Alpaka Training Notebook

In order to streamline the discussion we would like to ask you to write down any questions you would have to the Alpaka team in this notebook. During the “Q&A, Wrapup” session after the break we will try to answer those.

Requests for extra topics on Friday

- I will add it myself, as I believe this may be of interest to some participants. In the first day slides we mentioned that we have another library cupla on top of alpaka. It offers a very CUDA-like interface with C-style API, no lengthy templates, etc., and alpaka is used internally. So all the alpaka portability is there, but under a different API. It can be more convenient to port existing CUDA applications. We do not have slides for cupla, but could discuss it and show something via screen sharing (Sergei)

Questions / Thursday

- (Also put to the Questions / Future section)

Questions / Wednesday

- (Questions were put to the following section and answered there)

Questions / Future?

- You said about memory management : “In case of CPU Devices there is optimization potential in avoiding unnecessary copies !”. This optimization must be done by the developer, or taken in charge by Alpaka, or planned to be taken in charge by Alpaka ? (David)
    - In the current state, by developer. However, in our experience this is relatively straightforward (of course, application-dependent). To give a simplistic solution, one could have an application-level buffer abstraction (and many applications have it already regardless of alpaka), which has the logic to either copy the whole contents of a buffer, or just copy the raw pointer and not the contents. One can easily query whether the device type is CPU or not to make such a decision. (Sergei)
    - I feel alpaka might want to provide some construction for that, to make it easier, somewhat out-of-the-box, but we currently do not have it. Feel free to request a feature via a github issue! (Sergei)

    - To avoid unnecessary memory copies we need to integrate the concept of unified memory into alpaka. This is on our todo list. The tricky part is the CUDA unified memory integration: You need to take care that you do not access memory from the host and the device at the same time. If a CPU backend is used this is not the problem because the cache protocol will handle it (maybe with a performance decrease) (psychocoderHPC)
- (For when atomics will be introduced) Alpaka's atomics API is quite different from that of most other CPU and GPU APIs, which steepens the learning curve. What is the design/implementation concern which led to this interface divergence? (Hadrien)
    - I agree with the assessment. There is actually an open issue to simplify it: https://github.com/alpaka-group/alpaka/issues/1005 . Please feel free to contribute to the discussion! (Sergei)
    - I believe part of the reason for a more complex interface is to first avoid deducing whether addresses are in global or shared memory on the alpaka side, and have a user specify it. And secondly to potentially allow fine control over atomicity level, e.g. to allow atomic operation on global memory address, but to be atomic only between threads of a block. But perhaps those who contributed to the original design of Alpaka could shed more light on that. (Sergei)
    - As Sergei wrote, the alpaka interface gives us more fine control over atomic operations. When you work with an accelerator `AccCpuOmp2Blocks` where the block size is required to be one we can avoid using atomic operations if the user knows that the operation should be atomic between threads in a block. (psychocoderHPC)
- (Device query question, I don't know if you plan to cover this topic further) The provided cheatsheet showcases device selection with a simple number, which is perfect on HPC systems with a homogeneous device configuration (all GPUs are the same, all CPU cores are the same). But consumer systems, such as developer machines, can potentially be a lot messier, with things like one integrated GPU and one discrete GPU, or maybe in the future heterogeneous CPU configurations like Arm's big.LITTLE if Apple's "Arm everywhere" plans pan out. APIs like OpenCL (which Alpaka can interface via SyCL) expose this hardware mess. Do you have, or plan to add, some device queries for figuring out what a device number maps into and picking the right devices? (Hadrien)
    - It is not just any device with the given index. It is a device suitable for a particular accelerator (OpenMP 4.0, CUDA, etc.) with the given index. So a discrete CUDA GPU will not get mixed with an integrated GPU, neither will CPUs with GPUs. However, when having 2 different models of CUDA GPUs, both will be shown as CUDA devices. For such cases, one could query device properties via alpaka API, namely traits in alpaka/dev/Traits.hpp. The set of available properties is very limited at the moment compared to e.g. cudaGetDeviceProperties(), but it can be extended rather easily when necessary.
    - My personal workflow with PIConGPU (alpaka-based plasma simulation code) is that I normally enable only one alpaka backend at a time (via cmake options)

when developing, also to reduce compilation time. Then I can just use device 0 and not worry about it on my development machine. Then it's the job of CI/tests to check on all configurations. (Sergei)

