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

FPGA module training

Lecture-9: March 31st 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

Today

- HLS Pragmas
  - Latency
  - Inline
  - Stable
  - UNROLL

HLS Pragmas

"Pragmas": Instructions to tell your compiler how to build the hardware

- HLS tool provides different set of pragmas that can be used to optimize the design, reduce latency, improve performance etc. These pragmas can be directly added to the source code for the kernel.

Type	Attributes
Kernel Optimization	• pragma HLS aggregate • pragma HLS alias • pragma HLS disaggregate • pragma HLS expression_balance • pragma HLS latency • pragma HLS performance • pragma HLS protocol • pragma HLS reset • pragma HLS top • pragma HLS stable
Function Inlining	• pragma HLS inline
Interface Synthesis	• pragma HLS interface • pragma HLS stream
Task-level Pipeline	• pragma HLS dataflow • pragma HLS stream
Pipeline	• pragma HLS pipeline • pragma HLS occurrence
Loop Unrolling	• pragma HLS unroll • pragma HLS dependence
Loop Optimization	• pragma HLS loop_flatten • pragma HLS loop_merge • pragma HLS loop_tripcount
Array Optimization	• pragma HLS array_partition • pragma HLS array_reshape
Structure Packing	• pragma HLS aggregate • pragma HLS dataflow
Resource Utilization	• pragma HLS allocation • pragma HLS bind_op • pragma HLS bind_storage • pragma HLS function_instantiate

https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/HLS-Pragmas

TAC-HEP 2026

#pragma HLS Latency

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

Pragma HLS Latency

#pragma HLS latency min=<int> max=<int>

An important performance-guiding pragma: Constrain how many clock cycles a function/loop/region should take

- Specifies a minimum or maximum latency value, or both, for the completion of functions, loops, and regions
  - *min=<int>*: minimum latency for the function, loop, or region of code
  - *max=<int>*: maximum latency for the function, loop, or region of code

- **Latency:** # of CLK cycles required to produce an output

- **Function latency:** # of CLK cycles required for the function to compute all output values and return

- **Loop latency**: # of CLK cycles to execute all iterations of the loop

Pragma HLS Latency

#pragma HLS latency min=<int> max=<int>

- HLS always tries to minimize latency in the design

- When LATENCY pragma is specified
  - ***Min < Latency < Max***: Constraint is satisfied, No further optimization
  - ***Latency < min***: It extends latency to the specified value, potentially increasing sharing
  - ***Latency > max***: Increases effort to achieve the constraints
    - Still unsuccessful: issue a warning & produce design with the smallest achievable latency in excess of maximum

Pragma HLS Latency

- It is not a direction hardware structure pragma like Pipeline/Partition
- Instead it is a constrain that guides:
  - Scheduling
  - Operator sharing
  - Loop transformation
  - Function expansion
  - …

Pragma HLS Latency - Example

#pragma HLS latency min=<int> max=<int>

Example-1: Function foo is specified to have a minimum latency of 4 and a maximum latency of 8

```
int foo(char x, char a, char b, char c) {
  #pragma HLS latency min=4 max=8
  char y;
  y = x*a+b+c;
  return y
}
```

Example-2: loop_1 is specified to have a maximum latency of 12

```
void foo (num_samples, ...) {
  int i;
  ...
  loop_1: for(i=0;i< num_samples;i++) {
  #pragma HLS latency max=12
    ...
    result = a + b;
  }
}
```

Example-3: Creates a code region and groups signals that need to change in the same clock cycle by specifying zero latency

```
// create a region { } with a latency = 0
{
  #pragma HLS LATENCY max=0 min=0
  *data = 0xFF;
  *data_vld = 1;
}
```

Pragma HLS Latency - Example

```
void example (
  unsigned int in[N],
  short a,
  short b,
  unsigned int c,
  unsigned int out[N]
  ) {

   unsigned int x, y;
   unsigned int tmp1, tmp2, tmp3;

for_Loop: for (unsigned int i=0 ; i < N; i++) {
#pragma HLS latency min=4
        x = in[i];
        tmp1 = func(1, 2);
        tmp2 = func(2, 3);
        tmp3 = func(1, 4);

        y = a*x + b + squared(c) + tmp1 + tmp2 + tmp3;

        out[i] = y;
    }
}
```

Reminder: It is only a constraint, not a guarantee

- If impossible, HLS relaxes the constraint and gives the **best possible result**.

