

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

FPGA module training

Week-9

Lecture-17: 27/03/2025



Varun Sharma
University of Wisconsin – Madison, USA



Content



So far

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

Today

- HLS Pragmas:
 - Stable
 - Inline
 - Unroll

Example: Box blur operation



```
#include "example.h"

void read_data(din9_t in_r[N], din9_t out_r[N]) {
    for (size_t i = 0; i < N; i++) {
        out_r[i] = in_r[i]; // Simple pass through
    }
}

void compute_blur(din9_t in_c[N], din9_t out_c[N]) {
    for (size_t i = 1; i < N - 1; i++) {
        out_c[i] = (in_c[i - 1] + in_c[i] + in_c[i + 1]) / 3; // Box blur operation
    }
}

void write_data(din9_t in_w[N], din9_t out_w[N]) {
    for (size_t i = 0; i < N; i++) {
        out_w[i] = in_w[i]; // Simple pass-through
    }
}

void example(din9_t A[N], din9_t B[N]) {
    din9_t temp1[N], temp2[N];
    for (size_t i = 0; i < N; i++)
        temp2[i]=0;

    read_data(A, temp1);
    compute_blur(temp1, temp2);
    write_data(temp2, B);
}
```

Example: Resource



Timing:

* Summary:

Clock	Target	Estimated	Uncertainty
ap_clk	25.00 ns	4.478 ns	3.12 ns

Latency:

* Summary:

Latency (cycles)		Latency (absolute)		Interval		Pipeline
min	max	min	max	min	max	Type
35	35	0.875 us	0.875 us	15	15	dataflow

+ Detail:

* Instance:

		Latency (cycles)		Latency (absolute)		Interval		Pipeline
Instance	Module	min	max	min	max	min	max	Type
compute_blur_U0	compute_blur	4	4	0.100 us	0.100 us	2	2	function
read_data_U0	read_data	15	15	0.375 us	0.375 us	15	15	function
write_data_U0	write_data	14	14	0.350 us	0.350 us	15	15	function

Example: Resource



* Summary:

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	242	-
FIFO	0	-	290	1218	-
Instance	-	28	1363	1610	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	522	-
Register	-	-	58	-	-
Total	0	28	1711	3592	0
Available SLR	1440	2280	788160	394080	320
Utilization SLR (%)	0	1	~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
compute_blur_U0	compute_blur	0	28	807	776	0
read_data_U0	read_data	0	0	540	498	0
write_data_U0	write_data	0	0	16	336	0
Total		0	28	1363	1610	0

Lets implement some pragmas



```
#include "example.h"

void read_data(din9_t in_r[N], din9_t out_r[N]) {
    for (size_t i = 0; i < N; i++) {
        out_r[i] = in_r[i]; // Simple pass through
    }
}

void compute_blur(din9_t in_c[N], din9_t out_c[N]) {
    for (size_t i = 1; i < N - 1; i++) {
        out_c[i] = (in_c[i - 1] + in_c[i] + in_c[i + 1]) / 3; // Box blur operation
    }
}

void write_data(din9_t in_w[N], din9_t out_w[N]) {
    for (size_t i = 0; i < N; i++) {
        out_w[i] = in_w[i]; // Simple pass-through
    }
}

void example(din9_t A[N], din9_t B[N]) {
    din9_t temp1[N], temp2[N];
    for (size_t i = 0; i < N; i++)
        temp2[i]=0;

    read_data(A, temp1);
    compute_blur(temp1, temp2);
    write_data(temp2, B);
}
```

What all pragmas and optimization can we do with this code?

Example: with Pragmas



```
#include "example.h"

void read_data(din9_t in_r[N], din9_t out_r[N]) {
    #pragma HLS PIPELINE II=2
    #pragma HLS ARRAY_PARTITION variable=out_r cyclic factor=2
    for (size_t i = 0; i < N; i++) {
        out_r[i] = in_r[i]; // Simple pass through
    }
}

void compute_blur(din9_t in_c[N], din9_t out_c[N]) {
    #pragma HLS PIPELINE II=2
    #pragma HLS LATENCY min=4 max=8
    #pragma HLS ALLOCATION instances=mul limit=1 function
    #pragma HLS ARRAY_PARTITION variable=in_c cyclic factor=2
    #pragma HLS ARRAY_PARTITION variable=out_c cyclic factor=2
    for (size_t i = 1; i < N - 1; i++) {
        out_c[i] = (in_c[i - 1] + in_c[i] + in_c[i + 1]) / 3; // Box blur operation
    }
}

void write_data(din9_t in_w[N], din9_t out_w[N]) {
    #pragma HLS PIPELINE II=2
    #pragma HLS ARRAY_PARTITION variable=in_w cyclic factor=2
    for (size_t i = 0; i < N; i++) {
        out_w[i] = in_w[i]; // Simple pass-through
    }
}

void example(din9_t A[N], din9_t B[N]) {
    #pragma HLS DATAFLOW
    din9_t temp1[N], temp2[N];
    for (size_t i = 0; i < N; i++)
        temp2[i] = 0;

    read_data(A, temp1);
    compute_blur(temp1, temp2);
    write_data(temp2, B);
}
```

