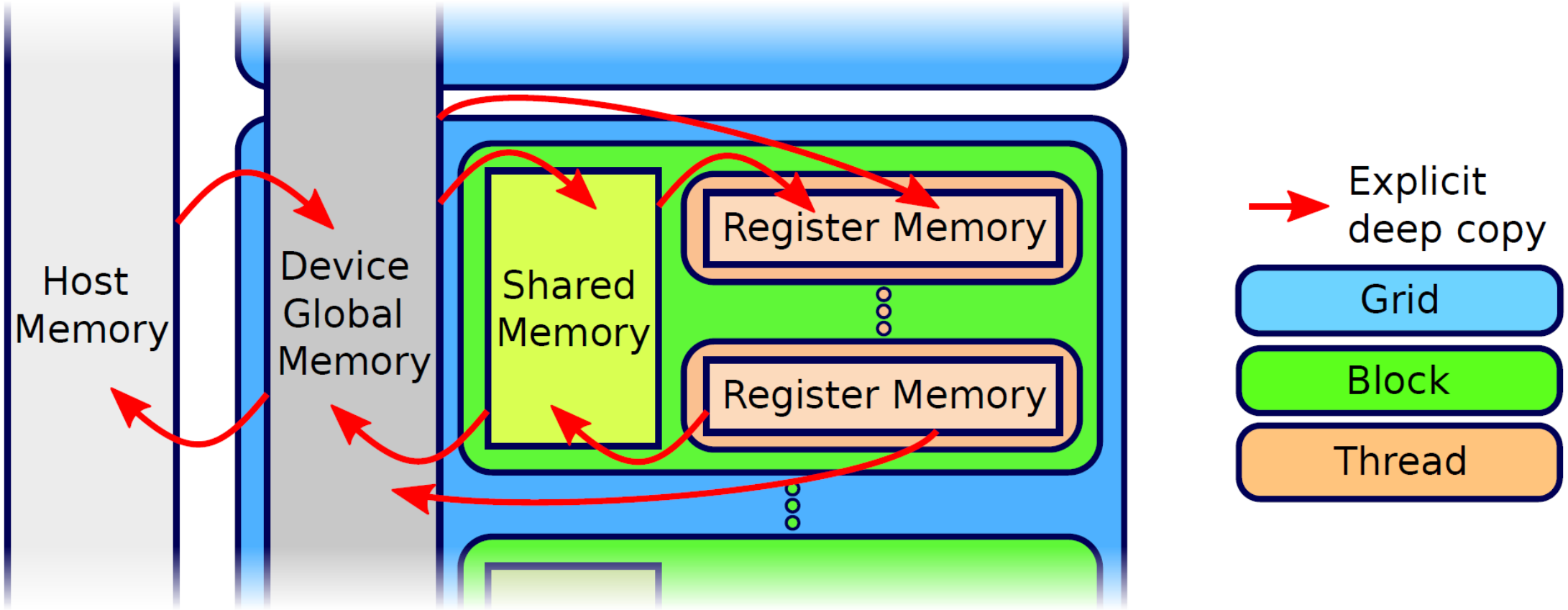
- Grid: whole parallel kernel task
- Blocks: execute independently within grid
- Threads: execute within blocks, can synchronize and communicate within block (cf. CUDA warps and SIMT)
- Elements: vector lanes

Memory hierarchy



- Grid: global memory
  - Usually buffer objects
- Block: shared memory
  - Special allocation function
- Thread: register memory
  - Local variables

Explicit deep copies between layers



Example, vector addition

1. Create 2 float buffers on host (a_{host}, b_{host}), fill with random values
2. Create 3 buffers on accelerator (a_{acc}, b_{acc}, c_{acc})
3. Copy 2 inputs from host (a_{host}, b_{host}) to accelerator (a_{acc}, b_{acc})
4. Run kernel, computes $c_{acc}[] = a_{acc}[] + b_{acc}[]$
5. Create result buffer on host (c_{host})
6. Copy results from accelerator (c_{acc}) to host (c_{host})
7. Read result in c_{host}

Vector addition – alpaka

```
using Idx = std::size_t;
using Dim = alpaka::dim::DimInt<1u>;
using Acc = alpaka::acc::AccGpuCudaRt<Dim, Idx>;
// using Acc = alpaka::acc::AccCpuOmp2Threads<Dim, Idx>;
using Queue
    = alpaka::queue::Queue<Acc, alpaka::queue::Blocking>;

const auto devAcc = alpaka::pltf::getDevByIdx<Acc>(0);
Queue queue{devAcc};
```

Vector addition – alpaka

```
const Idx size = 1024 * 1024;
const alpaka::vec::Vec<Dim, Idx> sizeVec{size};

using DevHost = alpaka::dev::DevCpu;
const auto devHost = alpaka::pltf::getDevByIdx<DevHost>(0);

auto bufHostA = alpaka::mem::buf::alloc<float, Idx>(devHost, sizeVec);
auto bufHostB = alpaka::mem::buf::alloc<float, Idx>(devHost, sizeVec);
```

