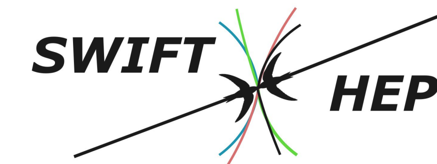


Activities on SWIFT-HEP: Running a cmssw algorithm on an FPGA

Alison Elliot *(STFC-RAL)*
Sam Harper *(STFC-RAL)*

Science and Technology Facilities Council

SWIFT HEP

FPGA card

Specification	U250	
	Active Cooling Version	**Passive Cooling Version**
Product SKU	A-U250-A64G-PQ-G	A-U250-P64G-PQ-G
Thermal cooling solution	Active	Passive
Weight	1122g	1066g
Form factor	Full height, full length, dual width	Full height, ¾ length, dual width
Total electrical card load[1]	215W	
Network interface	2x QSFP28	
PCIe Interface	Gen3 x16	
Look-up tables (LUTs)	1,728K	
Registers	3,456K	
DSP slices	12,288	
UltraRAMs	1,280	
DDR total capacity	64 GB	
DDR maximum data rate	2400 MT/s	
DDR total bandwidth	77 GB/s	

Science and Technology Facilities Council

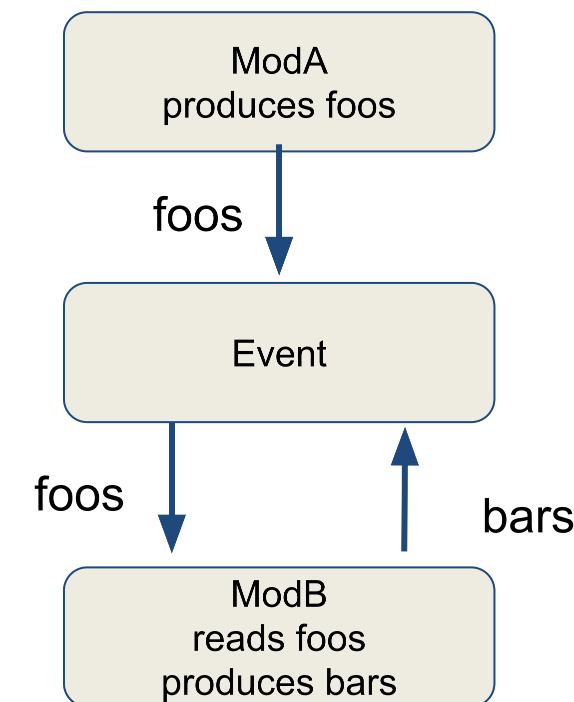
running kernels on the FPGA

SWIFT HEP

- To run a bit of code on an FPGA, we need to translate it into something that the FPGA will understand, and that can be called externally
- Using Vitas HLS to translate chunks of code into FPGA-readable binaries or kernels
- These binaries can then be used alongside c++ code, including them as extra data
- The Xilinx FPGA boards have 'runtime libraries' (xrt), enabling them to be included inside of a regular c++ job, and can run over the binaries that the HLS has translated from c++

CMSSW Primer

- CMSSW is the CMS software responsible for running all reconstruction/HLT/analysis jobs in CMS
- operates on a event data model, a CMSSW job consists of **multiple independent modules** which **produce**, **filter** and **analyse** products in the event
  - a product is some collection of data, eg EcalRecHits
  - modules can only communicate by reading/writing immutable products to/from the event
- fully multithreaded since 2015
  - has both inter event and intra event multithreading
  - intra module multithreading is also possible
  - typically runs with a large number of streams with each stream processing an event so multiple events are processed in parallel
- has full support for offloading to GPUs and other devices



