Heterogeneous particle tracking with SYCL

27.01.2023

Bartosz Soból

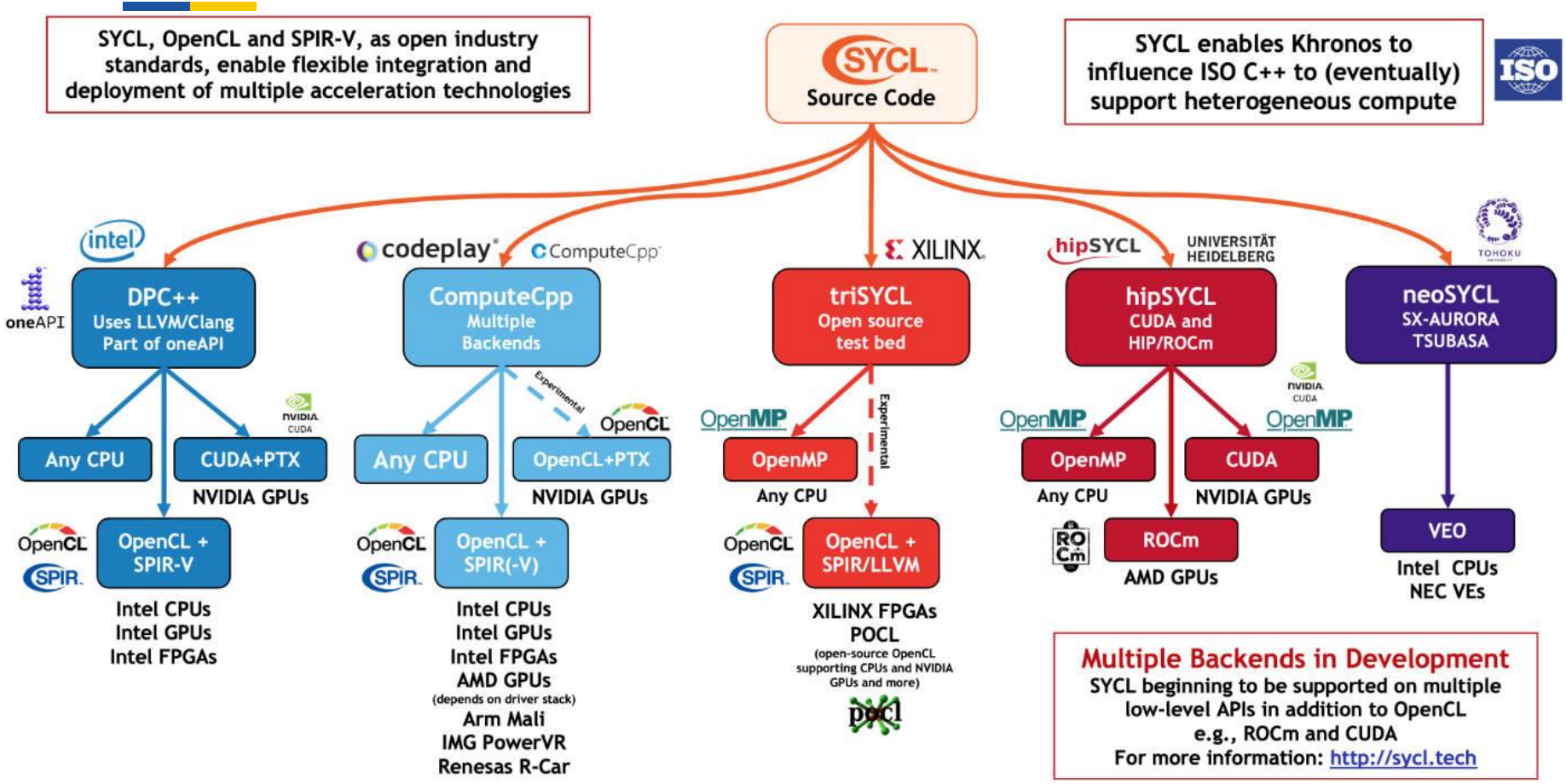
What is SYCL?

