Traineeships in Advanced Computing for High Energy Physics (TAC-HEP)

FPGA module training

Lecture-12: April 21st 2026

Varun Sharma

University of Wisconsin – Madison, USA

Content

So far

- Motivation & Introduction
- Comparison: FPGAs/ASICs/GPU/CPU
- Domain specific Accelerators
- HLS setup and first example project
- Unsupported C/C++ constructs
- Data types
- HLS Pragmas
  - Interface
  - Array partition
  - Array Reshape
  - Pipeline
  - DataFlow
  - Allocation
  - Latency
  - Inline
  - Stable
  - Loop Unrolling
  - Loop_flatten
  - Loop_merge
  - Loop_tripcount

Today

- HLS Pragmas
  - Aggregate
  - Expression_balance
  - Protocol
  - bind_op
  - bind_storage
  - Function_instantiate

Important Dates

- Last class would be May 5th
- Project and homework submission before May 10th

TAC-HEP 2026

Pragma HLS Aggregate

https://docs.amd.com/r/en-US/ug1399-vitis-hls/pragma-HLS-aggregate

Pragma HLS AGGREGATE

Collects and groups struct data fields into a single wide scalar (wider word width)

Uses the **AGGREGATE pragma** to enable simultaneous read/write of all struct members

Bit alignment is based on the order of struct elements:

- First element aligns with the LSB.
- Last element aligns with the MSB.

Arrays within the struct are:

- Fully partitioned
- Reshaped similarly to ARRAY_RESHAPE
- Packed into the wide scalar along with other elements

Pragma HLS AGGREGATE: Syntax

#pragma HLS AGGREGATE variable=<var> compact=<arg>

- **variable = <variable>:** Specifies the variable to be grouped
- **compact = [bit | byte | none | auto]:** Specifies the alignment of the aggregated struct
  - Alignment can be on the bit-level, the byte-level, none, or automatically determined by the tool which is the default behavior

Caution: Careful while using AGGREGATE optimization on struct objects with large arrays. If an array has 4096 elements of type int, this will result in a vector (and port) of width 4096×32=131072 bits. The Vitis HLS tool can create this RTL design, however it is very unlikely logic synthesis will be able to route this during the FPGA implementation.

Pragma HLS AGGREGATE: Usefulness

- **Without aggregation, each struct field becomes a separate port**
  - Inefficient for memory & interfaces

- **Advantages**
  - Reduces interface overhead
  - Improves memory burst efficiency
  - Cleaner AXI transactions

- **Some use cases:**
  - Passing packets, pixels, or feature vectors
  - High-throughput AXI memory transfers

HLS AGGREGATE: Example-1

```
#include <ap_int.h>

typedef struct {
    ap_uint<8>  a;
    ap_uint<16> b;
    ap_uint<32> c;
} my_struct;

void process(my_struct in[4], my_struct out[4]) {
#pragma HLS AGGREGATE variable=in compact=bit
#pragma HLS AGGREGATE variable=out compact=bit
#pragma HLS PIPELINE

    for (int i = 0; i < 4; i++) {
        out[i].a = in[i].a + 1;
        out[i].b = in[i].b + 2;
        out[i].c = in[i].c + 3;
    }
}
```

HLS combines all fields into a **single wide port**, improving efficiency on AXI streams or memory-mapped interfaces

HLS AGGREGATE: Example-1

```
#include <ap_int.h>

typedef struct {
    ap_uint<8>  a;
    ap_uint<16> b;
    ap_uint<32> c;
} my_struct;

void process(my_struct in[4], my_struct out[4]) {
#pragma HLS AGGREGATE variable=in compact=bit
#pragma HLS AGGREGATE variable=out compact=bit
#pragma HLS PIPELINE

    for (int i = 0; i < 4; i++) {
        out[i].a = in[i].a + 1;
        out[i].b = in[i].b + 2;
        out[i].c = in[i].c + 3;
    }
}
```

HLS combines all fields into a **single wide port**, improving efficiency on AXI streams or memory-mapped interfaces

Without AGGREGATE, the struct might use separate interface ports.