    - Unfortunately, I am not well-informed of the Apple stuff you mentioned, so can't really comment on that (Sergei)
        - Apple recently announced a plan to migrate Macs from Intel to in-house Arm chips. If they want to keep costs down, they'll stay as close as possible to their mobile SoCs, and that means heterogeneous designs with a mixture of high-power and low-power CPU cores. On mobile, the fast cores are intended for interactive work, the slow ones for processing background events (esp while a machine is sleeping). Unfortunately, I'm not familiar enough with Darwin system APIs to tell if applications need care to avoid spawning threads on both kinds of cores when unwanted.
        - Also, besides those Apple/Arm plans, Intel recently announced a "Lakefield" chip with a similarly heterogeneous core architecture: https://www.anandtech.com/show/15877/intel-hybrid-cpu-lakefield-all-you-need-to-know . It's unclear at this point in time if they plan to generalize this architecture across more future laptop CPUs.
- (Another device enumeration question) How do you handle / plan to handle backends with a notion of "platform" (multiple implementations of the same API for different hardware) like OpenCL/SyCL? (Hadrien)
    - This is a great question. We currently only have this situation with HIP, supporting both NVIDIA and AMD, but Alpaka currently only allows having one enabled at a time.
        - I am not sure if this will work with OpenCL-ish backends, where platform selection is a runtime decision. (FWIW, this is more generally true of APIs designed by Khronos.) (Hadrien)
    - We are currently working on this behavior during the implementation of OpenMP5 offloading and OpenAcc. My current idea to handle it is to have in CMake an option to select the target architecture. For CUDA you can already define which architecture you like to support e.g. -DALPAKA_CUDA_ARCH=60;75 (psychocoderHPC)
        - ...and then this would automatically generate some platform enumeration/query/selection code on application startup which e.g. selects the Intel SyCL platform if you want to target an Intel GPU? (Hadrien)
            - Yes this could activate the code to allow access to a device with a special abstraction model. It could be that this requires some interface extensions e.g. `getDevice(OpenMP5{}, model::Offloading{});`, `hasSupportFor(OpenMP5{},model::Offloading{});` (psychocoderHPC)

- What is the best way to query the properties of the target device in Alpaka as this may be needed to setup the optimal parameters for the grid, blocks and threads?
    - This is a good question and unfortunately not fully covered by Alpaka in its current state. The issue is that while CUDA/HIP have detailed property access as part of their API, while there is no standard (not third-party) API for other platforms. So alpaka offers some things in alpaka/dev/Traits.hpp, but there are generally not enough for a realistic application. How we actually do it for PIConGPU is to have a branch for this particular case (while all the computational part is actually alpakafied and single-source) and so use cudaGetDeviceProperties directly for the CUDA backend.
    - One important issue to solve this is to integrate `hwloc` https://github.com/alpaka-group/alpaka/issues/36 (psychocoderHPC)