- Open standard, higher-level heterogeneous programming model - CPU, GPU, FPGA, ...
- Based on standard C++11 and newer - without language extensions
- Single source for host and kernel/device code
  - Tools for C++ work with SYCL (IDEs, static analysis, linters, formatters, …)
  - Kernel == any callable (function, lambda, function object)*
  - C++ functions called by kernel are also compiled as a part of device code
  - Implicit device-host separation
- Implicit memory management and task scheduling
- OpenCL concepts reused (context, device, queue, memory layers, …)

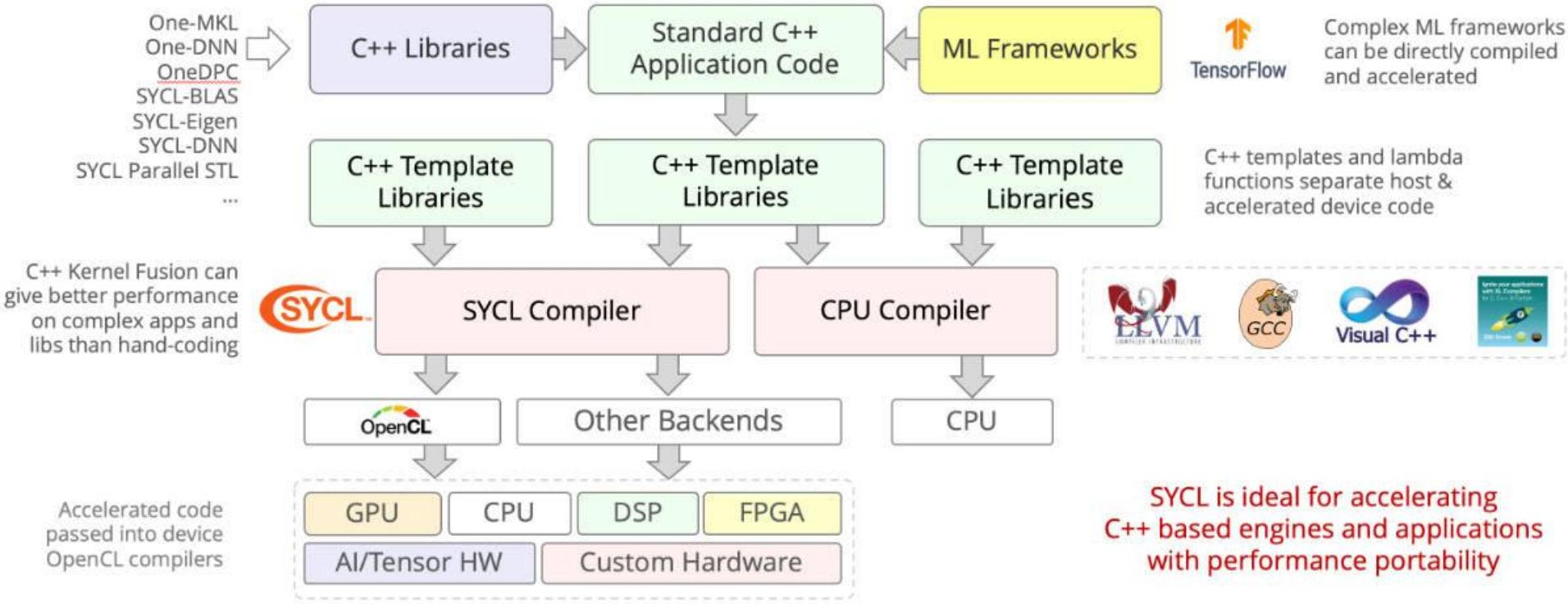
Implementations

- Intel is investing a lot in SYCL (oneAPI, DPC++)
  - SYCL is the default programming model for new Intel hardware
  - Supports Intel hardware - CPUs, GPUs, FPGAs with low-level optimisation extensions
  - Additional NVIDIA and AMD GPUs support by Intel/Codeplay now official
- hipSYCL - developed as a research project at Heidelberg University
  - Supports CPUs (OpenMP), GPUs - AMD, NVIDIA, (Intel)
  - Successfully used on AMD supercomputers (LUMI, Frontier) - GROMACS software
  - Interesting concept - plugin/additional layer for existing compilers
- AMD/Xilinx is also working on SYCL (triSYCL, fork of Intel's DPC++) for their FPGAs (Alveo)
  - Small research-like project, 2-3 devs part-time
- Codeplay's ComputeCpp - first SYCL compiler, now less relevant, bought by Intel
- Celerity - SYCL-like MPI-SYCL wrapper

