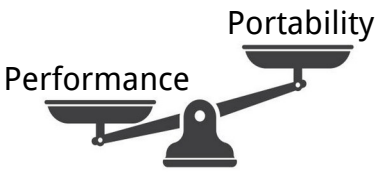
Offloading tools

ACTS Parallelization Meeting
04.03.2022

Motivation

- Software application's
  - Performance → How well a task was done (e.g. wall clock time)
  - Efficiency → How well the resources are used (e.g. portability, GPU usage)
- Nevertheless, both depend on the software's ability to expose parallelism (e.g. latest hardware & sequential code ensures neither performance nor efficiency)



- Trade-off solutions: Kokkos, SYCL, HIP, oneAPI, OpenMP, ...
- This talk focuses on yet another alternative, but motivated by efficiency rather than performance!

Proposed requirements

- Easy to use in plain C++ & limited code changes to adopt the API
- Abstract the notion of architecture (as much as possible)
- Support single-source code & generated code for different targets/optimizations
- Compilable by main stream C++ compilers (e.g. clang)
- Portable → target shared-memory CPU and NVIDIA/AMD GPUs (and potentially Big Data platforms: Google Cloud/AWS)
- Good (but most likely not peak) performance

Do you see the benefits of such an API?

Is there anything missing?

Clang-offload framework

Two-piece puzzle

- API → "marks" the parallel regions and offloads the computations to different architectures (same code base for all back-ends)
  - Header-only library which can be used independently from the tool
  - Relies heavily on vecmem data structures & memory models
  - Preconditions: no STL, thread-safe code
- Clang-tool → code duplication, validations and source-to-source translation (different code bases for different back-ends)
  - Implemented on top of clang-tooling mechanism
  - Generates extra code if needed (e.g. add function attributes)
  - Ensures further code optimizations for different architectures
  - Preconditions: projects built with cmake

API

Concerns when automatically **offload** & **distribute** computations

1) Memory allocations & transfers

2) Transfer function pointers and/or functor object pointers to the device

3) Distribute the computations based on thread index

Input from the user is needed → abstractions!

API – Offload

Data offloading

- Use vecmem vectors as iterable datasets
- If the input data is allocated in
  - managed memory → intermediate and final results are allocated on the device/managed memory
  - host memory → copy to device, allocate results on both host/device, copy results back to host

Function offloading

- Work around the polymorphism's restriction on the GPU by lambda captures
  - In some scenarios, constructing the object on the device can still be needed

API – Distribute computations

Abstractions from functional programming

- (map f coll) → apply f to each element of the collection; the output is another collection of the same size with elements of the same/other type
  - `(map inc `(1 2 3)) → (2 3 4)` // same size, same/different type
  - `(map toSpacePoints measurements) → (sp1 sp2.. spN)`
- (filter p coll) → keep in the output collection only the elements from the input collection which satisfy the predicate p
  - `(filter even? `(1 2 3)) → (2)` // same/smaller size, same type
  - `(filter isAboveThreshold? `(sp1 sp2 sp3)) → (sp1 sp2)`
- (reduce f coll init) → reduce the elements of the collection using function f and store it in the result initialized with init
  - `(reduce + `(1 2 3) 0) → (6)`

Do these abstraction cover all scenarios?

API – Distribute computations

```
template<typename Ri, typename... Args>
struct parallelizable_map_reduce_algorithm {

    virtual Ri& map(Ri& result_i, Args... args) = 0;

    virtual Ri* reduce(Ri* result, Ri& partial_result) = 0;

}
```

```
template<typename R, typename Ri, typename... Args>
struct parallelizable_map_filter_algorithm {

    virtual Ri& map(Ri& result_i, Args... args) = 0

    virtual bool filter(Ri& partial_result) = 0;

}
```

API – Code snippet – Seq vs par algorithm

```
// init memory resource (could as well be managed_memory here)
vecmem::host_memory_resource mr;

// define and init vector vec
...


// instantiate the algorithm
my_algorithm alg(mr);


// call the algorithm in sequential mode
vecmem::vector<double> result = alg(vec);


// call the algorithm in parallel mode
vecmem::vector<double> result = api::parallel_algorithm(alg, mr, vec);
```

API – Code snippet – Seq vs par algorithm

```
// init memory resource (could as well be managed_memory here)
vecmem::host_memory_resource mr;

// define and init vector vec and all the other input params
...
Another_object x();

// instantiate the algorithm
my_algorithm alg(mr);


// call the algorithm in sequential mode
vecmem::vector<double> result = alg(vec, x);


// call the algorithm in parallel mode
vecmem::vector<double> result = api::parallel_algorithm(alg, mr, vec, x);
```

