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

FPGA module training

Week-10

Lecture-18: 08/04/2025

Varun Sharma

University of Wisconsin – Madison, USA

Content

So far

- HLS Pragmas:
  - Interface
  - Array Partition
  - Array reshape
  - Pipeline
  - Dataflow
  - Latency
  - Allocation
  - Stable
  - Inline
  - Unroll

Today

- Revisit Pragma Interface
- Examples for HLS Pragmas:
  - Inline
  - Stable

Interface overview

Function arguments in HLS are synthesized into **interfaces and ports**

- These group multiple signals and define **communication protocols**.
- **The goal:** connect HLS components to external elements
  - Example: memory, host, sensors

Interfaces define 3 key aspects of an HLS kernel:

1. **Data Channels:**
   - Enable data transfer into/out of the HLS design
   - **Sources:** host application, sensors, external IPs, or other kernels.
   - Default: AXI adapters (e.g., AXI4-Stream, AXI4-Memory).
2. **Port-Level Protocols:**
   - Control **when** data is valid for read/write.
   - Protocols manage flow using signals like *TVALID*, *TREADY*, etc.
   - Can be customized in **Vivado IP flow**, fixed in **Vitis kernel flow**.
3. **Execution Control Scheme:**
   - Defines kernel/IP operation as **pipelined** or **sequential**.
   - Controlled by block-level protocols.

Defining Interfaces: examples

Function arguments in HLS are synthesized into **interfaces and ports**

- These group multiple signals and define **communication protocols**.
- **The goal:** connect HLS components to external elements
  - Example: memory, host, sensors

Image processing from a camera

- Apply a filter to live video
- Camera sends pixel data via **AXI-4 stream** to an HLS block
- Processes image in real time
- Output stream sent to display or written to memory

HLS block connected to **external camera (sensor)** via streaming interface

Accelerating Matrix Multiplication with external DDR

- Large matrix multiplication that can't fit on-chip
- Host application sends matrices to **external DDR memory**
- Reads matrices using **AXI-4 memory** mapped interface
- Performs multiplication
- Writes result back to memory

HLS block connected to **DDR memory** through an AXI interface

Defining Interfaces: examples

Function arguments in HLS are synthesized into **interfaces and ports**

- These group multiple signals and define **communication protocols**.
- **The goal:** connect HLS components to external elements
  - Example: memory, host, sensors

Data Transfer from Host CPU

- Offload compute-heavy operation (e.g. encryption) to FPGA
- Sends data via PCIe to FPGA
- HLS kernel receives data through **AXI-4 stream**
- Processes encription
- Sends results back to host

HLS block interfaces using **AXI4-stream** and **AXI-Lite** (for control)

Interface type

Interface type is chosen automatically by the tool based on:

- **Data type and direction** of each argument
- The **target flow** (Vivado IP flow or Vitis Kernel flow)
- **Default interface settings**
- Any applied **INTERFACE pragmas/directives**

Choosing and configuring the right interface is **critical** for design success

Target Flows

Vivado IP Flow: Creates custom RTL IP blocks to integrate into a larger FPGA design (via Vivado)

- HLS generates RTL IP → Added to **Vivado Block Design**.
- One can use **AXI interfaces**, but you **have full control** over customization.
- Supports **manual control** of interfaces (can customize protocols, ports, etc.).
- Great for **FPGA developers** building entire systems in Vivado
- Packaged IP block that you integrate into a **Vivado design project**

Vitis Kernel Flow: Used in application acceleration design

- Writing **accelerator kernels** that run on an FPGA as part of a **heterogeneous system** (with host CPU + FPGA), e.g., on a **Zynq** or **Alveo** card
- HLS function becomes a **Vitis kernel**.
- Interacts with host software via **OpenCL/XRT runtime**.
- Interfaces are **automatically set** (AXI-Stream for data, AXI-Lite for control).
- No manual tweaking of interface protocols — standardized for portability.
- A **compiled kernel** (e.g., .xo) used in **Vitis platforms and applications**

Advanced eXtensible Interface (AXI)

AXI is part of ARM AMBA, a family of micro controller buses first introduced in 1996

There are three types of AXI4 interfaces:

- **AXI4 (m_axi):** For high-performance memory-mapped requirements
  - Specify on array and pointers (and references in C++)

- **AXI4-Lite (s_axilite):** For simple, low-throughput memory-mapped communication (for example, to and from control and status registers)
  - Can be used on any type of argument except streams