Implementations



Ecosystem


One-MKL
One-DNN
OneDPC
SYCL-BLAS
SYCL-Eigen
SYCL-DNN
SYCL Parallel STL
...
C++ Libraries
Standard C++ Application Code
ML Frameworks
TensorFlow
Complex ML frameworks can be directly compiled and accelerated
C++ Template Libraries
C++ Template Libraries
C++ Template Libraries
C++ templates and lambda functions separate host & accelerated device code
C++ Kernel Fusion can give better performance on complex apps and libs than hand-coding
SYCL
SYCL Compiler
CPU Compiler
LLVM
GCC
Visual C++
OpenCL
Other Backends
CPU
Accelerated code passed into device OpenCL compilers
GPU
CPU
DSP
FPGA
AI/Tensor HW
Custom Hardware
SYCL is ideal for accelerating C++ based engines and applications with performance portability

Example: vector addition

```
// vadd.cpp
 1  #include <SYCL/sycl.hpp
 2  #include <array>
 3  int main() {
 4    constexpr int SIZE = 4;
 5    std::array<int, SIZE> vec_a{1, 2, 3, 4}, vec_b{5, 6, 7, 8}, vec_c;
 6
 7    sycl::queue queue{sycl::gpu_selector()};
 8    sycl::range<1> rng{SIZE};
 9    {
10      sycl::buffer<int, 1> a_buff{vec_a.data(), rng};
11      sycl::buffer<int, 1> b_buff{vec_b.data(), rng};
12      sycl::buffer<int, 1> c_buff{vec_c.data(), rng};
13      queue.submit([&](sycl::handler &cgh) {
14        const sycl::accessor a_acc{a_buff, cgh, sycl::read_only};
15        const sycl::accessor b_acc{b_buff, cgh, sycl::read_only};
16        sycl::accessor c_acc{c_buff, cgh, sycl::write_only, sycl::no_init};
17
18        auto kernel = [=](sycl::id<1> id) {
19          c_acc[id] = a_acc[id] + b_acc[id];
20        };
21
22        cgh.parallel_for(rng, kernel);
23       });
24     } // vec_c == {6, 8, 10, 12}
25  }
```

Example: compilation (hipSYCL, CMake)

```
1  # CMakeLists.txt
2
3  project(sycl_vadd LANGUAGES CXX)
4
5  set(HIPSYCL_TARGETS "omp;cuda:sm_70" CACHE STRING "cpu|rocm:gfxXXX|cuda:sm_XX" FORCE)
6
7  find_package(hipSYCL CONFIG REQUIRED)
8
9  add_executable(sycl_vadd vadd.cpp)
10 add_sycl_to_target(TARGET sycl_vadd SOURCES vadd.cpp)
11
```

Main kernel code limitations

- C++ features not allowed in kernel/device code:
  - Dynamic memory allocation
  - Recursion
  - Exception handling
  - Function pointers
  - Virtual function calls
  - RTTI
  - ...
- Kernel must return `void`
- In `sycl::buffer<T, Dim>`, type `T` must be trivially copyable

How to try (hip)SYCL?

- hipSYCL supports wide range of devices
  - https://github.com/illuhad/hipSYCL/blob/develop/doc/installing.md
- Can be installed from repository
  - CUDA/ROCm must be installed separately (if needed)
- Or built from sources
  - Requirements:
    - C++17 compiler, CMake, python3,
    - boost (fiber, context), recent LLVM and Clang (>10),
    - CUDA/ROCm,

Project overview

- Track reconstruction algorithm for PANDA Forward Tracker
  - Two different procedures: for free particle and in EM field
- Investigating possibilities for online processing
- Using heterogeneous computing platforms
  - Multicore CPU, GPU, AMD/Xilinx Alveo FPGA
  - What platform and type of accelerator performs best?
- We’ve chosen SYCL programming model for our software