HLS AGGREGATE: Example-2

```
#include <hls_stream.h>
#include <ap_axi_sdata.h>

typedef struct {
   ap_uint<8>  id;
   ap_uint<16> val;
} packet_t;

typedef ap_axiu<32, 0, 0, 0> axis_t;

void pack_stream(hls::stream<packet_t> &in, hls::stream<axis_t> &out) {
#pragma HLS AGGREGATE variable=in compact=bit
#pragma HLS INTERFACE axis port=in
#pragma HLS INTERFACE axis port=out
#pragma HLS PIPELINE

   packet_t pkt = in.read();
   axis_t out_pkt;
   out_pkt.data = (pkt.val, pkt.id);  // concat into 24 bits
   out_pkt.last = 0;
   out.write(out_pkt);
}
```

```
typedef struct{
unsigned char R, G, B;
} pixel;

pixel AB[17];
#pragma HLS aggregate variable=AB
```

TAC-HEP 2026

Pragma HLS Expression_balance

https://docs.amd.com/r/en-US/ug1399-vitis-hls/pragma-HLS-expression_balance

Pragma HLS Expression_balance

- Sequential C/C++ operations can form long RTL operation chains, increasing latency with small clock periods.
- Vitis HLS rearranges operations using associative and commutative properties to reduce latency
- This rearrangement forms a balanced tree of operations, known as **expression balancing.** Defaults:
  - **Enabled for integer operations**
  - **Disabled for floating point operation**

#pragma HLS Expression_balnce

Expression_balance: Example

```
void top_function(int A[SIZE], int B[SIZE], int C[SIZE]) {
#pragma HLS DATAFLOW
#pragma HLS expression_balance

    hls::stream<int> a_s, b_s, sum_s;

    load(A, B, a_s, b_s);
    add(a_s, b_s, sum_s);
    multiply_by_two(sum_s, C);
```

Helps the compiler build a tree instead of a long chain

- Balanced -> faster (low latency)
- Chained -> few resources

Can be used in DSP heavy pipelines

TAC-HEP 2026

Pragma HLS Performance

https://docs.amd.com/r/en-US/ug1399-vitis-hls/pragma-HLS-performance

Pragma HLS Performance

- The PERFORMANCE pragma in Vitis HLS provides a mechanism for defining high-level performance goals for your design, guiding the synthesis tool's optimization efforts.
  - It can be applied at the **top level** of the design (Kernel/IP) and at **specific loop** levels within the design.
- ***Top-Level PERFORMANCE pragma***
  - Applied at kernel/top-level function to define **overall performance goal (throughput)**
  - Triggers **design-wide performance analysis**
  - HLS tool evaluates feasibility of meeting the target
  - Automatically infers **loop-level optimizations** across the design
- ***Loop-Level PERFORMANCE pragma***
  - Applied to **specific loops or loop nests**
  - Sets performance goal for that section using **target_ti**
  - Tool derives lower-level directives like: UNROLL, PIPELINE, ARRAY_PARTITION, and INLINE

Pragma HLS Performance

It uses parameters **target_ti** to guide optimization toward specific performance goals.

The directive does **not guarantee** the specified performance — it's only a goal.

target_ti: target transaction interval - number of clock cycles for the function, loop or region of code to complete an iteration
target_tl: target latency **-** number of clock cycles for the loop to complete all iterations

#pragma HLS performance target_ti=<value> target_tl=<value> unit=[sec | cycle]

HLS Performance: Example

```
for (int i =0; i < 1000; ++i) {

#pragma HLS performance target_ti=1000

  for (int j = 0; j < 8; ++j) {
    int tmp = b_buf[j].read();
    b[i * 8 + j] = tmp + 2;
  }
}
```

```
1 const int T = 100;
2 L1: for (int i=0; i<N; i++)
3 L2: for (int j=0; j<M; j++)
4 { pragma HLS performance target_ti=T   ...   }
```

TAC-HEP 2026

Pragma HLS Protocol

https://docs.amd.com/r/en-US/ug1399-vitis-hls/pragma-HLS-protocol

Pragma HLS Protocol

- Defines a **protocol region** where **no clock operations** are inserted by Vitis HLS unless explicitly coded
- Ensures no clocks between operations, including reads/writes to function arguments
- Maintains the exact order of reads and writes in the synthesized RTL
- A protocol region is defined using **braces {}** and a name
- Ensures that **reads and writes happen in the exact order** written in the C/C++ code

#pragma HLS protocol [floating | fixed]

Pragma HLS Protocol

#pragma HLS protocol [floating | fixed]

Floating: Lets code statements outside the protocol region overlap and execute in parallel with statements in the protocol region in the final RTL. The protocol region remains cycle accurate, but outside operations can occur at the same time. This is the default mode

Fixed: The fixed mode ensures that statements outside the protocol region do not execute in parallel with the protocol region.