Pragma HLS Latency - Results

```
Timing:
  * Summary:
  +--------+----------+----------+------------+
  |  Clock |  Target  | Estimated| Uncertainty|
  +--------+----------+----------+------------+
  |ap_clk  | 25.00 ns | 7.401 ns |   3.12 ns  |
  +--------+----------+----------+------------+

Latency:
  * Summary:
  +---------+---------+----------+----------+-----+-----+---------+
  |  Latency (cycles) |  Latency (absolute) |  Interval | Pipeline|
  |   min   |   max   |    min   |    max   | min | max |   Type  |
  +---------+---------+----------+----------+-----+-----+---------+
  |      301|      301| 7.525 us | 7.525 us |  301|  301|   none  |
  +---------+---------+----------+----------+-----+-----+---------+

  + Detail:
      * Instance:
      N/A

      * Loop:
      +-----------+---------+---------+----------+-----------+-----------+-----+----------+
      |           |  Latency (cycles) | Iteration|  Initiation Interval  | Trip |          |
      | Loop Name |   min   |   max   |  Latency |  achieved |   target  | Count| Pipelined|
      +-----------+---------+---------+----------+-----------+-----------+-----+----------+
      |- for_Loop |      300|      300|         5|          -|          -|   60|    no    |
      +-----------+---------+---------+----------+-----------+-----------+-----+----------+
```

```
================================================================
= Utilization Estimates
================================================================
 Summary:
---------------------+---------+-------+---------+---------+-----+
        Name         | BRAM_18K| DSP48E|    FF   |   LUT   | URAM|
---------------------+---------+-------+---------+---------+-----+
DSP                  |        -|      -|        -|        -|    -|
Expression           |        -|      5|        0|      154|    -|
FIFO                 |        -|      -|        -|        -|    -|
Instance             |        -|      -|        -|        -|    -|
Memory               |        -|      -|        -|        -|    -|
Multiplexer          |        -|      -|        -|       47|    -|
Register             |        -|      -|      120|        -|    -|
---------------------+---------+-------+---------+---------+-----+
Total                |        0|      5|      120|      201|    0|
---------------------+---------+-------+---------+---------+-----+
Available SLR        |     1440|   2280|   788160|   394080|  320|
---------------------+---------+-------+---------+---------+-----+
Utilization SLR (%)  |        0|     ~0|       ~0|       ~0|    0|
---------------------+---------+-------+---------+---------+-----+
Available            |     4320|   6840|  2364480|  1182240|  960|
---------------------+---------+-------+---------+---------+-----+
Utilization (%)      |        0|     ~0|       ~0|       ~0|    0|
---------------------+---------+-------+---------+---------+-----+
```

TAC-HEP 2026

#pragma HLS INLINE

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

Pragma HLS Inline

#pragma HLS inline <recursive | off>

- Removes a function as a separate entity in the hierarchy
- The function is dissolved into the calling function and no longer appears as a separate level of hierarchy in RTL design
- May improve area by allowing the components within the function to be better shared or optimized with the logic in the calling function

- Improves performance by exposing more parallelism
- Increase area as it duplicates logic
- Reduce latency (fewer control steps)

- Top()
  - Adder()
  - Multiplier()

- Top() -> Adder() -> Multiplier()

Pragma HLS Inline

#pragma HLS inline <recursive | off>

- Removes a function as a separate entity in the hierarchy
- The function is dissolved into the calling function and no longer appears as a separate level of hierarchy in RTL design
- May improve area by allowing the components within the function to be better shared or optimized with the logic in the calling function

- ***INLINE***
  - *Without arguments,* the function it is specified in should be inlined upward into any calling functions
- ***Recursive***: Inlines all functions recursively within the specified function or region
  - By default, only one level of function inlining is performed
- ***Off:*** Disables function inlining to prevent specified functions from being inlined
  - For example, HLS automatically inlines small functions & with the off option, automatic inlining can be prevented

Pragma HLS Inline - Example

#pragma HLS inline <recursive | off>

- Inlines all functions within the body of *foo_top*
- Inlining recursively down through the function hierarchy, except function foo_sub is not inlined.
- The recursive pragma is placed in function *foo_top*
- The pragma to disable inlining is placed in the function foo_sub

```
foo_sub (p, q) {
#pragma HLS inline off
  int q1 = q + 10;
  foo(p1,q);// foo_3
  ...
}
void foo_top { a, b, c, d} {
#pragma HLS inline region recursive
  ...
  foo(a,b);//foo_1
  foo(a,c);//foo_2
  foo_sub(a,d);
  ...
}
```

Pragma HLS Inline

#pragma HLS inline <recursive | off>

```
#include "example.h"

void example ( unsigned int in[N], short a, short b, unsigned int c, unsigned int out[N]) {
  unsigned int x, y;
  unsigned int tmp1, tmp2, tmp3;

for_Loop: for (unsigned int i=0 ; i < N; i++) {
                    x = in[i];
                    tmp1 = func(1, 2);
                    tmp2 = func(2, 3);
                    tmp3 = func(1, 4);

                    y = a*x + b + squared(c) + tmp1 + tmp2 + tmp3;

                    out[i] = y;
}}

unsigned int squared(unsigned int a){
#pragma HLS INLINE OFF
  unsigned int res = 0;
  res = a*a;
  return res;
}

unsigned int func(short a, short b){
#pragma HLS INLINE OFF
  unsigned int res;
  res= a*a;
  res= res*b*a;
  res= res + 3;

  return res;
}
```