Project status

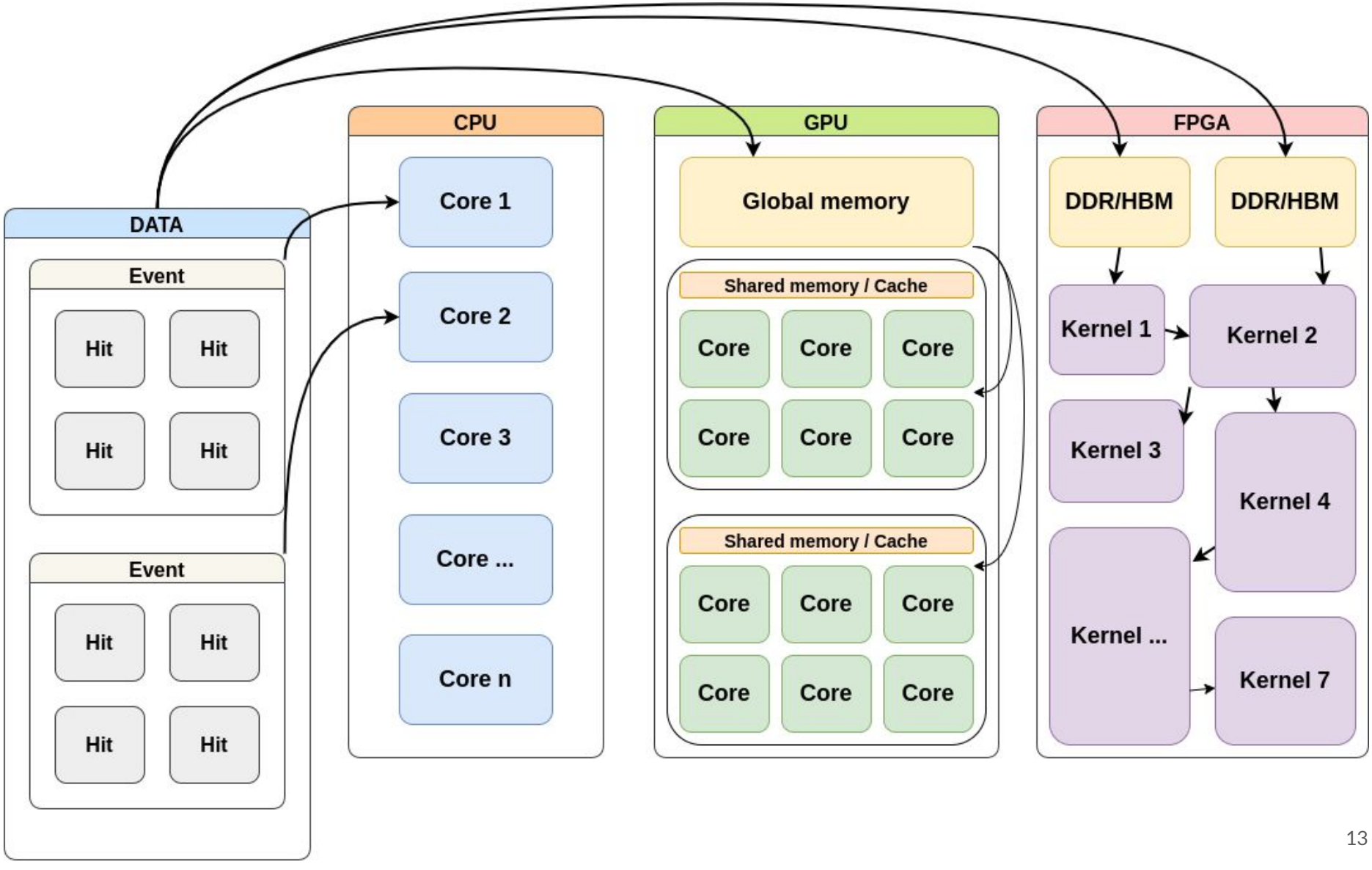
- Whole track reconstruction algorithm implemented
  - In plain C++ (single threaded)
  - In SYCL - 7 Kernels + helper functions, ~1.5k lines of accelerated code
- SYCL implementation was tested and benchmarked
  - With three SYCL implementations - hipSYCL, DPC++ and AMD/Xilinx's triSYCL/sycl*
  - On different Intel and AMD CPUs
  - AMD and NVIDIA GPUs - Tesla V100, A100, Instinct MI250
  - AMD/Xilinx Alveo U280 FPGA*
  - All with single source code**