Vector addition – alpaka

```
float* a = alpaka::mem::view::getPtrNative(bufHostA);
float* b = alpaka::mem::view::getPtrNative(bufHostB);


std::default_random_engine eng;
std::uniform_real_distribution<float> dist{1.0f, 42.0f};
for (Idx i = 0; i < size; i++) {
    a[i] = dist(eng);
    b[i] = dist(eng);
}
```

Vector addition – alpaka

```
auto bufAccA = alpaka::mem::buf::alloc<float, Idx>(devAcc, sizeVec);
auto bufAccB = alpaka::mem::buf::alloc<float, Idx>(devAcc, sizeVec);
auto bufAccC = alpaka::mem::buf::alloc<float, Idx>(devAcc, sizeVec);

alpaka::mem::view::copy(queue, bufAccA, bufHostA, sizeVec);
alpaka::mem::view::copy(queue, bufAccB, bufHostB, sizeVec);
```

Vector addition – alpaka

```
const Idx elementsPerThread = 4;
const auto workDiv
    = alpaka::workdiv::getValidWorkDiv<Acc>(
        devAcc, sizeVec, elementsPerThread,
        false, alpaka::workdiv::
            GridBlockExtentSubDivRestrictions::
            Unrestricted
    );
```

Vector addition – alpaka

```
VectorAddKernel kernel;
const auto task = alpaka::kernel::createTaskKernel<Acc>(
    workDiv, kernel,
    alpaka::mem::view::getPtrNative(bufAccA),
    alpaka::mem::view::getPtrNative(bufAccB),
    alpaka::mem::view::getPtrNative(bufAccC),
    size
);
alpaka::queue::enqueue(queue, task);
```

Vector addition – alpaka

```
using Idx = std::size_t;
struct VectorAddKernel {
    template<typename Acc>
    ALPAKA_FN_ACC
    void operator()(const Acc& acc, const float* a, const float* b, float* c, Idx size) const {
        const Idx threadId = alpaka::idx::getIdx<alpaka::Grid, alpaka::Threads>(acc)[0];
        const Idx elementsPerThread
            = alpaka::workdiv::getWorkDiv<alpaka::Thread, alpaka::Elems>(acc)[0];
        const Idx beginElemId = threadId * elementsPerThread;

        if(beginElemId < size) {
            const Idx endElemId = alpaka::math::min(acc, beginElemId + elementsPerThread, size);
            for(Idx i = beginElemId; i < endElemId; i++)
                c[i] = a[i] + b[i];
        }
    }
};
```

Vector addition – alpaka

```
auto bufHostC = alpaka::mem::buf::alloc<float, Idx>(devHost,
sizeVec);
alpaka::mem::view::copy(queue, bufHostC, bufAccC, sizeVec);
alpaka::wait::wait(queue);

const float* c = alpaka::mem::view::getPtrNative(bufHostC);
// ...
```

Vector addition – CUDA

```
const auto size = 1024 * 1024;
std::vector<float> a(size);
std::vector<float> b(size);

std::default_random_engine eng;
std::uniform_real_distribution<float> dist{1.0f, 42.0f};
for (auto i = 0; i < size; i++) {
    a[i] = dist(eng);
    b[i] = dist(eng);
}
```

Vector addition – CUDA

```
float* bufAccA;
float* bufAccB;
float* bufAccC;
cudaMalloc(&bufAccA, sizeof(float) * size);
cudaMalloc(&bufAccB, sizeof(float) * size);
cudaMalloc(&bufAccC, sizeof(float) * size);

cudaMemcpy(bufAccA, a.data(), sizeof(float) * size,
    cudaMemcpyHostToDevice);
cudaMemcpy(bufAccB, b.data(), sizeof(float) * size,
    cudaMemcpyHostToDevice);
```

Vector addition – CUDA

```
constexpr auto blockSize = 256;
constexpr auto blocks = (size + blockSize - 1) / blockSize;
vectorAdd<<<blocks, blockSize>>>(
    bufAccA, bufAccB, bufAccC, size);

__global__ void vectorAdd(const float* a, const float* b,
    float* c, size_t size) {
    const auto i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < size)
        c[i] = a[i] + b[i];
}
```

Vector addition – CUDA

```
std::vector<float> c(size);
cudaMemcpy(c.data(), bufAccC, sizeof(float) * size,
    cudaMemcpyDeviceToHost);

cudaFree(bufAccA);
cudaFree(bufAccB);
cudaFree(bufAccC);
```

Vector addition – C++17 (& nvc++)

```
// same setup of a and b as for CUDA

std::vector<float> c(size);
std::transform(std::execution::par_unseq,
    begin(a), end(a), begin(b), begin(c),
    std::plus<float>{}
);
```

Miscellaneous

- Pinning memory
  - alpaka::mem::buf::pin(buffer);
- Synchronization within block
  - alpaka::block::sync::syncBlockThreads(acc);
- Shared memory
  - auto& s = alpaka::block::shared::st::allocVar<int[32], __COUNTER__>(acc);
- Cupla library
  - Alpaka wrapper for easier porting from CUDA
  - https://github.com/alpaka-group/cupla