- **AXI4-Stream (axis):** For high-speed streaming data
  - Use this protocol on input arguments or output arguments only,

Pragma HLS interface

C-argument type	Paradigm	Interface protocol (I/O/Inout)
Scalar(pass by value)	Register	AXI4-Lite (`s_axilite`)
Array	Memory	AXI4 Memory Mapped (`m_axi`)
Pointer to array	Memory	`m_axi`
Pointer to scalar	Register	`s_axilite`
Reference	Register	`s_axilite`
hls::stream	Stream	AXI4-Stream (`axis`)

TAC-HEP 2025

Pragmas INLINE & STABLE

Pragma HLS Inline

#pragma HLS INLINE <recursive | OFF>

- Controls function inlining

- The function is dissolved into the calling function and no longer appears as a separate level of hierarchy in RTL design

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

Pragma INLINE: Example

```
#include "example.h"

void example (int input[N], int output[N]) {
  for_Loop:
  for (size_t i=0 ; i < N; i++) {
    output[i] = square(abs_val(input[i]));
    }
}

int square(int a){
//#pragma HLS INLINE OFF
 return a*a;
}

int abs_val(int a){
//#pragma HLS INLINE OFF
 return (a < 0) ? -a : a;
}
```

```
Timing:
  * Summary:
  +--------+----------+----------+------------+
  |  Clock |  Target  | Estimated| Uncertainty|
  +--------+----------+----------+------------+
  |ap_clk  | 25.00 ns | 7.922 ns |   6.75 ns  |
  +--------+----------+----------+------------+

Latency:
  * Summary:
  +---------+---------+----------+----------+-----+-----+---------+
  |  Latency (cycles) |  Latency (absolute) |  Interval | Pipeline|
  |   min   |   max   |    min   |    max   | min | max |   Type  |
  +---------+---------+----------+----------+-----+-----+---------+
  |      102|      102| 2.550 us | 2.550 us |  103|  103|   none  |
  +---------+---------+----------+----------+-----+-----+---------+

  + Detail:
      * Instance:
      +-----------------+--------+---------+---------+----------+----------+-----+-----+----------+
      |                 |        |  Latency (cycles) |  Latency (absolute) |  Interval | Pipeline |
      |     Instance    | Module |   min   |   max   |    min   |    max   | min | max |   Type   |
      +-----------------+--------+---------+---------+----------+----------+-----+-----+----------+
      |tmp_square_fu_85 |square  |        0|        0|   0 ns   |   0 ns   |    1|    1| function |
      +-----------------+--------+---------+---------+----------+----------+-----+-----+----------+

      * Loop:
      +-----------+---------+---------+----------+-----------+-----------+-----+----------+
      |           |  Latency (cycles) | Iteration|  Initiation Interval  | Trip |          |
      | Loop Name |   min   |   max   |  Latency |  achieved |   target  | Count| Pipelined|
      +-----------+---------+---------+----------+-----------+-----------+-----+----------+
      |- for_Loop |      100|      100|         2|          1|          1|  100|    yes   |
      +-----------+---------+---------+----------+-----------+-----------+-----+----------+


 Summary:
+---------------------+---------+------+---------+---------+-----+
|         Name        | BRAM_18K|  DSP |    FF   |   LUT   | URAM|
+---------------------+---------+------+---------+---------+-----+
|DSP                  |        -|     -|        -|        -|    -|
|Expression           |        -|     -|        0|      101|    -|
|FIFO                 |        -|     -|        -|        -|    -|
|Instance             |        -|     3|        0|       20|    -|
|Memory               |        -|     -|        -|        -|    -|
|Multiplexer          |        -|     -|        -|       45|    -|
|Register             |        -|     -|       20|        -|    -|
+---------------------+---------+------+---------+---------+-----+
|Total                |        0|     3|       20|      166|    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|
+---------------------+---------+------+---------+---------+-----+
```

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 2025

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 2025

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++

Assignment-6

- Use example in slide-3 of lecture 17 to reduce resource utilization – specially the DSP usage (https://github.com/varuns23/TAC-HEP-FPGA/tree/main/tutorial/wk9lec17/ex-func)
  - You can use a combination of sub-set of following pragmas:
    - Array Partition
    - Array reshape
    - Pipeline
    - Dataflow
    - Latency
    - Allocation
    - INLINE
  - **Objective: To have DSP usage less than 10**
- Refer to ex-all folder for example with pragmas

Reminder: Assignments

- Assignment-1 (13-02-2025)
- Assignment-2 (18-02-2025)
- Assignment-3 (27-02-2025)
- Assignment-4 (18-03-2025)
- Assignment-5 (18-03-2025)

Uploaded to cernbox: https://cernbox.cern.ch/s/gmUqRDHTxDLqx4M

Send via email: **varun.sharma@cern.ch**

Submit in 2 weeks from date of assignment

TAC-HEP 2025

Questions?

