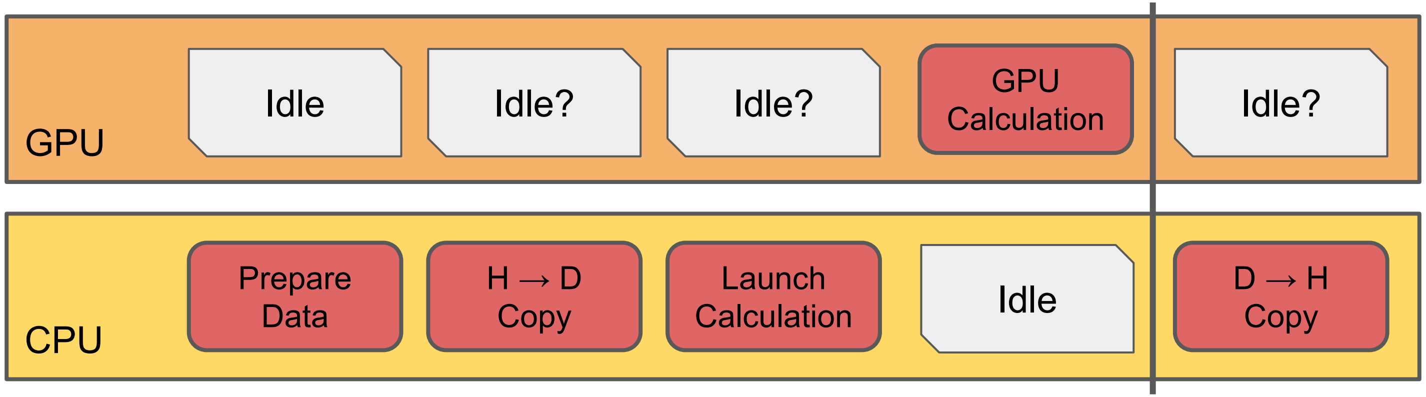
Heterogeneous Experimental Frameworks

Chris Jones, Kyle Knoepfel, Attila Krasznahorkay

HEP Software Foundation

Accelerator Programming



- The simplest accelerator/GPU applications do things fairly trivially
  - Copy input variable arrays to the accelerator/GPU memory
  - Launch a calculation on the accelerator/GPU
  - Wait for the calculation to finish
  - Copy the results back from the accelerator/GPU memory to the host
- Which can leave the CPU/accelerator idle for a fair amount of time…
  - However note that some of the operations are not as lightweight on the CPU as you may assume

Framework Organisation Categories

- Separate Processes
  - Multiple (many) processes cooperate in performing a job, possibly sharing some (conditions) data in memory, and sending event data between each other through some IPC method
  - Each process is expected to use just one CPU core/thread, and (directly) any accelerator that it needs
- Primary Accelerator Processing
  - The framework is designed primarily around the usage of (an) accelerator(s)
  - Execution orchestration on the CPU(s) is not a primary concern, as most things happen on the accelerator(s)
- Hybrid CPU/Accelerator Processing
  - A multi-threaded (CPU) process orchestrates the execution of sub-tasks (“modules” or “algorithms”), balancing efficient CPU and accelerator usage at the same time

Separate Processes

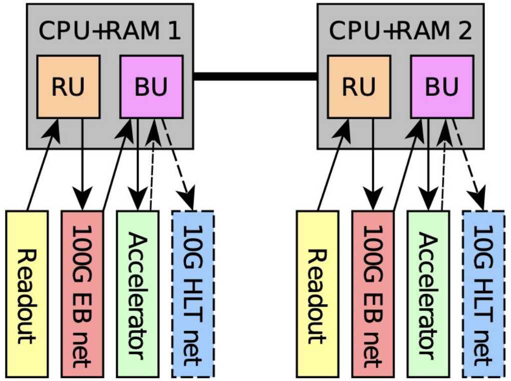
- Used by ALICE-FAIR (ALFA) for online reconstruction and data analysis.
- Parallelism achieved through separate processes, not multithreading, *per se*.
- Each process:
  - Executes tasks in its own memory space, avoiding thread-safety issues
  - Can access memory shared with other processes on the same node
  - Communicates with other devices/processes through (e.g. ZeroMQ) message queues
- Data is transferred between devices via the messages
- Offload to accelerators by launching another process on a GPU device
  - CPU load balancing across processes not a concern as vast majority of processing done on Accelerator