Questions / Tuesday

- Including a simple struct with header only getters/setters is problematic in CUDA because the functions are not declared __global__ or __device__. Can Alpaka work around this problem?
    - We will talk about it in today's materials. Alpaka introduces a macro for function specifier, that should be used for all functions that are run on the device side (so both __global__ and __device__ functions in CUDA speech). So alpakafied functions all just use this specifier, and depending on the backend it will be "assigned" a correct value
    - Upon discussion in voice, so in case the included code does not have any specifiers and those can't be added, then it will be possible to use such code with alpaka CPU backends, but not GPU backends. Since alpaka is template-based, you could still write it all as normal with alpaka, just not instantiate with unsupported accelerator types. Here the root issue is that the specifier requirements come from CUDA/HIP, not alpaka, so nothing we can do about it.
- (Another queue question, again may want to postpone until queues are covered) In the cheatsheet, I see that `wait::wait(event)` waits for all commands before the requested event in the queue. At least OpenCL (with out of order queues), and maybe Vulkan too, allows finer grained synchronization, waiting for some commands to have completed but not others. Is this in scope for the Alpaka synchronization model? (Hadrien)
    - The alpaka way to achieve it is to use multiple queues for a device. Each single alpaka queue executes tasks in order. So a way to wait for a certain kernel or memory operation to complete is to enqueue an event just after it, and wait for this event. It is also possible to test if the event is finished. So altogether, with multiple queues, it allows proper flexibility and efficiency in our opinion.
        - I see. There are performance implications to this (as cross-queue sync is more expensive than intra-queue sync), but right now I cannot think of a scenario in which this would be a problem so…

