SLAC CPAD RDC5
HLS In A DAQ Environment
A Tale of 2 Experiments

SLAC TID & FPD

7 Nov 2023

Outline

- The Tools; HLS, C++ + Metaprogramming/Templates
    - How these facilitate the FPGA DAQ implementations
- nEXO and the HLS/FPGA compression code
- Mathusla and the HLS/FPGA triggering code
- Takeaways

The Tools: FPGAs, HLS & C++

- Why FPGAs
  - High throughput
  - Low Latency
  - Deterministic & Controllable throughput and latency

- Why HLS
  - Makes fairly complex processing at the ‘edge’ possible
    - Can do things that would be impractical in a hardware language like VHDL or Verilog

- Why C++
  - Familiar
  - Templates/Metaprogramming are a good fit with FPGA’s strengths

HLS - Brief Overview

- Many talks comparing performance of HLS to VHDL/Verilog generated code

- While important, misses other important points
    - HLS allows one to target problems that would be too complex for hardware languages
    - Fast turn-around times allow exploration of alternative ideas, *what-if'ing*
    - Simple not possible in hardware languages

- More about *can you do it at all* than *can it be written more performant*

HLS + C++ Templates/Metaprogramming - The Good

- C++ Templates/Metaprogramming play into the strengths of FPGAs
- Two advantages
  - Templates/Metaprogramming allows more to be done at compile time
    - The more that can be statically specified at compile rather than execution time, the better
  - The abstraction that these bring allows fairly generic code
- Combined, these make flexible and efficient frameworks possible
  - Can specify not only flexible array dimensions and loop sizes, but also
  - Even algorithmic selections based on compile time knowledge
  - Can query and use what the compiler knows
- But, isn't there always…

HLS + C++ Templates/Metaprogramming - The Bad

- While C++ syntax is familiar, but an FPGA is not a serial CPU
  - The compute model is very different, hence
    - Must code to the strengths of the FPGA, think like an FPGA
  - The algorithmic code and structure is very different
    - The code will look alien to a CPU programmer
- HLS compiler has a more difficult job because of a bigger playing field of parameters
  - Typical CPU compiler is concerned with optimizing performance and perhaps memory
  - HLS compiler must consider
    - Performance
    - Resource Usage of BRAMs/LUTs/FFs/DSPs and tradeoffs between these
    - Timing constraints & capabilities of a host of different FPGAs
- Tradeoffs are nuanced and complicated
  - Compiler cannot make all the decisions, needs some user hand-holding

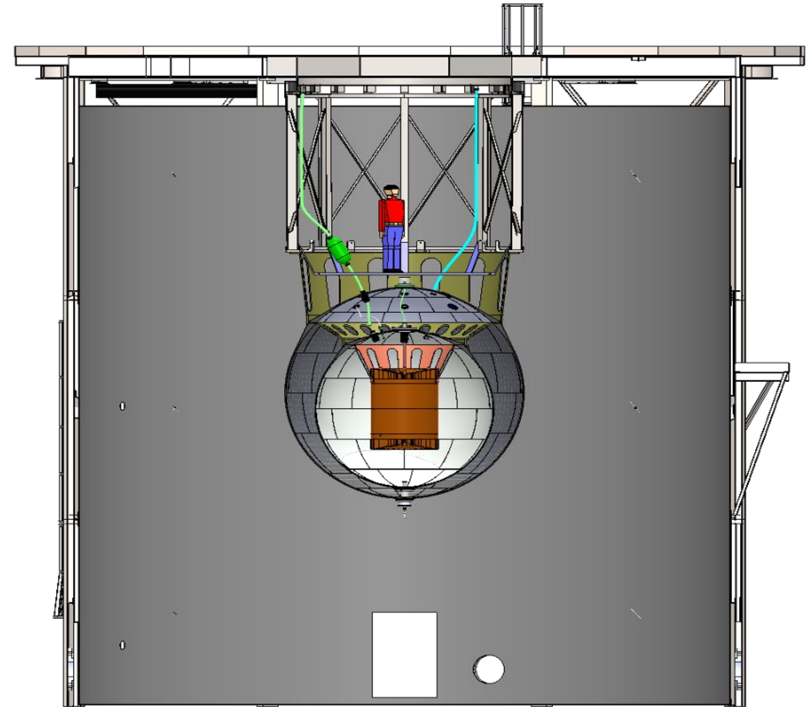
HLS + C++ Templates/Metaprogramming - The Ugly

