Detray on FPGA

Jasmijn Bookelmann
Main supervisor: *Julian Wollrath*
Secondary supervisor: *Joanna Niermann*

Abstract
This report covers the process of implementing the ACTS `detray` library on an AMD FPGA accelerator using Vitis HLS. The two main issues encountered are C++ support, as Vitis HLS supports C++14 but `detray` is written in C++17. Another issue is the difference in memory models of a GPU compared to an FPGA. After solving these issues, it is shown to be possible to load data onto the FPGA, show statistics about this data, and perform local-to-global transformation of coordinates.

1 Introduction

This report covers the process of the ACTS `detray` library [1], [2] on an AMD FPGA accelerator card using Vitis HLS.

`detray` is a header-only C++ library providing a tracking detector description compatible with GPU accelerators. Another type of accelerator are Field Programmable Gate Arrays (FPGAs). FPGAs can be considered reprogrammable hardware: given a circuit description, the FPGA 'becomes' this circuit. Having custom hardware for specific applications allows for low energy consumption and provides opportunities for strong concurrency.

In order to synthesize `detray` to an FPGA compatible circuit description, High Level Synthesis (HLS) is used. HLS translates C++ code to a circuit description. Vitis HLS is a specific HLS developed by AMD. This means it is compatible with the AMD FPGA accelerators.

In this report, first the main concepts are discussed, next an explanation of which steps have been taken in order to get `detray` on an FPGA and what has been accomplished. Finally, it is concluded whether it is viable to have a full working `detray` implementation for FPGA.

2 Background

2.1 Vitis HLS

Vitis HLS is the toolchain used to translate C++ code into RTL and compile this to a binary, which can be loaded onto the FPGA. A Vitis program is split into two parts: the host program, which runs on the host device, usually a CPU. And the kernel program, which is translated into an FPGA circuit description and loaded onto the FPGA.

The host and kernel code are compiled separately, then the host code loads the kernel binary and data onto the FPGA and signals the FPGA to start.

The compilation of C++ kernel code into hardware description is called synthesis. This hardware description only determines the behavior of the FPGA kernel, not specifying how this kernel interfaces with the host. This is performed by linking the hardware description together with a platform provided by Vitis are used. The generated a binary can be loaded onto the FPGA.

Development on FPGA is different from that on GPU. FPGA programming allows for a lot more flexibility, as FPGAs allow for custom hardware, while that of a GPU is fixed. However, this also makes FPGA programming more difficult: More properties need to be explicitly defined, and hardware design

is fundamentally different from the instructions-based sequential programming that C++ is designed for. Because of this, HLS tools have limited support for C++ features and patterns.

2.2 Detray

`detray` is a C++ library providing a detector description. `detray` provides a framework allowing definitions of how detectors components are structured, and tools to reconstruct particle tracks in this detector. `detray` strongly depends on compile-time polymorphism, instead of the alternative ACTS runtime polymorphism and pointer-logic. This allows it to be implemented on GPUs.

`detray` provides both host and device code, and attempts to reuse core classes for both.

The memory backend for `detray` is called `vecmem` [3], which provides interfaces to allocate, store and access memory.

3 Implementation

In order to use `detray` on an AMD FPGA, multiple issues had to be resolved. In this section these are discussed step-by-step.

3.1 Including `detray` in the kernel

3.1.1 C++ Support

The first step is to actually be able to include the in kernel code. In more concrete terms, have the following statement in the kernel source and be able to synthesize this kernel.

```
#include "detray/core/detector.hpp"
```

However, this is not as straightforward as it seems: Older versions of `detray` are written in C++17, with the current version using C++20 features. However, Vitis HLS only supports synthesizing C++14. This means it is necessary to downgrade `detray` and any other external libraries the kernel uses such that it can be compiled using C++14.

As a first step, in order to avoid C++20 concepts, `detray` version v0.72.0 is used instead of the current release. This version is still written is C++17, so does not include any C++20 features.

In addition to this, all host-only functions are removed. Host-only functions or structures are functions marked by `DETRAY_HOST` before the function signature. These are then translated into CUDA qualifiers such that the GPU does not call these. Vitis HLS does not support such qualifiers, so instead these functions have to be surrounded by header guards as follows:

```
#ifndef DETRAY_COMPILE_VITIS
... // host-only function
#endif
```

This is performed using a python script which estimates where the functions begin and end and inserts the guards. The python script can be applied to all the libraries if necessary and can be found in the repository [4].

With the now narrower scope the following step is to remove the remaining unsupported C++17 features.

One widely-used C++17 feature in `detray` is the use of the shorthand '`[type_trait]_v<...>`' instead of '`[type_trait]<...>::value`'. This pattern has also been replaced using a python script to parse and rewrite all files.