Pragma HLS Inline

#pragma HLS inline OFF

With HLS INLINE

```
Timing:
  * Summary:
  +--------+----------+----------+------------+
  |  Clock |  Target  | Estimated| Uncertainty|
  +--------+----------+----------+------------+
  |ap_clk  | 25.00 ns | 7.401 ns |   3.12 ns  |
  +--------+----------+----------+------------+

Latency:
  * Summary:
  +---------+---------+----------+----------+-----+-----+---------+
  |  Latency (cycles) |  Latency (absolute) |  Interval | Pipeline|
  |   min   |   max   |    min   |    max   | min | max |   Type  |
  +---------+---------+----------+----------+-----+-----+---------+
  |      121|      121| 3.025 us | 3.025 us |  121|  121|   none  |
  +---------+---------+----------+----------+-----+-----+---------+

  + Detail:
      * Instance:
      N/A

      * Loop:
      +------------+---------+---------+----------+-----------+-----------+------+----------+
      |            |  Latency (cycles) | Iteration|  Initiation Interval  | Trip |          |
      |  Loop Name |   min   |   max   |  Latency |  achieved |   target  | Count| Pipelined|
      +------------+---------+---------+----------+-----------+-----------+------+----------+
      |- for_Loop  |      120|      120|         2|          -|          -|    60|    no    |
      +------------+---------+---------+----------+-----------+-----------+------+----------+
```

With HLS INLINE OFF

```
Timing:
  * Summary:
  +--------+----------+----------+------------+
  |  Clock |  Target  | Estimated| Uncertainty|
  +--------+----------+----------+------------+
  |ap_clk  | 25.00 ns | 7.911 ns |   3.12 ns  |
  +--------+----------+----------+------------+

Latency:
  * Summary:
  +---------+---------+----------+----------+-----+-----+---------+
  |  Latency (cycles) |  Latency (absolute) |  Interval | Pipeline|
  |   min   |   max   |    min   |    max   | min | max |   Type  |
  +---------+---------+----------+----------+-----+-----+---------+
  |      121|      121| 3.025 us | 3.025 us |  121|  121|   none  |
  +---------+---------+----------+----------+-----+-----+---------+

+ Detail:
    * Instance:
    +--------------------+---------+---------+---------+----------+----------+-----+-----+---------+
    |                    |         |  Latency (cycles) |  Latency (absolute) |  Interval | Pipeline|
    |      Instance      |  Module |   min   |   max   |    min   |    max   | min | max |   Type  |
    +--------------------+---------+---------+---------+----------+----------+-----+-----+---------+
    |tmp_squared_fu_105  |squared  |        0|        0|   0 ns   |   0 ns   |    0|    0|   none  |
    |grp_func_fu_110     |func     |        0|        0|   0 ns   |   0 ns   |    0|    0|   none  |
    |tmp2_func_fu_119    |func     |        0|        0|   0 ns   |   0 ns   |    0|    0|   none  |
    +--------------------+---------+---------+---------+----------+----------+-----+-----+---------+

    * Loop:
    +------------+---------+---------+----------+-----------+-----------+------+----------+
    |            |  Latency (cycles) | Iteration|  Initiation Interval  | Trip |          |
    |  Loop Name |   min   |   max   |  Latency |  achieved |   target  | Count| Pipelined|
    +------------+---------+---------+----------+-----------+-----------+------+----------+
    |- for_Loop  |      120|      120|         2|          -|          -|    60|    no    |
    +------------+---------+---------+----------+-----------+-----------+------+----------+
```

Pragma HLS Inline

#pragma HLS inline **off**

With HLS INLINE

* Summary:

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	5	0	154	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	30	-
Register	-	-	117	-	-
Total	0	5	117	184	0
Available SLR	1440	2280	788160	394080	320
Utilization SLR (%)	0	~0	~0	~0	0
Available	4320	6840	2364480	1182240	960
Utilization (%)	0	~0	~0	~0	0

With HLS INLINE OFF

* Summary:

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	2	0	189	-
FIFO	-	-	-	-	-
Instance	-	5	0	58	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	45	-
Register	-	-	117	-	-
Total	0	7	117	292	0
Available SLR	1440	2280	788160	394080	320
Utilization SLR (%)	0	~0	~0	~0	0
Available	4320	6840	2364480	1182240	960
Utilization (%)	0	~0	~0	~0	0