- HLS compiler is not like the reliable & familiar gcc
  - Quirky and sometimes unruly
- The generated code can be fragile
  - A small change can result in large changes in resources/performance *etc.*
  - New releases can sometimes dramatically change the resultant code
- At times, feels like fighting the built-in optimizations
  - Sometimes the programmer does know best - case example is local vs global optimization
- The error reporting is often obscure to frustratingly misleading
  - Involved/intricate templates can produce error messages of 1000s of characters

nExo - A Neutrinoless Double Beta Decay Experiment

- More sensitive follow-on to EXO-200
- Will be installed at SNOLAB
- TPC with
  - 3840 Charge Channels
  - 7680 Photon Channels
  - Both digitizing 12 bits @ 2MHz

- Will concentrate on the charge channels



nEXO Data Compression

- The Challenge - Physics Data Rate vs Calibration Data Rate
  - Physics rate is low (~couple HZ), no problem
    - With drift times of ~1.5 msec, 12 Mbit ADCs @ 2MHz * 3840 Channels = ~140Mb/event
  - Calibration is continuous, 92Gb/sec → big problem

- Calibration data must be continuously streamed to a SSD for 2 hours
  - Requires a data volume reduction of 3.5 - 4.0 to meet bandwidth and reasonable storage sizes

- Solution: Compress the data
  - Possible since the data is mostly pedestals
  - Noise is low ⇒ Low entropy ⇒ Highly compressible

nExo Compression

- Compression is more than the compression factor, desirable features
  - High throughput
  - Capped & predictable worst case performance
  - Reasonable resource usage
  - Lossless
    - Easy to verify, no arguing about what might be lost
    - Enables offline removal of common mode noise (side benefit)

- nEXO will use Arithmetic Probability Encoding (APE) on FPGA *vs* EXO-200’s Huffman on CPU

nEXO Compression - APE vs Huffman

Huffman +'s / -'s

- \- Difficult to build encoding table
  - Non-FPGA friendly sort
- \+ Easy encoder
  - Simple lookup and bit-stuffer
- \- Inefficient/unstable at low entropy
  - Best when probabilities = powers of 2

APE +'s / -'s

- \+ Easy to build encoding table
  - Simple histogram of the distribution
- \- Encoder is non-trivial
  - Next slide
- \+ Achieves entropy limited compression
- \+ Takes advantage of what FPGAs do best
  - Wacky bit manipulation

nEXO FPGA APE Compression

- Initial code base was developed for another TPC based experiment
    - Using C++ metaprogramming/templates makes it easy to adapt to nEXO

- Two problems with the textbook implementations
    - Involves an integer division in the inner encoding loop
        - Solved by carefully rescaling the histogram to a binary power, division → shift

    - Encoding loop outputs one bit/FPGA clock cycle, neither predictable nor performant
        - Found a method to do the encoding in 1 symbol/FPGA clock cycle
        - Made possible by the ability to try lots of different ideas before found the solution

nEXO Compression - What is compressed?

- The differences of subsequent ADCs are compressed
  - Avoids knowing the pedestals
- Only the pedestals are compressed
  - Have picked to compress on those +/- 3 (a 3-bit value)
  - On EXO-200, almost all the non-signal differences were contained in this range
  - nEXO advertised to have similar noise, typically achieved 4-6 compression
- nEXO will combine successive differences to make a 6-bit value to be compressed
  - Slightly better compression since this is a joint probability and can remove any correlations
  - More important, now do 2 values/clock cycle
- Values outside this range are placed in a lookaside list
  - Number of bits is determined by the largest value in this list, i.e. if 80 is largest value, encode all in 7 bits
  - Because of the shaping times, differences of even these values are contained to smaller than 12 bits