Science and Technology Facilities Council

Set up on CMSSW for FPGA Kernels

SWIFT HEP

- Communication is done by the Xilinx Runtime Library (XRT)
  - https://www.xilinx.com/products/design-tools/vitis/xrt.html
  - https://github.com/Xilinx/XRT Apache 2.0 licence
- allows c++ programs to call FPGA kernels loaded on the FPGA
- pure c++, ***treat like any other external tool*** in CMSSW
  - cavate, it needs to be installed under /opt/xilinx/
- steps to include in CMSSW
  - 1) compile xrt with appropriate gcc version to allow linking against CMSSW release (currently gcc11 on slc7)
  - 2) tell CMSSW build system to link against the compiled libraries and where to find the headers

```
<use name="FWCore/Framework"/>
<use name="DataFormats/EcalRecHit"/>
<flags CXXFLAGS="-I/opt/xilinx/xrt_slc7_amd64_gcc11//opt/xilinx/xrt/include/ -L//opt/xilinx/xrt_slc7_amd64_gcc11/opt/xilinx/xrt/lib -lxrt_coreutil"/>
<flags EDM_PLUGIN="1"/>
```

package build file, currently hacking it by hard coding the locations for now

will obviously need to integrated it into the scram build system

Science and Technology Facilities Council

SWIFT HEP

Running FPGA Kernels in CMSSW

- CMSSW has the concept of "ExternalWork" which allows offloading to a device
  - https://twiki.cern.ch/twiki/bin/view/CMSPublic/FWMultithreadedFrameworkGlobalModuleInterface#edm_ExternalWork
  - currently used for CMS GPU modules, but applicable to any device
- has a special "acquire" function which sends non-blocking work to device
- once this has completed, the produce function places it in the event
  - "normal" modules would only have the produce function

```
class RALFPGATestProducerPhII : public edm::stream::EDProducer<edm::ExternalWork> {
public:
  explicit RALFPGATestProducerPhII(edm::ParameterSet const& ps);
  ~RALFPGATestProducerPhII() override = default;
  static void fillDescriptions(edm::ConfigurationDescriptions&);

private:
  void beginStream(edm::StreamID) override;
  void acquire(edm::Event const&, edm::EventSetup const&, edm::WaitingTaskWithArenaHolder) override
;
  void produce(edm::Event&, edm::EventSetup const&) override;

};
```

Running FPGA Kernels in CMSSW

- initialize FPGA:

```
void RALFPGATestProducerPhII::beginStream(edm::StreamID){
  device_ = xrt::device(deviceIndex_);
  edm::FileInPath xclbinName("RALFPGATest/FPGATest/data/varr.xclbin");
  uuid_ = device_.load_xclbin(xclbinName.fullPath());
}
```

- create a kernel instance to call

```
auto krnl = xrt::kernel(device_, uuid_, "varr");
```

- create a buffer object to manage host/device memory

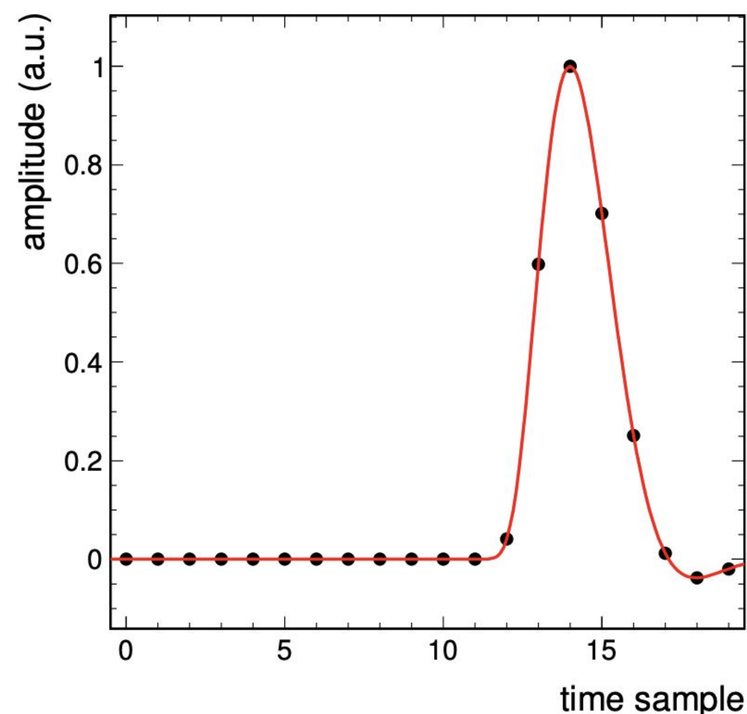
```
auto boIn0a = xrt::bo(device_, vector_size_bytes, krnl.group_id(0));
```