Separate Processes

- Pros:
  - Can use a synchronous API for the offloading
  - Possible to implement the offloaded calculations in a simpler way
- Cons:
  - Dynamic load-balancing is hard to achieve, ideal resource usage requires some sort of pre-run profiling

Primary Accelerator Processing

- Approach used by LHCb for their first trigger level (CHEP 2019 link)
  - Allen framework designed to support HLT execution on GPUs
- All memory allocated at start of application -- no dynamic memory management is allowed on the accelerator
- Configuring which steps to be executed happens at compile time
  - This is specific to Allen, a more dynamic configuration system is not impossible with this type of a framework
- One process running per core
  - Vast majority of computation done on Accelerator
    - CPU allowed to not be 100% occupied

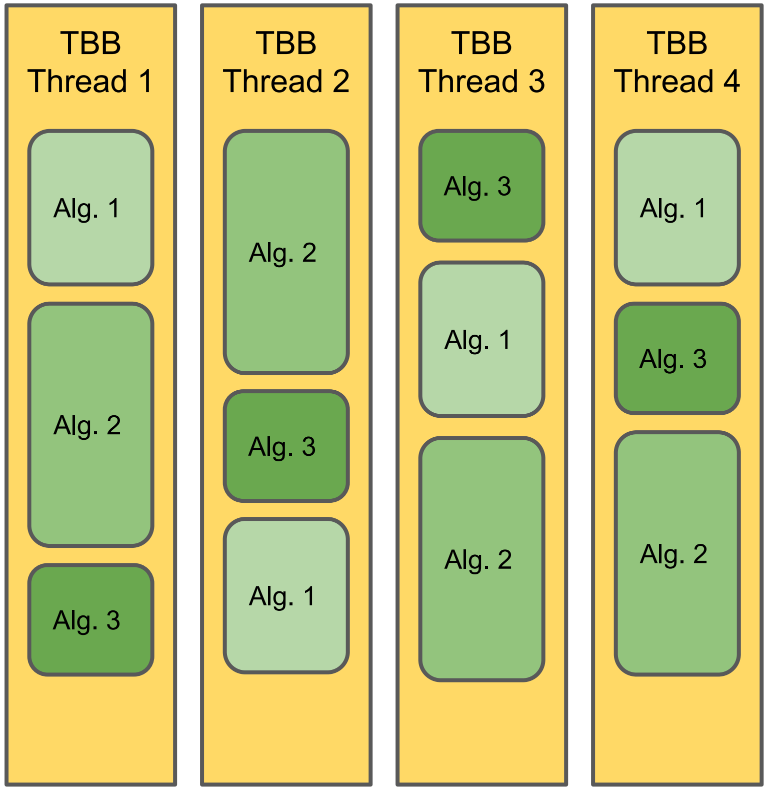


Primary Accelerator Processing

- Pros:
  - Code written to only work on a(n) (specific) accelerator can have a much better performance than code written to work both on CPUs and accelerators
  - Can rely on “hardware manufacturer solutions” for optimal accelerator code organisation
- Cons:
  - Code can only run on specific types of hardware
  - Learning curve for the developers can be steep