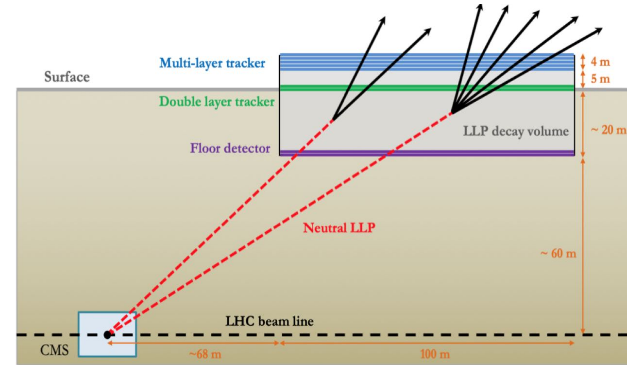
nEXO Compression - Macro Architecture

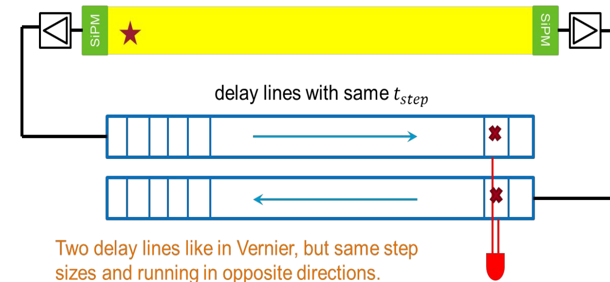
- Encoding is on a channel-by-channel basis
  - Prevents poorly performing channels from inflating the entropy distribution of other channels
  - Side Benefit: Allows some flexibility when decoding,
    - If only wish to look at selected channels
    - With random access to the channels, can decode channels in parallel

- Code is arranged in a number of independent compression engines each encoding a number of channels
  - Input rate of 2MHz and an FPGA clock of 200MHz, can encode 100 symbols/cycle = 200 ADCs/cycle
  - Picking 128 ADCs => 3840 channels/128 = 30 encoding engines
    - Gives a cushion to encode any lookaside list + other small non-deterministic but predictable worst cases
  - Engines all run in parallel
    - HLS makes configuring and realizing this fairly easy

Mathusla - Detector Description

- Goal is to detect an Long Lived Particle from CMS IP
- In a 100m x 100m pit
  - Pit is 60m above & 68 m downstream of CMS IP
- Populate by a 10 x 10 grid of 9m x 9m towers
  - Each tower has 6 trigger layers, 3x, 3y
  - Layers consist of 3.5 cm x (4x2.45) m bars
  - Readout is by SIPMs at each end
    - 1 dimension measured by bar number
    - 1 dimension measured by time difference of SIPMs
    - Hit = Coincidence with a window of twice the time length





Mathusla - Triggering

- Goal is to deliver a trigger to CMS if an LLP is detected
  - Trigger must be delivered within 9.5 *usec* ('size' of CMS front-end buffers and external trigger requirements)
  - Fixed latencies (travel time, data aggregation, transmission time back to CMS) leaves only 2.5 *usecs*

- Two pieces, track-finding and vertexing
  - Track-finding is independently done for each 3 x 3 group of towers, called a Trigger Unit
    - Involves 100 FPGAs in the central module of each Trigger Unit
    - Only upward going tracks are considered
      - Timing distinguishes upward-going from downward-going

  - Vertexing
    - Aggregates all the found tracks into a central FPGA to locate a potential decay vertex
    - Will not talk about this

Mathusla - Track Finding Basics

- Allocated 2.0 *usecs* for track finding
  - @ 200 MHz FPGA clock, means 400 cycles
  - Must take advantage of massive parallelism

- Explored many different approaches
  - Lookup tables
  - Hough transforms

- But all these involve either a memory bottleneck or something that kills massive parallelism
  - Settled on the ultra simple taking pairs of hits, projecting to other layers, looking for coincidences
  - By limiting the number of hits to 10 per layer, leads to a permutation loop of 100, which can be unrolled
  - The projections to the other 4 layers can also be done in parallel
  - All arithmetic is done with arbitrary precision integers (no floating point)
  - Coincidences are done by a simple AND into a bit mask representing the hits in the projected layer
  - There are 3 such engines (seed layers 0,4 1,5 2,3) to allow tracks with any 4/6 layers

Mathusla - Triggering, Results & Conclusions

- Only the first coincidence stage has been coded and synthesized, but still not tested
- This stage is taking ~250 cycles of the available 400 cycles
  - Based on experience (hope) the remaining steps (duplication elimination, track parameters, packaging) can be done in 150 cycles
- Resource usage is reasonable, leaving enough logic for the remaining steps
- Without HLS/C++ and templates/metaprogramming even this simple track-finding in an FPGA would have been very hard to impossible

Takeaways

- HLS is almost necessary to
  - Take advantage of the very large FPGAs now available …and…
  - The more complex computing projects these large FPGAs allow
- C++ and Templates/Metaprogramming are a good fit for FPGAs
  - The more that is known at compile-time the better

- HLS is not like coding for a CPU
  - Some due to it is still an evolving technology, it has gotten much better through the years
  - Some due to the inescapable complexity of an FPGA, it isn't a simple clocked CPU
- C++ + Templates/Metaprogramming has its own problems
  - Not an easy to master

- BOTTOM LINE: This takes patience, perseverance, and a thick-skin, either you are
  - An Optimist - You will be rewarded
  - A Fatalist - TINA (There Is No Alternative)