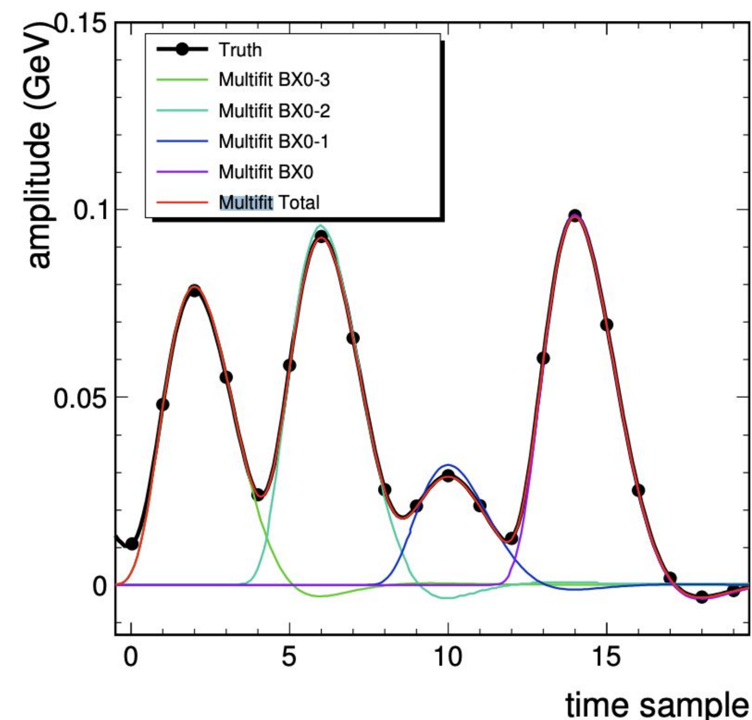
- sync data to FPGA, run and sync back the results

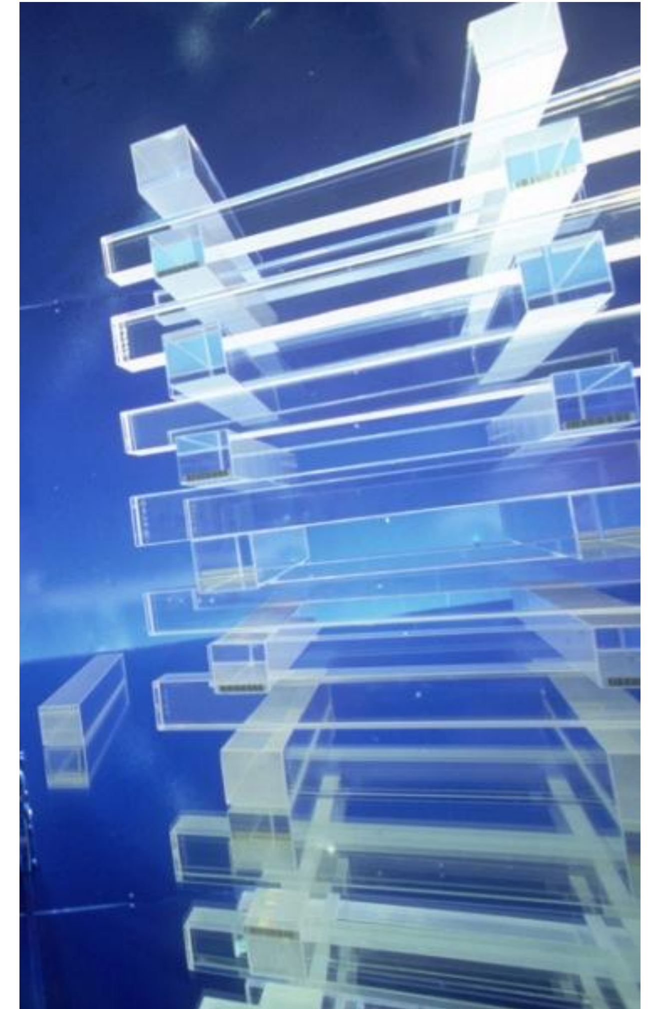
```
boIn0a.sync(XCL_BO_SYNC_BO_TO_DEVICE);
auto run = krnl(boIn0a, boIn1a, boIn2a, boIn0t, boOutA, boOutT, boOutG, nDigis, nSamples);
run.wait();
boOutA.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
```