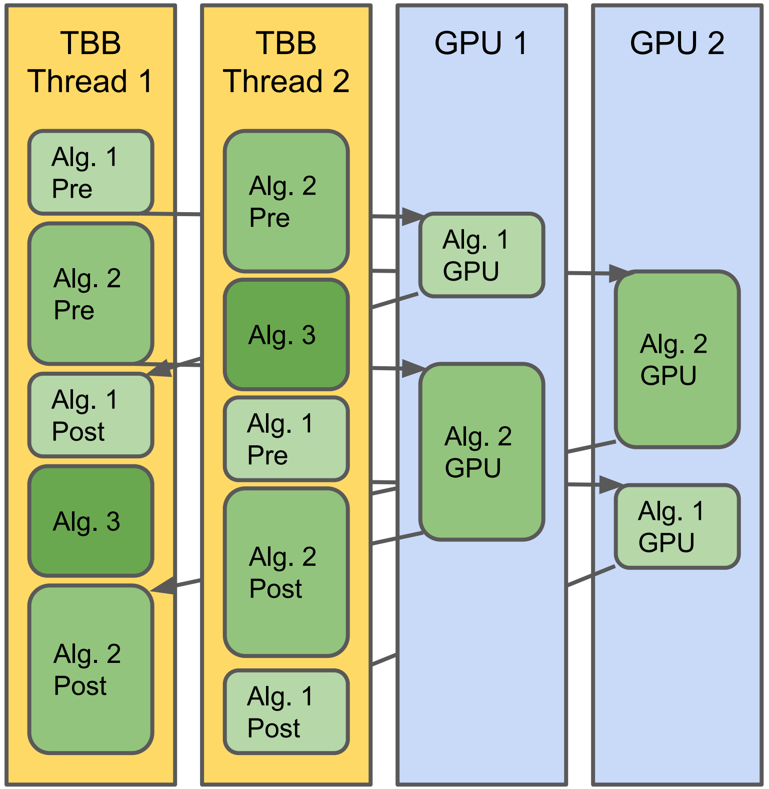
Hybrid Approach

- ATLAS and CMS process (relatively) large events in many, (relatively) long running steps
  - Running the different steps/modules/algorithms in parallel, using multiple (TBB) threads
- The execution of accelerated tasks can not be allowed to block TBB threads, as it would lead to wasted CPU resources
  - GPU calculations need to be launched asynchronously, and the framework needs to be notified about their completion -- so that tasks depending on them may be launched



Hybrid Asynchronous Execution (1)

- Accelerated modules need to be split into a pre- and post-execution step
  - For CPU-only operations that need to run before/after the accelerated calculation
- The goal is to let the CPU threads wait as little as possible
  - As long as the overhead from launching accelerated calculations and collecting their results is not significant, even not-super-efficient accelerated calculations speed up the overall job this way
- It is up to the framework to launch (CPU) tasks in the best way to maximise the usage of all available resources
  - Which is of course a fairly non-trivial task to do…



Hybrid Asynchronous Execution (2)

- There are also some other possibilities…
- The framework may not need to know explicitly about callbacks from accelerated calculations if:
    - An accelerated module/algorithm is launched in an "oversubscribed" std::thread, in which it launches the calculation, and then waits for its results using a synchronisation point
    - If the pre-/post-execution steps are CPU intensive, they can still be "outsourced" into TBB tasks, which would coordinate with the other calculations launched by the framework
- This setup's performance can be similar to the one explained previously, in some simple artificial tests…

Hybrid Asynchronous Execution (3)

- Pros:
  - Can be used with very large applications to efficiently use all the resources of the underlying machine (through memory sharing using MT)
  - Using a smart MT scheduling system (like TBB or HPX) can make it possible to maximise the CPU usage of the job
- Cons:
  - Ideal parallel asynchronous execution of accelerated tasks, launched from multiple threads, is not a widely needed use-case. Accelerator APIs generally don’t natively support this setup.
  - The user code has to be written with asynchronous execution in mind, which can be a steeper learning curve for the developers

Summary

- Different kinds of software frameworks allow the implementation of accelerated calculations in quite different ways
  - As there’s no size that would fit all experiments for the CPU-only software frameworks, this is even more true when it comes to accelerators
- For a new project the choice should be made along multiple guidelines
  - If the implemented calculations are easily vectorisable, the framework should be okay to use synchronous APIs for offloading calculations
    - If not, care needs to be taken not to impede the CPU performance
  - If the applications don’t need a lot of memory, multi-processing can be a lot easier to work with than multi-threading
  - Different implementations may also be preferred depending on the “portability” technology being used for the offloading