Acknowledgements:
- https://docs.amd.com/r/2024.1-English/ug1399-vitis-hls
- ug871-vivado-high-level-synthesis-tutorial.pdf

List of Available Pragmas

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

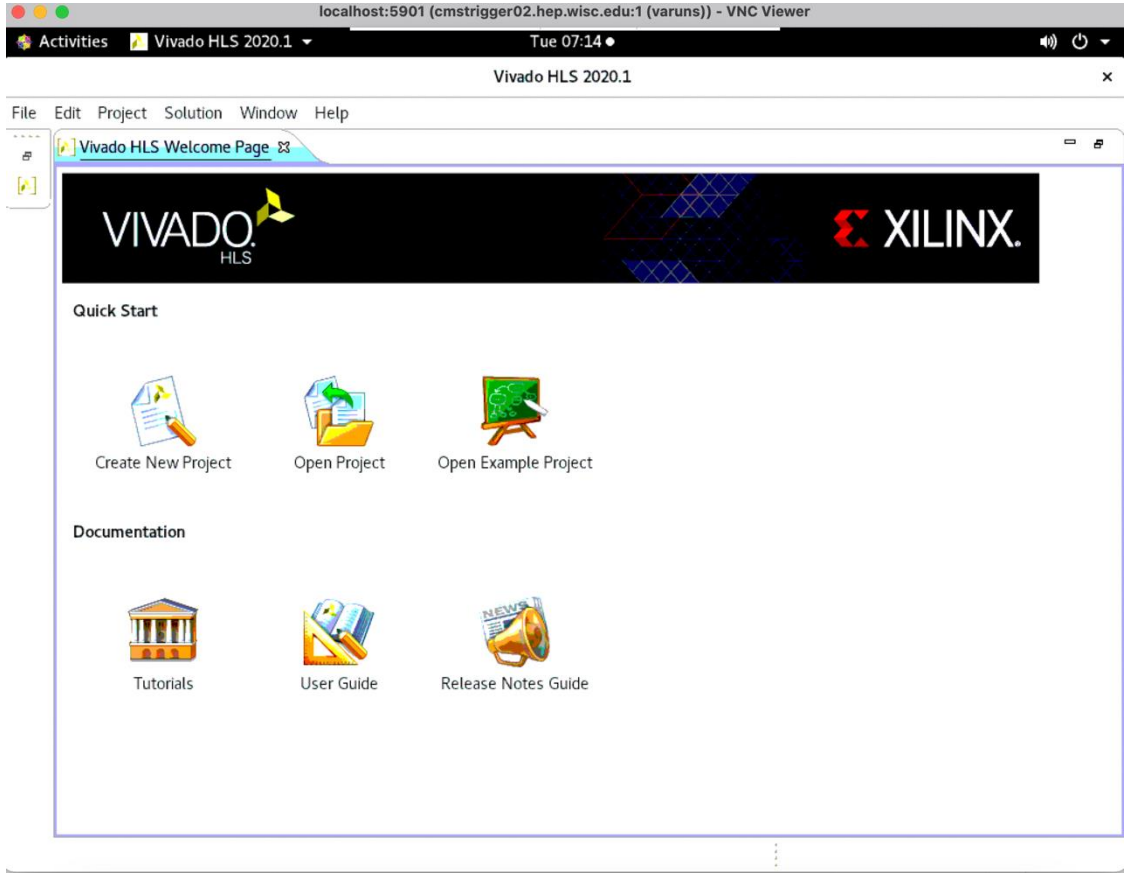
Reminder: HLS Setup

- ssh <username>@cmstrigger02-via-login -L5901:localhost:5901
  - Or whatever ***:1*** display number
  - Sometimes you may need to run vncserver -localhost -geometry 1024x768 again to start new vnc server

- **Connect to VNC server (remote desktop) client**
- **Open terminal**
  - source /opt/Xilinx/Vivado/2020.1/settings64.sh
  - cd /scratch/`whoami`
  - vivado_hls

OR

  - Source /opt/Xilinx/Vitis/2020.1/settings64.sh
  - Cd /scratch/`whoami`
  - vitis_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