Intro to the algorithm in CMSSW

- CMS ECAL weights for Run-2 and beyond:
  - The multifit algorithm is an amplitude reconstruction algorithm modelling a pulse as a sum of in-time and out-of-time pulses
  - The core of the algorithm is the summation of amplitudes x weights over the pulse shape inside of a calorimeter crystal
  - This algorithm is run over the >60k crystals in the barrel of CMS ECAL







First iteration of the algorithm

To begin, I took the core of the calculation and translated that to run on the FPGA with Vitas HLS:

```
float sumA = 0;
float sumT = 0;

for(int i = 0; i < size; ++i)
  {
    if(in2a[i]==1) *outG=1;
    sumA = sumA + in0a[i] * in1a[i] * (float)in2a[i];
    sumT = sumT + in0t[i] * in1a[i] * (float)in2a[i];
  }
*outA = sumA;
*outT = sumT;
```

This algorithm is then run per crystal on the CMS ECAL barrel

Science and Technology Facilities Council

SWIFT HEP

First iteration timing

… SLOW!

Not measured, because it obviously slowed things to unusable levels. I estimate that the performance was down to the order of 1 event per second

Made the assumption that calling the FPGA to do a calculation 60k times per event might be the issue

The algorithm can be modified…

Algorithm updated

The updated version used arrays of all the numbers, to do all of the calculations at once, and return an array of results, the core becomes:

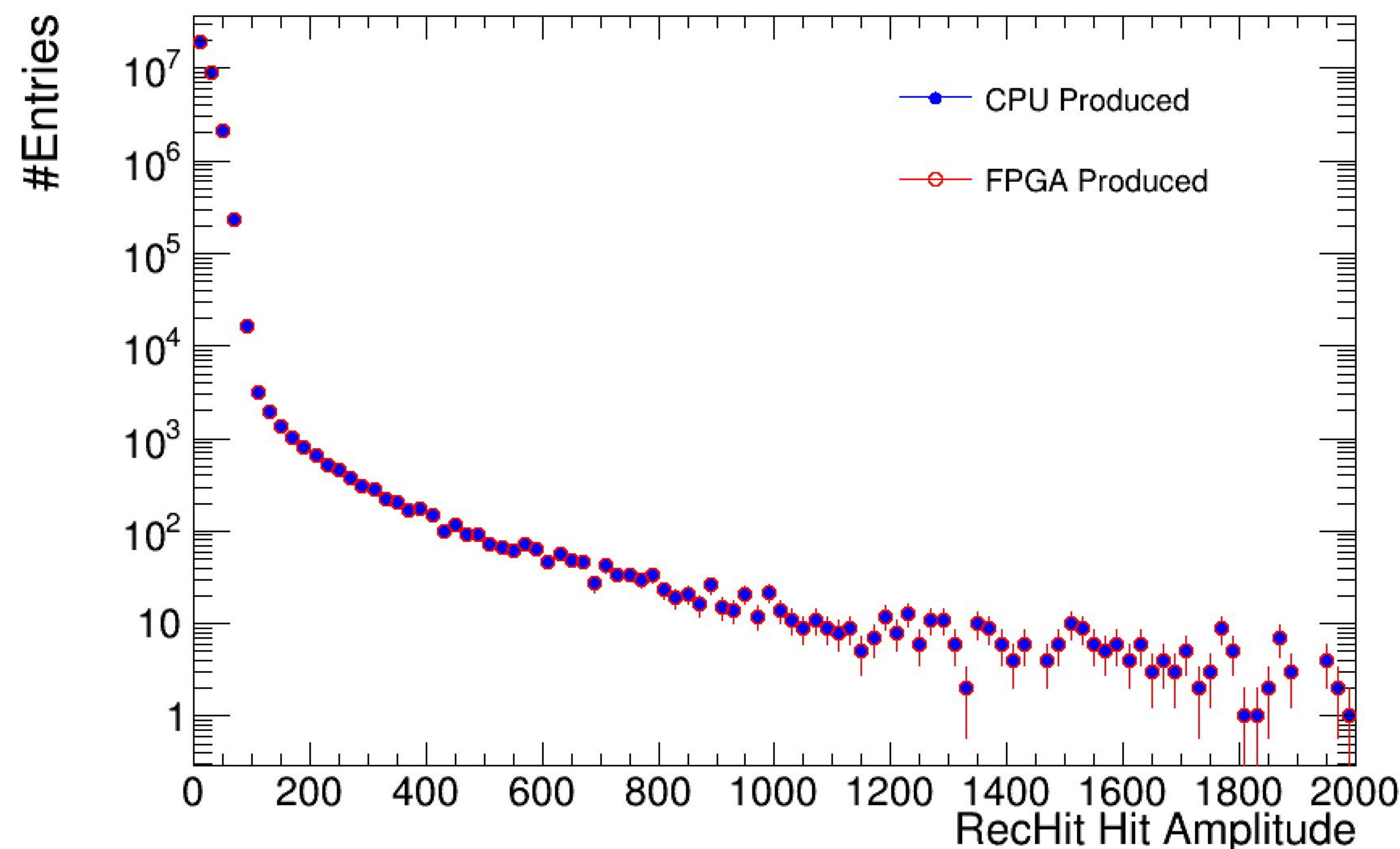
```
float sumA = 0;
float sumT = 0;

for(int j=0; j<N; ++j){
  sumA = 0;
  sumT = 0;

  for(int i = 0; i < size; ++i){
    if(*(&in2a[i]+j*size)==1) *(&outG[j])=1;
    sumA = sumA + (*(&in0a[i]) * (float)*(&in1a[i]+(j*size)) * (float)*(&in2a[i]+(j*size)));
    sumT = sumT + (*(&in0t[i]) * (float)*(&in1a[i]+(j*size)) * (float)*(&in2a[i]+(j*size)));
  }

  *(&outA[j]) = sumA;
  *(&outT[j]) = sumT;
```