In addition to this, CTAD (Class template argument deduction) is also a commonly used C++17 feature in `detray`. Constructors using this feature included by the kernel have been removed by hand. It

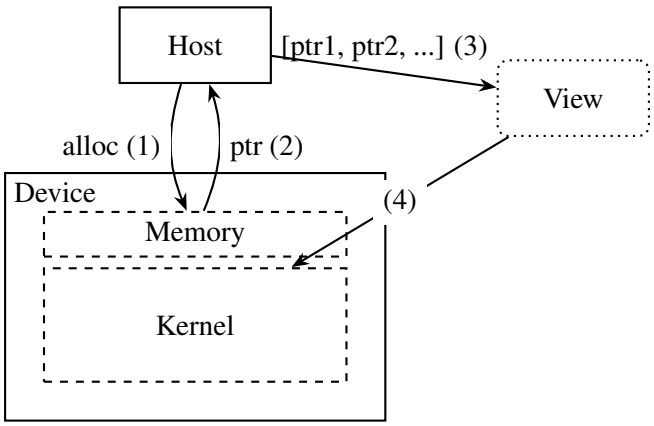

Figure 1: Schematic of loading of the detector description using `detray` onto a GPU

is possible that the removal of the constructors is an issue if more extensive code has to be implemented on the FPGA, then a constructor factory pattern can be used as replacement.

Lastly, references to the `vecmem` polymorphic allocator have to removed. This is done by replacing the polymorphic allocator types by normal arrays and vectors. The polymorphic allocator is not actually used by the device, so the exact implementation of these is not relevant. However, it should also not be included as this results in errors.

A lot of C++17 features are able to be compiled, given warnings. It can be assumed that this is properly handled by the compiler.

With these modifications, it is possible to include `detray` in the kernel.

3.2 Adjusting memory to Vitis HLS

It is necessary to be able to actually get the detector data onto the FPGA.

Currently `detray` does the following for GPUs, as visualised by Figure 1. When the hosts needs to allocate data for the detector description, it requests an allocation in GPU memory (1). The GPU memory allocates the buffer and returns a pointer to this (2). These pointers and then collected and put into a struct called the 'view' (3). Finally, this view is passed to the kernel as argument, allowing the kernel to access the allocated data (4).

However, this approach is not as straightforward on an FPGA accelerator card. Memory cannot be allocated directly in global memory and then accessed by pointers.

An FPGA accelerator has multiple sources of memory and different ports to access these, there is global memory (e.g. DDR or HBM) which is used to interface with the host. There can be multiple ports to global memory, each of which have their own address space. In addition to this, there is local memory (e.g. BRAM or registers), which is on the FPGA itself. These each have different methods of access and need to be differentiated at compile time.

This is why is not possible to access data just by using a pointer. The pointer does not contain any information about which of the different memories it refers to. To explicitly describe these different memory buffers, arrays or pointer parameters are used. Data is then accessed by indexing into these buffers.

In order to pass global memory buffers to the kernel in Vitis HLS, each allocation needs to be passed to the kernel as an array or pointer; this can be seen in Figure 2. It is not feasible to create a separate kernel argument for each. `detray` makes numerous allocations, e.g. for a small toy example more than 20 are made. Allocation is also performed at runtime, while the kernel arguments would need to be manually written or generated.

```
1        void kernel_main(int * in, int * out) {...}
```

Figure 2: Example kernel interface showing how data is passed to the kernel in Vitis HLS

Another issue is passing the device view, this is a class possibly containing pointers. Vitis HLS does not have good support for passing non-scalar data as kernel arguments, and explicitly does not allow the passing of C++ `structs` with pointers.

3.2.1 Passing the device view

It is not possible to directly pass the device view. This has been experimentally verified.

In order to solve this, the device view is cast to a byte array (`uint8_t *`), passed to the kernel and cast back into a device view. This prevents Vitis HLS from assuming that any pointers in the device view should create an allocation. The byte array type is just there to signify to the kernel that raw data needs to be passed. This data can then later be interpreted as different structures.

Using the device view, it is possible to print certain statistics on the FPGA such as how many rectangular detector surfaces are in the description transferred onto the FPGA accelerator card.

3.2.2 Solving the allocation and device view

Instead of allocating directly in memory of the device, a local buffer on the host is used. This is depicted in Figure 3.

The host makes allocations in a local memory buffer located on the host (1). The allocator returns the relative pointer (Δptr) from the start of the buffer (2). This means that e.g. the first allocation will be at pointer value 0. These relative pointers are used to construct the view (3). Finally, both the buffer and view are cast to byte arrays, allocated in memory and passed to the kernel (4).

In the kernel, the buffer is read from global memory into a statically allocated local buffer. The underlying vector types `detray` uses is rewritten such that instead of using pointers it indexes into this locally allocated buffer.