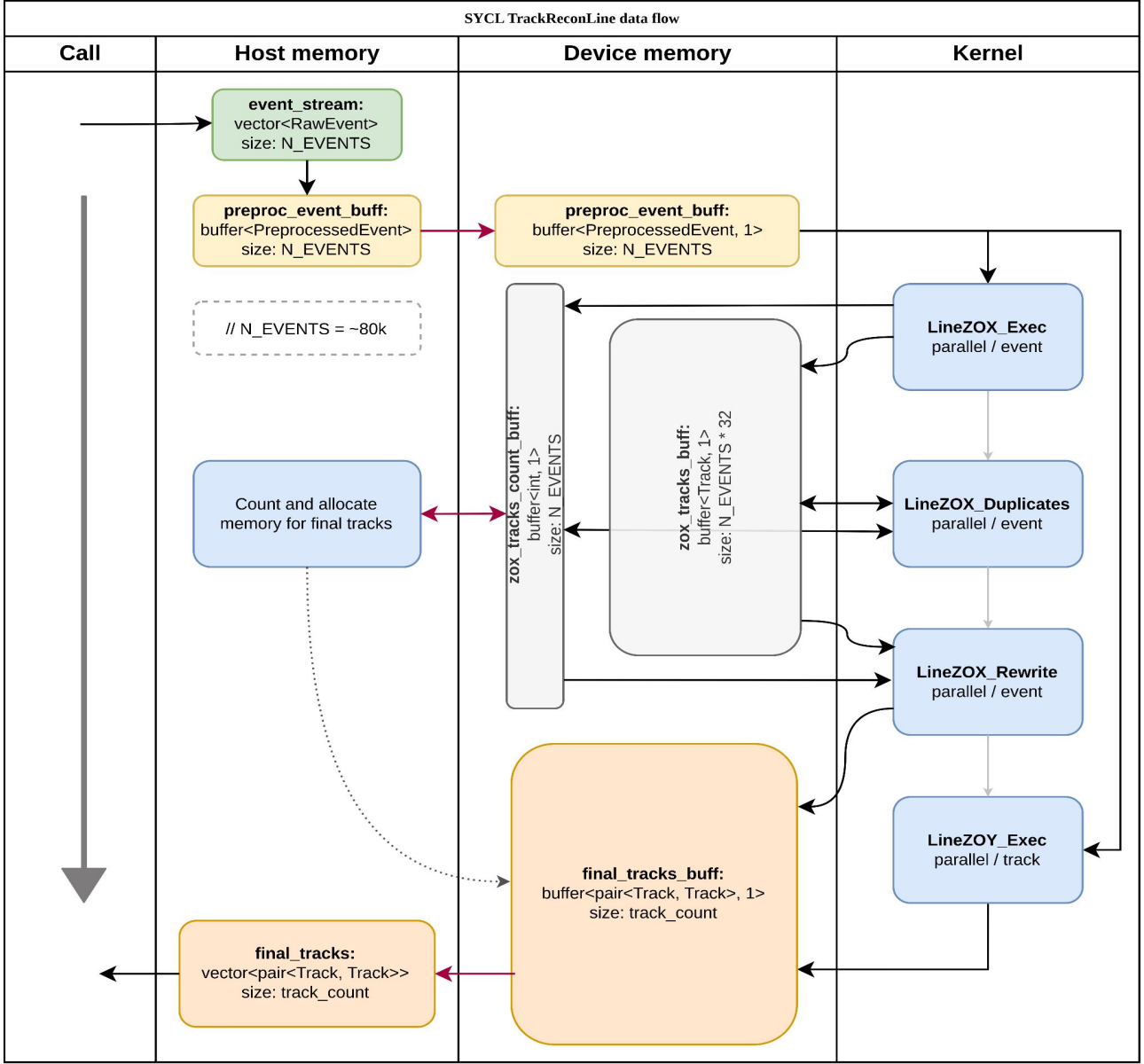
*partially

**triSYCL required minor changes in ~40 lines of host code

SYCL implementation strategy

- I started with single threaded C++ code
- Which was adapted to SYCL with minimal effort
- Than introduced optimisations with as small as possible code changes
  - Mostly data-flow and memory layout changes
- Algorithm (kernel) code stayed very similar to pure C++ version