Protocol: Example

```
#include <hls_stream.h>
#include <ap_int.h>

void stream_modifier(hls::stream<ap_uint<32>> &in_stream,
                     hls::stream<ap_uint<32>> &out_stream) {
    #pragma HLS interface axis port=in_stream
    #pragma HLS interface axis port=out_stream
    #pragma HLS interface ap_ctrl_none port=return

    {
        #pragma HLS protocol fixed  // ensures tight control over read-modify-write

        ap_uint<32> data_in = in_stream.read();  // AXI handshaking happens here
        ap_uint<32> data_out = data_in + 1;
        out_stream.write(data_out);              // AXI handshaking happens here
    }
}
```

Protocol: when to use

- **Cycle-accurate control** over I/O or memory accesses
  - Allows to define exact timing and handshaking signals for function interface – crucial when interacting with external hardware or memory interfaces with strict timing requirement
- Custom protocols (e.g., GPIO, SPI-like behavior)
  - Interfacing with custom hardware using protocols not directly supported by standard HLS interfaces
  - Define sequence of reads, writes and control signal transitions
- **Low-latency pipelines** with precise read/write sequencing
- To override default behavior where HLS inserts waits or splits across cycles

TAC-HEP 2026

Pragma HLS bind_op

https://docs.amd.com/r/en-US/ug1399-vitis-hls/pragma-HLS-bind_op

HLS bind_op

- Vitis HLS assigns specific operations (*e.g., mul, add, div)* to specific device resources for RTL implementation
- The BIND_OP pragma specifies that for a specific variable, an operation (mul, add, div) should be mapped to a specific device resource for implementation (impl) in the RTL.
- If BIND_OP is **not** used, Vitis HLS automatically selects appropriate resources for operations.
- BIND_OP allows setting **latency** for operations, but only if the operation supports **multi-stage implementations**.
- Multi-stage implementations are provided for:
  - Basic arithmetic: add, subtract, multiply, divide
  - All floating-point operations

HLS bind_op: Syntax

#pragma HLS bind_op variable=<variable> op=<type> impt=<value> latency=<int>

Variable: Defines the variable to assign the BIND_OP pragma. The variable in this case is one that is assigned the **result** of the operation that is the target of this pragma

Op=<type>:

- Defines the operation to bind to a specific implementation resource.
- Supported functional operations include: *mul, add, and sub*
- Supported floating poin operations include: *fadd, fsub, fdiv, fexp, flog, fmul, frsqrt, frecip, fsqrt, dadd, dsub, ddiv, dexp, dlog, dmul, drsqrt, drecip, dsqrt, hadd, hsub, hdiv, hmul, and hsqrt*

Impl=<value>: Defines the implementation to use for the specified operation.
Supported *fabric* and *dsp*

Latency=<int>: Defines the default latency for the implementation of the operation.

- Default: -1 (allows Vitis HLS choose the latency)

HLS bind_op: Example

```
#include <ap_int.h>

void pipelined_mul(ap_int<16> a, ap_int<16> b, ap_int<32> &result) {
   ap_int<32> tmp;

#pragma HLS bind_op variable=tmp op=mul impl=fabric latency=3

   tmp = a * b;
   result = tmp;
}
```

```
int multiply(int a, int b) {
#pragma HLS bind_op op=mul impl=DSP

   return a * b;
}
```

Operation	Implementation	Min Latency	Max Latency
add	fabric	0	4
add	dsp	0	4
mul	fabric	0	4
mul	dsp	0	4
sub	fabric	0	4
sub	dsp	0	0

TAC-HEP 2026

Pragma HLS bind_storage

https://docs.amd.com/r/en-US/ug1399-vitis-hls/pragma-HLS-bind_storage

HLS bind_storage

- Used to specify the type of hardware resource that should be used to implement a storage element (like an array or a variable)
  - Assigns a **variable (array or function argument)** to a specific **memory type** in RTL
  - Else Vitis HLS automatically selects the memory type and implementation
- Helps in controlling memory mapping such as:
  - Choosing between single-port or dual-port RAM
  - Can specify the **memory implementation** (e.g., block RAM, LUTRAM, fabric)
- The **latency option** allows:
  - Modeling of **off-chip or non-standard SRAMs** (e.g., latency of 2–3)
  - Adding **pipeline stages** internally to help **meet timing** in RTL synthesis
- Allows you to have fine-grained control over how memory structures in your C/C++ code are translated into hardware.