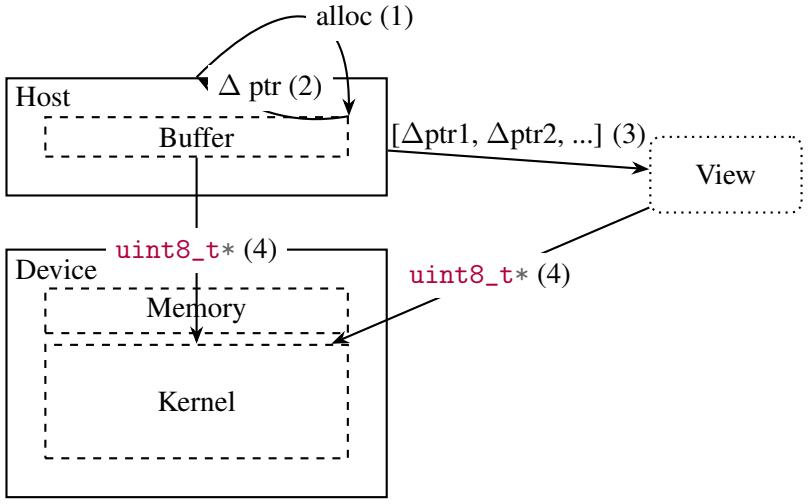

Figure 3: Schematic of loading the detector description onto the FPGA accelerator

This approach solves multiple problems:

– It is explicitly stated from which memory buffer the kernel must read the data, instead of just having a pointer to somewhere.

– Vitis technically does not allow 'pointer-to-pointer'. As documented in the Vivado HLS [1] reference manual [5], it is only possible to have pointers to either scalars, or arrays of scalars. However, `detray` containers do contain pointers. By using a global buffer instead of pointers directly, there is an 'index-to-index' structure, which is allowed.

– By passing the whole buffer containing multiple different objects as a single byte array, multiple different objects can be allocated in memory without this having to be explicitly defined as a kernel argument.

For implementing this approach, the underlying memory library, `vecmem`, had to be adjusted. First of all, an allocator had to be written which returns relative pointers instead of absolute ones. This allocator takes a buffer and incrementally allocates memory to it. De-allocation is not supported, but could theoretically be implemented in future work.

Secondly, the device vector class had to be adjusted. Instead of using pointers directly, data had to be read relative from the input buffer. The input buffer is defined as a static array in the `vecmem` namespace with a given size (`static vecmem::memory_buffer[DATA_SIZE]`). This means that when including `vecmem`, this array is created and filled with data, then the device vectors use this data. Given a relative pointer, which is described in subsubsection 3.2.2, the device vector calculates the offset from the buffer and return a reference to the object at that position.

With this implementation, it is possible to perform local-to-global coordinate transformation on the FPGA accelerator.

3.2.3 Exploring the use of pointers directly

It is possible to access values using pointers with Vitis HLS. Statements such as `int a = *12` successfully synthesize. However, when experimenting with this approach, which of the memory buffers is accessed, is undefined behaviour and not predictable. Hence, this approach is not viable.

4 Conclusion

In conclusion, it is possible to transfer detector descriptions onto FPGA accelerators and perform simple calculations such as local-to-global coordinate transformations.

The primary challenges are the C++ support and the difference in memory model compared to GPUs. The C++ support is manageable with an older version of `detray`, however gets increasingly difficult with newer versions using more modern C++ features. Because of this, long-term maintenance is difficult, as two separate versions of `detray` and supporting libraries would have to be maintained. Either usage of newer features should be avoided in further development, or the two different versions will be increasingly distinct.

The differences in memory model between GPUs and FPGA accelerators can be solved using a local buffer which is copied over, and mostly includes rewriting the memory backend `vecmem` without any need to adjust `detray` itself. This is not an obstacle to long-term maintenance and further development of `detray` on FPGA.

[1] Vivado HLS is the predecessor of Vitis HLS, there is no documentation about this error in the Vitis reference manual, only in older Vivado ones

5 Discussion and future work

Possible future work includes getting more `detray` features onto the FPGA, such as propagation.

There is a possibility that eventually Vitis HLS will accept a C++17, or higher, front-end, however there has been no news of this as of August 2025. It is possible to explore other HLS tool chains, for example Intel HLS does support C++17 [6].

References

[1] Xiaocong Ai et al. 'A Common Tracking Software Project'.
In: *Computing and Software for Big Science* 6.1 (Apr. 2022), p. 8. ISSN: 2510-2044.
DOI: 10.1007/s41781-021-00078-8.

[2] *Acts-Project/Detray (Version 0.72.0)*. https://github.com/acts-project/detray. July 2024.

[3] *Acts-Project/Vecmem (Version 1.4.0)*. https://github.com/acts-project/vecmem. Mar. 2024.

[4] *JasmijnB/Detray-Fpga*. https://github.com/JasmijnB/detray-fpga. Aug. 2025.

[5] *Vivado Design Suite User Guide: High-Level Synthesis UG902 (v2020.1)*. May 2021.

[6] *Overview of the Intel® High Level Synthesis (HLS) Compiler Pro Edition*.
https://www.intel.com/content/www/us/en/docs/programmable/683456/24-1/overview-of-the-pro-edition.html. (Visited on 22/08/2025).