+ Detail:
    * Instance:

Instance	Module	BRAM_18K	DSP48E	FF	LUT	URAM
grp_func_fu_110	func	0	1	0	19	0
tmp2_func_fu_119	func	0	1	0	19	0
tmp_squared_fu_105	squared	0	3	0	20	0
Total		0	5	0	58	0

Pragma HLS INLINE

Limitations:

- Large functions increase area (so duplicating may increase the resource significantly)
- Loss of Modular hierarchy
  - Debugging RTL becomes harder
- Recursive inline can explode design size
  - Useful for optimizations but dangerous for operations like CNN blocks, video pipelines
- Can worsen synthesis runtime
- Prevents Intentional sharing (reusability of hardware block)

TAC-HEP 2026

#pragma HLS Stable

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

Pragma HLS Stable

#pragma HLS stable variable=<name>

- The STABLE pragma marks variables within a DATAFLOW region as being stable
- Applies to both scalar and array variables whose content can be written/read by the process inside the DATAFLOW region
- Eliminates the extra synchronization involved for DATAFLOW region

Example:

- Specifies the array A as stable
- If A is read by proc2, then it will not be written by another process while the DATAFLOW region is being executed

```
void foo(int A[...], int B[...]){
#pragma HLS dataflow
#pragma HLS stable variable=A
  proc1(...);
  proc2(A, ...);

...
}
```

Pragma HLS Stable – Example 1

#pragma HLS STABLE variable=<name>

```
void matrix_multiply(int A[N][N], int B[N][N], int C[N][N]) {
   #pragma HLS stable variable=A
   #pragma HLS stable variable=B

   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
         C[i][j] = 0;
         for (int k = 0; k < N; k++) {
            C[i][j] += A[i][k] * B[k][j];
         }
      }
   }
}
```

- Arrays **A** and **B** maintain STABLE and deterministic behavior across multiple synthesis runs
- Especially helpful when you're performing matrix multiplication

Pragma HLS Stable – Example 2

```
void process_buffer(int data[N]) {
   #pragma HLS stable variable=data

   int temp[N];
   for (int i = 0; i < N; i++) {
      temp[i] = data[i] * 2;
   }

   for (int i = 0; i < N; i++) {
      data[i] = temp[i] + 5;
   }
}
```

Working with a buffer and want to pipeline the operations:

- Use of STABLE directive can ensure that the input/output behavior of the buffer doesn't change unpredictably during pipelining
- Without the stable directive, the tool might optimize the loop in a way that would change the data access patterns, leading to unpredictable results.

When NOT stable

In HLS, a variable is considered **unstable** if its behavior

- Such as value, timing, or memory access pattern: **can change between synthesis runs**, even when the functional source code stays the same
- This instability is usually due to **compiler optimizations** or **design transformations** performed by the HLS tool

- These changes are often valid from a software perspective, but in **hardware design**, they might **affect timing**, **control logic**, **latency**, or **verification**.

TAC-HEP 2026

What can cause these IN-STABILITY

Instability factors

Loop transformations (loop unrolling, pipelining)

- Number of times variables is accessed or updated
- Pipelining can overlap iterations & change the timing

Function Inlining & Optimization

- HLS may choose to inline a function or change the way it handles intermediate variables or buffers inside a function, affecting how a variable is stored, updated, or synthesized

Data Dependency Analysis

- Assumes a variable has no dependencies when it actually does (or vice versa), it may reorder or parallelize operations in a way that changes the variable's behavior.

Resource Sharing

- Share registers between variables.
- Map different variables to the same memory resource

Unstability factors

Bit-Width or Precision Optimizations

- If HLS tools infer or optimize variable bit-widths based on usage patterns, the inferred width (and resulting hardware) might vary between synthesis runs if inputs change slightly

Changing Clock Constraints / Timing Goals

- When timing goals change (e.g., tighter latency or initiation interval), HLS might restructure logic around a variable, making its storage or update behavior different

Random Seeds or Tool Heuristics

- Some HLS tools use internal seeds that can affect scheduling, binding, or resource allocation
- This can cause small changes in synthesis output from run to run

TAC-HEP 2026

Can we make all variables STABLE

All Variables as Stable

Loss of Optimization:

- Marking variables as stable restricts the HLS tool's freedom to optimize things like:
  - Loop pipelining
  - Resource sharing
  - Parallelization

Increased Resource Usage

- Variables marked stable might be mapped to **dedicated registers or memory**, avoiding sharing even when safe

Longer Synthesis Time

- The tool might work harder to maintain the "stable" access/timing pattern, leading to Longer synthesis runs and Harder timing closure

Use Stable Like You'd Use const in C++

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