HLS bind_storage: Syntax

#pragma HLS bind_storage variable=<variable> type=<type> impt=<value> latency=<int>

variable=<variable_name>: Specifies the name of the array or variable in your C/C++ code that you want to bind to a specific storage resource

type=<storage_type>: Defines the interface type for the storage

- Supported: *fifo, ram_1p, ram_1wnr, ram_2p, ram_s2p, ram_t2p, rom_1p, rom_2p, rom_np*

impl=<implementation_type>: Specifies the physical implementation of the storage

- Supported: *bram, bram_ecc, lutram, uram, uram_ecc, srl, memory, and auto*

latency=<int>: Defines the default latency for the binding of the type

HLS bind_storage: Syntax

Supported storage

Type	Description
FIFO	A FIFO. Vitis HLS determines how to implement this in the RTL, unless the `-impl` option is specified.
RAM_1P	A single-port RAM. Vitis HLS determines how to implement this in the RTL, unless the `-impl` option is specified.
RAM_1WNR	A RAM with 1 write port and N read ports, using N banks internally.
RAM_2P	A dual-port RAM that allows read operations on one port and both read and write operations on the other port.
RAM_S2P	A dual-port RAM that allows read operations on one port and write operations on the other port.
RAM_T2P	A true dual-port RAM with support for both read and write on both ports.
ROM_1P	A single-port ROM. Vitis HLS determines how to implement this in the RTL, unless the `-impl` option is specified.
ROM_2P	A dual-port ROM.
ROM_NP	A multi-port ROM.

Supported Implementation

Name	Description
MEMORY	Generic memory lets the Vivado tool choose the implementation.
URAM	UltraRAM resource
URAM_ECC	UltraRAM with ECC
SRL	Shift Register Logic resource
LUTRAM	Distributed RAM resource
BRAM	Block RAM resource
BRAM_ECC	Block RAM with ECC
AUTO	Vitis HLS automatically determine the implementation of the variable.

https://docs.amd.com/r/2024.2-English/ug1399-vitis-hls/pragma-HLS-bind_storage

HLS bind_storage: Example

```
#include <iostream>
#include <vector>

void process_data(int input_data[10], int output_data[10]) {
#pragma HLS bind_storage variable=input_data type=RAM_2P impl=BRAM
#pragma HLS bind_storage variable=output_data type=RAM_1P impl=LUTRAM

   for (int i = 0; i < 10; ++i) {
      output_data[i] = input_data[i] * 2;
   }
}

int main() {
   int in[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   int out[10];
   process_data(in, out);
   for (int i = 0; i < 10; ++i) {
      std::cout << out[i] << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

TAC-HEP 2026

Pragma HLS function_instantiate

https://docs.amd.com/r/en-US/ug1399-vitis-hls/pragma-HLS-function_instantiate

HLS function_instantiate

FUNCTION_INSTANTIATE pragma is an optimization technique that has the area benefits of maintaining the function hierarchy but provides an additional powerful option:

- Performing targeted local optimizations on specific instances of a function
- This can simplify the control logic around the function call and potentially improve latency and throughput

- Allows each instance of a function to have its own unique RTL implementation
- Enables local optimizations per instance, especially when some inputs are constant, leading to simpler control logic, improved latency, and potentially better throughput.

HLS function_instantiate: Syntax

#pragma HLS function_instantiate variable=<variable>

variable=<variable> A required argument that defines the function argument to use as a constant

```
void swInt(unsigned int *readRefPacked, short *maxr, short *maxc, short *maxv){
#pragma HLS FUNCTION_INSTANTIATE variable=maxv
   uint2_t d2bit[MAXCOL];
   uint2_t q2bit[MAXROW];
#pragma HLS array partition variable=d2bit,q2bit cyclic factor=FACTOR

   intTo2bit<MAXCOL/16>((readRefPacked + MAXROW/16), d2bit);
   intTo2bit<MAXROW/16>(readRefPacked, q2bit);
   sw(d2bit, q2bit, maxr, maxc, maxv);
}
```

HLS function_instantiate: Example