Science and Technology Facilities Council

SWIFT HEP

Algorithm updated

- The updated version used arrays of all the numbers, enabling the calculation to be done over all of the crystals all at once.
- The amplitudes then can be trivially sent to the calculation for the reconstructed hits inside of the calorimeter
- Satisfying to compare the values calculated from CPU algorithm and FPGA algorithm →
- How about the timing? Well, a few things first…



Science and Technology Facilities Council

Benchmarking Setup

SWIFT HEP

reference ECAL RecHits Producer exists for CPU, GPU (heavy RAL involvement, alpaka version ready for 2024) and FPGA

- run each producer flavor + another module which exists to load the CPU so it other work to do
    - run with 8 threads, 8 events processed in parallel (“streams”)
    - GPU:A100, FPGA u250, CPU: Intel(R) Xeon(R) Gold 6242R CPU @ 3.10GHz
- look at individual module times to get a feel of the relative times

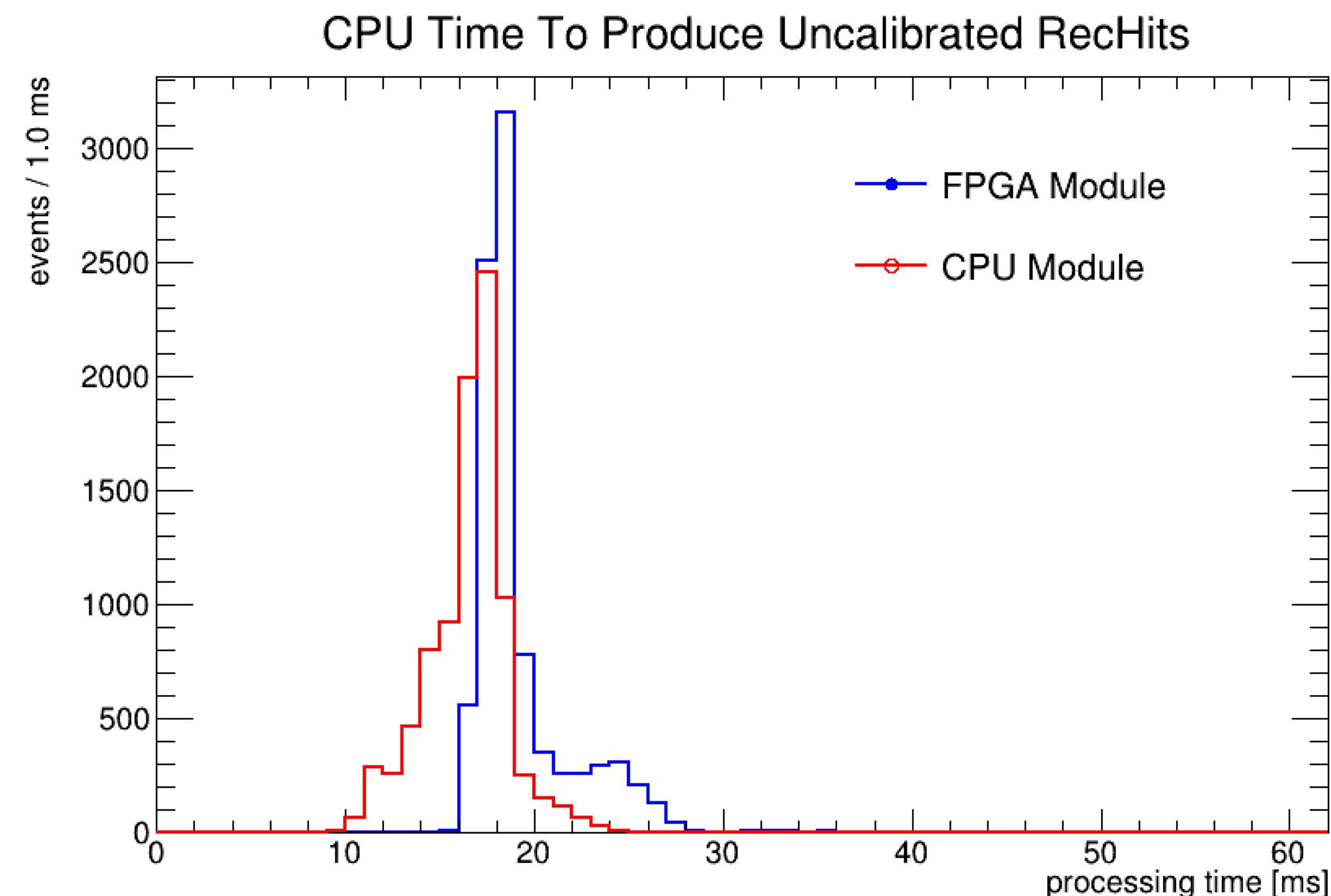
caveat: these numbers are preliminary and are indicative only, need to develop more realistic benchmark

Science and Technology Facilities Council

SWIFT HEP

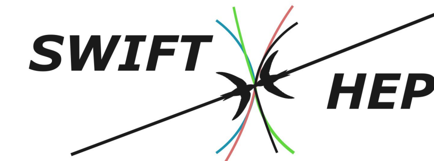
Timing Results

- currently slightly more CPU usage offloading to FPGA
  - offloading module highly unoptimised, expect this to significantly improve
  - the algorithm was chosen for its simplicity, more complicated ones may get more of a speed up
- key thing is that FPGA represents a large latency
  - trick will be ensuring that there is plenty of other work for the CPU to do while waiting for FPGA calculation to complete
  - need to figure out efficient ways of running this and characterise limitations