DATA
Event
Hit
Hit
Hit
Hit
Event
Hit
Hit
Hit
Hit
CPU
Core 1
Core 2
Core 3
Core ...
Core n
GPU
Global memory
Shared memory / Cache
Core
Core
Core
Core
Core
Core
Shared memory / Cache
Core
Core
Core
Core
Core
Core
FPGA
DDR/HBM
DDR/HBM
Kernel 1
Kernel 2
Kernel 3
Kernel 4
Kernel ...
Kernel 7


SYCL TrackReconLine data flow
Call
Host memory
Device memory
Kernel
event_stream:
vector<RawEvent>
size: N_EVENTS
preproc_event_buff:
buffer<PreprocessedEvent>
size: N_EVENTS
preproc_event_buff:
buffer<PreprocessedEvent, 1>
size: N_EVENTS
// N_EVENTS = ~80k
zox_tracks_count_buff:
buffer<int, 1>
size: N_EVENTS
zox_tracks_buff:
buffer<Track, 1>
size: N_EVENTS * 32
LineZOX_Exec
parallel / event
Count and allocate
memory for final tracks
LineZOX_Duplicates
parallel / event
LineZOX_Rewrite
parallel / event
final_tracks_buff:
buffer<pair<Track, Track>, 1>
size: track_count
LineZOY_Exec
parallel / track
final_tracks:
vector<pair<Track, Track>>
size: track_count

Computing performance

Execution time (average from simulation events of 1,3,5,8 muons) without mom est **[us/event]**:

Implementation / device type	Device	Batch 10k events	Batch 20k events	Batch 80k events	Batch 160k events
C++ single thread / CPU	Xeon E5-2667 v4				~60
hipSYCL@OpenMP / CPU	**Xeon E5-2667 v4**	**10.7**	**8.6**	**7.4**	**7.3**
hipSYCL@OpenMP / CPU	**Xeon Platinum 8268**	**7.7**	**7.0**	**6.7**	**6.6**
hipSYCL@OpenMP / CPU	**EPYC 7742**	**8.2**	**7.5**	**6.3**	**5.5**
hipSYCL@OpenMP / CPU	**EPYC 7763**	**4.5**	**4.0**	**4.0**	**4.0**
hipSYCL@CUDA / GPU	**RTX 2080 Ti**	**10.7**	**8.6**	**7.4**	**7.3**
hipSYCL@CUDA / GPU	**Tesla V100 SXM**	**10.4**	**8.1**	**6.9**	**6.7**
hipSYCL@CUDA / GPU	**Tesla A100 SXM**	**9.3**	**6.9**	**5.3**	**5.0**
hipSYCL@HIP / GPU	**Instinct MI250**	**14.8**	**9.6**	**6.6**	**6.3**
triSYCL@HLS / FPGA	Alveo U280				TBD

Computing performance

- Algorithm itself isn't ideal for GPU performance
  - Lot's of branches and not parallelizable loops
  - More data-bounded than compute-heavy
- CPU parallelization is quite good
  - AMD EPYC: running e.g. 4 16 core runs instead of single 64 core would probably result in higher throughput
- GPU performance is OK, but not great
  - Increases with larger *batch* size
  - In should be possible to further fine-tune for GPU performance
  - GPU optimisations didn't negatively affect CPU performance so far

Computing performance

- Performance evaluation with Intel's DPC++ compiler is a TODO
- Native CUDA implementation exists, but is outdated
  - GPU performance was on par with SYCL when last checked
  - In general, (hip)SYCL is few percent behind or in par with CUDA, depending on use case
- FPGA is expected to be slow (Pentium III kind-of-slow)
  - Running such high-level software on FPGA is an achievement itself
  - More FPGA-specific optimisation options are available for Intel hardware
    - Require code changes for FPGA, but still uses the same API as base

Conclusions

- While this algorithm isn't great for GPU, it's a good demonstrator of SYCL
- SYCL can be used to build software for heterogeneous platform
  - With single programming model / API
- I GPU fails to deliver performance, parallel CPU version is *free*
- Hardware in the GreenCube will change
  - Before CBM start (and certainly before PANDA)
  - SYCL code will most probably work on Virgo successor
- Resources saved