API – Code snippet – More algorithms

```
// init memory resource (could as well be managed_memory here)
vecmem::host_memory_resource mr;

// define and init vector vec
...



// instantiate the algorithms
my_algorithm1 alg1(mr);
my_algorithm2 alg2(mr);

// call algorithms in parallel mode
vecmem::vector<double> result1 = api::parallel_algorithm(alg1, mr, vec);
double result2 = api::parallel_algorithm(alg2, mr, result1);

// OR EQUIVALENT
api::parallel_algorithm(alg2,mr, api::parallel_algorithm(alg1, mr, vec));
```

API – Code snippet – Simple functions

- Ad-hoc offload offload & parallelization using managed memory

```
// if vec is allocated in managed memory
api::parallel_map(vec.size(), [=] __device__ (int idx,
                        vecmem::data::vector_view<int>& vec_view) mutable {
                             vecmem::device_vector<int> d_vec(vec_view);
                             // add code here
                        }, vecmem::get_data(vec));
```

API – Code snippet – Simple functions

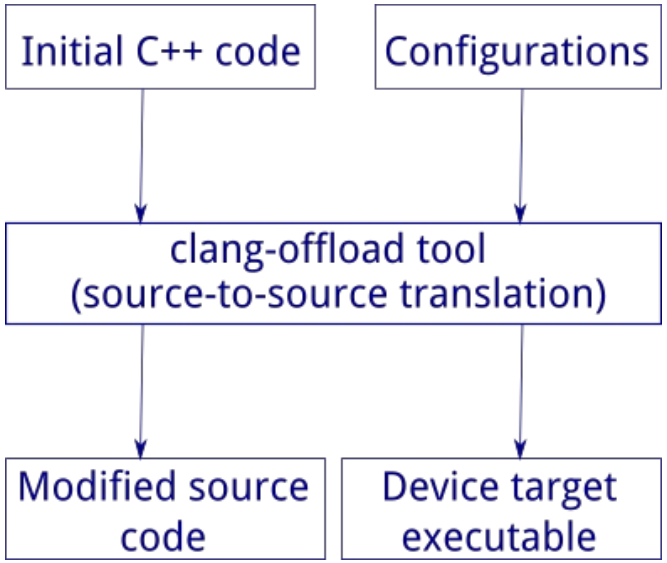
- Ad-hoc offload offload & parallelization using host memory

```
// if vec is allocated in host memory
vecmem::cuda::device_memory_resource d_mem;
vecmem::cuda::copy copy;

// copy host to device
auto vec_buffer = copy.to(vecmem::get_data(vec), d_mem,
vecmem::copy::type::host_to_device);

api::parallel_map(vec.size(), [=] __device__ (int idx,
                      vecmem::data::vector_view<int>& vec_view) mutable {
                        vecmem::device_vector<int> d_vec(vec_view);
                        // add code here
                    }, vecmem::get_data(vec_buffer));

// copy device to host
copy(vec_buffer, vec, vecmem::copy::type::device_to_host);
```

Clang-tool

Steps

- Duplication
  - Copy the code into a new folder because the changes are destructive
- Validation
  - GPU backends: Identify STL calls in the C++ code
- Code modification
  - (if needed) Annotate functions from the AST rooted in the API call
  - Hardware-aware optimizations



- (Optional) Compile & link → executable for the given backend

Development status – API

- Done
  - Basic infrastructure to support the execution of a (parallelizable) algorithm and/or a lambda function using CPU OpenMP and CUDA
  - Kernel configuration based on problem size
  - Unit tests based on google test library
- On-going
  - Extend kernel parametrization to include memory considerations and hardware capabilities
  - More efficient reductions/filtering
- Proposed next
  - Algorithm composition API
  - Add support for AMD backend
  - Extend the offloading functions to support jagged-vectors (if needed)

Development status – clang-tool

- Done
  - Basic tooling infrastructure for backends: CPU OpenMP, GPU OpenMP and GPU CUDA
  - Polymorphic validators (STD checker), translators and builders
- On-going
  - Adapt the translators based on the latest changes in the API
  - Automated tests
- Proposed next
  - Performance optimizations for tailored for backends

Outlook

- When on-going dev is complete, make the API repository public (~1 week)
- Any feedback is highly appreciated!
  - Could these tools be somehow useful to ACTS?
  - What benefits would they need to provide in order to be adopted?
  - Name suggestions?
- Test on an algorithm from traccc. Any recommendation?

Thank you!