```
char func_sub(char inval, char incr) {
#pragma HLS INLINE OFF
#pragma HLS FUNCTION_INSTANTIATE variable=incr
 return inval + incr;
}
void func(char inval1, char inval2, char inval3,
 char *outval1, char *outval2, char * outval3)
{
 *outval1 = func_sub(inval1, 1);
 *outval2 = func_sub(inval2, 2);
 *outval3 = func_sub(inval3, 3);
}
```

TAC-HEP 2026

Questions?

TAC-HEP 2026

Extra Slides

TAC-HEP 2026

HLS Setup on cmstrigger02

https://github.com/varuns23/TAC-HEP-FPGA/blob/main/hls-setup/readme.md

Setup

Setup Port Forwarding for RDP

- From your local machine, run:
  - `ssh -L 3389:localhost:3389 -J <username>@login.hep.wisc.edu <username>@cmstrigger02.hep.wisc.edu`
- Keep this terminal open - it maintains the RDP tunnel

Connect Using Remote Desktop (RDP Client)

- Use Microsoft's RDP client (called `Windows App`) available for macOS and Windows.
- **Setup:** Download & install the `Windows App`, Open the app and click the **+** icon, then select `Add PC`
- **Configure:** Enter the IP address: `localhost:3389` or `127.0.0.1:3389`
- **Connect:** Double-click the PC icon, enter your UW computing and , and click **Connect**
- You should now see the remote desktop of `cmstrigger02`

https://github.com/varuns23/TAC-HEP-FPGA/blob/main/hls-setup/readme.md

Setup workign Directory

After logging in via RDP, open a terminal inside the remote desktop

- `mkdir -p /scratch/`whoami`` (if not there already)
- `cd /scratch/`whoami``
  - **For Vitis HLS:**
    - `Source /opt/Xilinx/Vitis/2020.1/settings64.sh`
    - `cd /scratch/`whoami``
    - `vitis_hls`
  - **For Vivado HLS:**
    - `Source /opt/Xilinx/Vivado/2020.1/settings64.sh`
    - `cd /scratch/`whoami``
    - `vivado_hls`

Jargons

- **ICs - Integrated chip:** assembly of hundreds of millions of transistors on a minor chip
- **PCB:** Printed Circuit Board
- **LUT - Look Up Table aka 'logic'** - generic functions on small bitwidth inputs. Combine many to build the algorithm
- **FF - Flip Flops** - control the flow of data with the clock pulse. Used to build the pipeline and achieve high throughput
- **DSP - Digital Signal Processor** - performs multiplication and other arithmetic in the FPGA
- **BRAM - Block RAM** - hardened RAM resource. More efficient memories than using LUTs for more than a few elements
- **PCIe or PCI-E - Peripheral Component Interconnect Express**: is a serial expansion bus standard for connecting a computer to one or more peripheral devices
- **InfiniBand** is a computer networking communications standard used in high-performance computing that features very high throughput and very low latency
- **HLS** - High Level Synthesis - compiler for C, C++, SystemC into FPGA IP cores
- **HDL** - Hardware Description Language - low level language for describing circuits
- **RTL** - Register Transfer Level - the very low level description of the function and connection of logic gates
- **FIFO** – First In First Out memory
- **Latency** - time between starting processing and receiving the result
  - Measured in clock cycles or seconds
- **II - Initiation Interval** - time from accepting first input to accepting next input

Terminology

- **HLS file:** C/C++ code that will be synthesised and run on FPGA
- **Test bench (TB) file:** C/C++ code that is run to test the HLS code. It calls the HLS functions and can run tests on their output, e.g. C asserts
- **Tcl scripts:** set of tcl instructions executed by the Vivado HLS shell

- **Synthesis:** C/C++ ➞ HDL lang (VHDL/Verilog)
- **Project:** Collection of HLS and test bench (TB) files
  - Has a top-level function name that is the starting point for synthesis
- **Solution:** specific implementation of project
  - Runs on a specific device at a specific clock frequency
- **C simulation:** HLS+TB files are compiled with gcc against HLS headers and lib and plainly run as any other executable
- **C/RTL cosimulation:** synthesized HLS code is run on simulator and results tested on the C/C++ test bench