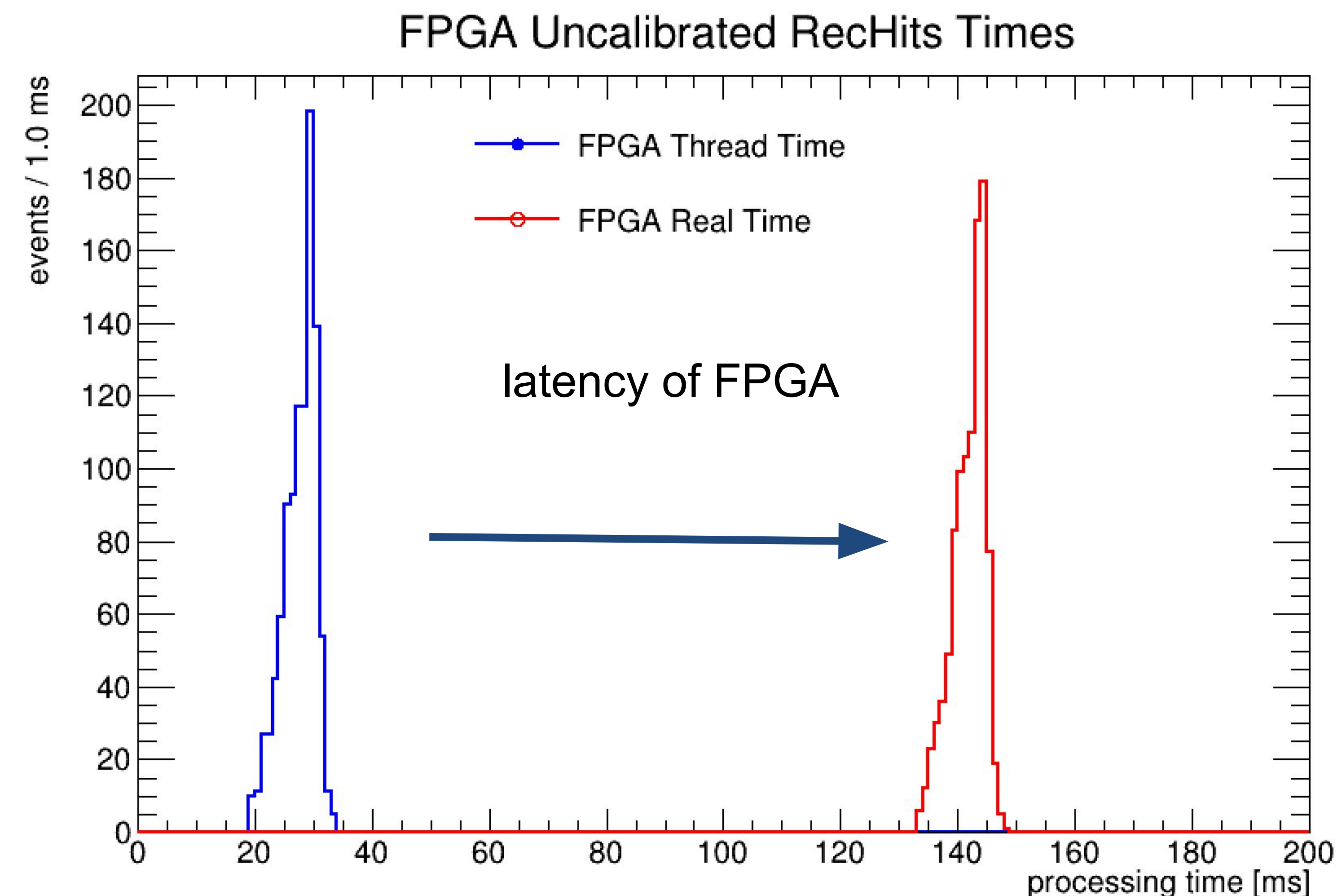


Science and Technology Facilities Council

Timing Results



- currently slightly more CPU usage offloading to FPGA
  - offloading module highly unoptimised, expect this to significantly improve
  - the algorithm was chosen for its simplicity, more complicated ones may get more of a speed up
- key thing is that FPGA represents a large latency
  - trick will be ensuring that there is plenty of other work for the CPU to do while waiting for FPGA calculation to complete
  - need to figure out efficient ways of running this and characterise limitations



Science and Technology Facilities Council

SWIFT HEP

Results

- Without further setup, running a single thread, running the FPGA takes approximately 5x the CPU alone