            - Good point. Perhaps other alpaka developers have an idea there? We could also discuss it further here (maybe during Q&A after the lectures), or via a github issue. Please feel free to create one: we use issues as general discussion points, not just bugs!
- (Yet another queue/sync question, sorry) Vulkan has multiple synchronization primitives of varying run-time cost depending on what you want to synchronize (successive commands in a queue, non-successive commands in a queue, queue & host or multiple queues). How would you express something like this in the Alpaka abstraction vocabulary? (Hadrien)
    - Alpaka can synchronize queues, devices and events. As in CUDA you can enqueue an event behind each kernel or memcopy. It is not possible to sync non-successive tasks/events. The reason is that alpaka is not holding a list with not finished tasks, this would introduce runtime overhead. (psychocoderHPC)
    - Synchronizing `multiple queues` is currently not possible (with one API call) but a good suggestion for an interface extension. We will transform this request into an issue to discuss this with other developers. (psychocoderHPC)
        - Thanks, I'll probably create a couple tickets on my own after this training as well. In general, I kind of like the Vulkan synchronization model as a stress test for cross-API abstractions, because it's a pretty good highest common denominator of every possible synchronization point that a GPU API can ever expose ;) This stresses the abstraction design compromise between not exposing more than the lowest common denominator that some APIs can handle (which requires emulation on those APIs) and not hiding power from the most expressive target API.
            - Yes, this is very welcome! (Sergei)
- What do you do when your domain size cannot be split into an even number of blocks without using performance-killing 1x1 blocks? For example, if your domain size is 201x248 and your GPU likes 16x16 thread blocks for this problem. (Hadrien)
    - Option 0: (not universally applicable) align your problem size to block size by padding with more elements/tasks/… that do nothing and do not interfere with the original problem.
    - Yes, so there is a straightforward option 1, like used in CUDA tutorials and also our alpaka workshop. Just keep it simple, have one thread working on one portion of data, and so have the number of threads being equal to your problem size. Then you would normally hard-pick or query from your device the block size. And the number of blocks is the upper integer part of problem_size / block_size. To cope with problem_size being not a multiple of block_size, you just check inside a kernel if the thread index is within the problem size range, and do nothing if it is out of range.
    - More generic, option 2. Write the kernel in a way that works for any number of threads, blocks, and problem size. In this case the kernel normally has the following structure (that i called "strided loop" in voice):

```
    for (int idx = global_thread_index; idx < problem_size; idx +=
    global_number_of_threads)
    {
       // process element idx
    }
    ```
        - What I missed here is what global_number_of_threads should be. Is it akin to the elem_per_block kernel launch parameter?
            - It is just number_of_threads_per_block * number_of_blocks (which are both kernel launch parameters), you can see an example in the kernel of helloWorld_lesson16 in the examples repository
    - Most generic, for potentially better vectorization, option 3.
      Alpaka element level is used for loop blocking, basically how one may write code for CPUs. number_of_alpaka_elements is part of kernel launch configuration. Can be e.g. 1 for GPU backends and the loop blocking size for CPU ones.
      ```
      for (int idx = global_thread_index * number_of_alpaka_elements; idx <
      problem_size; idx += global_number_of_threads * number_of_alpaka_elements)
      {
         // #pragma omp simd or otherwise help vectorization when necessary
         for (int i = idx; i < idx + number_of_alpaka_elements; i++)
          {
              // code to process element i logically
              // (for vector execution will process multiple at once)
           }
      }
      ```

Questions / Monday

- What is the current status of Alpaka's interfaces to non-NVidia hardware? (Hadrien)
    - Full support for CUDA, CPUs through OMP (2.0, 4.0) and TBB. Full support for HIP for both AMD and NVIDIA. Experimental SYCL support (partly working on Intel GPUs, some features like atomics aren't working yet, some features are working differently wrt Alpaka => requires thought. Only works on Intel's SyCL compiler for now because other compilers have impl issues, not tested on FPGAs yet).
- What benefits does Alpaka currently offer, and plan to offer in the future, over direct use of the underlying API(s)? (Hadrien)
    - Main benefit is portability over NVidia, AMD, and CPUs.
    - Another benefit is C++-style API which is optimized away by the compiler. CUDA-style C APIs can encounter optimization issues at times.
- How would you compare Alpaka to other C++ heterogeneous programming framework projects such as Kokkos and RAJA? (Hadrien)

    - Comes from the same place, tries to do the same things, also single source.
    - Alpaka is a bit lower-level / more detailed. But not a big difference.
    - Kokkos also offers utilities like parallel algorithms, which Alpaka does not yet offer out of the box. It will ultimately offer them.
    - For now, Alpaka is a bit more developed in the “portability” area, and Kokkos is a bit more developed when it comes to high-level utilities.
- Does Alpaka support using the various memory layers which are available on a GPU? E.g. global, shared, constant, etc memory? (Stefan)
    - Constant memory is not currently supported.
    - Follow-up: GPUs use similar general concepts. But what if we want to use API functionality which is specific to e.g. CUDA? E.g. texture memory.
        - Alpaka provides abstractions, based on the CUDA/OpenCL API style. If new hardware appears that doesn’t match, the model will need to be extended.
        - If you use something like texture interpolation, it’s up to you to provide an emulation of it or alternative to it for CPUs, Alpaka will not do it for you because there’s no good general solution (emulation is slow, alternative is alg-specific).
- Talking to some GPU programmers, their view is that one can abstract on various GPU programming systems (CUDA, HIP, …) via C++ macros. What is your viewpoint on this? (Stefan)
    - Goal is to support all systems, not just GPUs but also CPUs, and that’s harder to do via macros because there start to be qualitative hardware differences.
    - Also, if you write your own macros, you must support the code yourself...
- Looking at the vectorAdd example, I see that kernel code is generic over the index type. Why is this needed? Please do not bother discussing this now if it will be explained in a later session ;) (Hadrien)
    - Allows adapting index type to the work domain: if the domain is small, you can use 32-bit ints. If the domain is large, you must use 64-bit ints. If you don’t care and want the machine type, you can use size_t.
    - Received some feedback that this may be an excessive customization point, examining this kind of feedback to ultimately investigate a simpler API.
- What kind of profiling & debugging tools do you recommend when using alpaka? Context: I imagine it to be a bit difficult to trace down where bottlenecks are in the single source if you run this on various architectures/backends. (Stephan)
    - You can use the native tools.
    - Commercially available: Vampir (not free of charge)
    - Use generic tools like Tau or Score-P
    - Better to use the native vendor tools, check caches or specific metrics.
- (May be too advanced / for a later session) I see that Alpaka uses a single queue type. I’ve experimented a bit with Vulkan as a GPU API that has multiple queue types, general purpose ones + special purpose ones for things like data transfers. Will Alpaka need to adapt if CUDA eventually exposes something like this? (Hadrien)

    - Will be (somewhat) covered in later days. A brief reply: should not be a big problem for alpaka. Alpaka already has queue properties and supports choosing a queue with desired properties. So it will probably be rather straightforward to add support for special-purpose queues by introducing more queue properties when needed.