- running 8 threads, 8 streams in parallel writing the collection to disk throughput:
  - CPU: 5.7± 0.1 ev/s
  - FPGA: 5.7 ± 0.1 ev/s
  - GPU: 6.2 ± 0.2 ev/s

note: this was heavily IO bound, doing further tests indicated that FPGA code had a max throughput of 8.8ev/s

- highly dependent on keeping the CPU busy with other things while we wait for the FPGA to return the result
- work is ongoing to optimise and understand this

Future work

- The timing results are needs some careful investigation with multi-threading and offloading in a more realistic job – How to report events/s need careful thought!!
- The workflow took a while to develop across the cmssw software and the HLS algorithms, but now that this is in place, a more complicated and realistic benchmark can be tackled
  - Synergy and interest inside cms group for work on:
    - Tracking algorithms
    - ECAL algorithms
  - These more complicated algorithms have expertise that can be drawn from for comparison and performance
- Will continue to document the process of translating a c++ algorithm into something translatable for an FPGA

backup

Science and Technology Facilities Council

SWIFT HEP

First iteration (full code)

```
extern "C" {
        void vamp(
              const float *in0a, // Read-Only Vector 1
              const float *in1a, // Read-Only Vector 2
              const unsigned int *in2a,   // Read-Only Vector 3
              const float *in0t, // Read-Only Vector 1
              float *outA,       // Output Result
              float *outT,       // Output Result
              int *outG,         // Output Result
              int size          // Size in integer
              )
        {
#pragma HLS INTERFACE m_axi port=in0a bundle=aximm0
#pragma HLS INTERFACE m_axi port=in1a bundle=aximm1
#pragma HLS INTERFACE m_axi port=in2a bundle=aximm2
#pragma HLS INTERFACE m_axi port=in0t bundle=aximm2
#pragma HLS INTERFACE m_axi port=outA bundle=aximm3
#pragma HLS INTERFACE m_axi port=outT bundle=aximm3
#pragma HLS INTERFACE m_axi port=outG bundle=aximm3
          float sumA = 0;
          float sumT = 0;

          for(int i = 0; i < size; ++i)
            {
             if(in2a[i]==1) *outG=1;
             sumA = sumA + in0a[i] * in1a[i] * (float)in2a[i];
             sumT = sumT + in0t[i] * in1a[i] * (float)in2a[i];
            }
          *outA = sumA;
          *outT = sumT;
        }
}
```

Second iteration (full code)

```
extern "C" {
        void varr(
              const float *in0a, // Read-Only Vector 1
              const int *in1a,   // Read-Only Vector 2
              const int *in2a,   // Read-Only Vector 3
              const float *in0t, // Read-Only Vector 1
              float *outA,       // Output Result
              float *outT,       // Output Result
              int *outG,         // Output Result
              int N,             //N is number of arrays (aka, the number of digits or crystals)
              int size          ///size is the number of samples (size of the array in0a or in0t)
              )
        {
#pragma HLS INTERFACE m_axi port=in0a bundle=aximm0
#pragma HLS INTERFACE m_axi port=in1a bundle=aximm1
#pragma HLS INTERFACE m_axi port=in2a bundle=aximm2
#pragma HLS INTERFACE m_axi port=in0t bundle=aximm2
#pragma HLS INTERFACE m_axi port=outA bundle=aximm3
#pragma HLS INTERFACE m_axi port=outT bundle=aximm3
#pragma HLS INTERFACE m_axi port=outG bundle=aximm3

          float sumA = 0;
          float sumT = 0;
          for(int j=0; j<N; ++j){
            sumA = 0;
            sumT = 0;
            for(int i = 0; i < size; ++i){
              if(*(&in2a[i]+j*size)==1) *(&outG[j])=1;
              sumA = sumA + (*(&in0a[i]) * (float)*(&in1a[i]+(j*size)) * (float)*(&in2a[i]+(j*size)));
              sumT = sumT + (*(&in0t[i]) * (float)*(&in1a[i]+(j*size)) * (float)*(&in2a[i]+(j*size)));
            }
            *(&outA[j]) = sumA;
            *(&outT[j]) = sumT;
          }
        }
}
```