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 Inline

Function inlining



```
#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

Function inlining



```
#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:

Loop Name	Latency (cycles)		Iteration Latency	Initiation achieved	Interval target	Trip Count	Pipelined
	min	max					
- 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:

Instance	Module	Latency (cycles)		Latency (absolute)		Interval		Pipeline
		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:

Loop Name	Latency (cycles)		Iteration Latency	Initiation achieved	Interval target	Trip Count	Pipelined
	min	max					
- 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 unroll



- Unroll loops to create multiple independent operations rather than a single collection of operations
- **UNROLL** pragma transforms loops by creating multiples copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel
- Loops in the C/C++ functions are kept rolled by default
 - When loops are rolled, synthesis creates the logic for one iteration of the loop, and the RTL design executes this logic for each iteration of the loop in sequence
- **UNROLL** pragma allows the loop to be fully or partially unrolled
 - Fully unrolling the loop creates a copy of the loop body in the RTL for each loop iteration, so the entire loop can be run concurrently
 - Partially unrolling a loop lets you specify a factor N

Pragma HLS unroll



```
#pragma HLS unroll factor=<N> skip_exit_check off=true
```

factor=<N>: Specifies a non-zero integer indicating that partial unrolling is requested.

- If factor= is not specified, the loop is fully unrolled.

skip_exit_check: An optional keyword that applies only if partial unrolling is specified with **factor=**

- **Fixed (known) bounds:** No exit condition check is performed if the iteration count is a multiple of the factor. If the iteration count is not an integer multiple of the factor, the tool:
 - Prevents unrolling.
 - Issues a warning that the exit check must be performed to proceed.
- **Variable (unknown) bounds:** The exit condition check is removed as requested. You must ensure that:
 - The variable bounds is an integer multiple of the specified unroll factor.
 - No exit check is in fact require

Ex: Pragma HLS unroll



```
#pragma HLS unroll factor=<N> skip_exit_check off=true
```

The following example fully unrolls loop_1 in function foo

Place the pragma in the body of loop_1 as shown:

```
loop_1: for(int i = 0; i < N; i++) {  
    #pragma HLS unroll  
    a[i] = b[i] + c[i];  
}
```

This example specifies an unroll factor of 4 to partially unroll loop_2 of function foo, and removes the exit check:

```
void foo (...) {  
    int8 array1[M];  
    int12 array2[N];  
    ...  
    loop_2: for(i=0;i<M;i++) {  
        #pragma HLS unroll skip_exit_check factor=4  
        array1[i] = ...;  
        array2[i] = ...;  
        ...  
    }  
    ...  
}
```


Assignment-6



- Use example in slide-3 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



Questions?

Acknowledgements:

- <https://docs.amd.com/r/en-US/ug1399-vitis-hls/HLS-Pragmas>
- ug871-vivado-high-level-synthesis-tutorial.pdf

List of Available Pragmas



Type	Attributes
Kernel Optimization	<ul style="list-style-type: none"> • <code>pragma HLS aggregate</code> • <code>pragma HLS alias</code> • <code>pragma HLS disaggregate</code> • <code>pragma HLS expression_balance</code> • <code>pragma HLS latency</code> • <code>pragma HLS performance</code> • <code>pragma HLS protocol</code> • <code>pragma HLS reset</code> • <code>pragma HLS top</code> • <code>pragma HLS stable</code>
Function Inlining	<ul style="list-style-type: none"> • <code>pragma HLS inline</code>
Interface Synthesis	<ul style="list-style-type: none"> • <code>pragma HLS interface</code> • <code>pragma HLS stream</code>
Task-level Pipeline	<ul style="list-style-type: none"> • <code>pragma HLS dataflow</code> • <code>pragma HLS stream</code>
Pipeline	<ul style="list-style-type: none"> • <code>pragma HLS pipeline</code> • <code>pragma HLS occurrence</code>

Loop Unrolling	<ul style="list-style-type: none"> • <code>pragma HLS unroll</code> • <code>pragma HLS dependence</code>
Loop Optimization	<ul style="list-style-type: none"> • <code>pragma HLS loop_flatten</code> • <code>pragma HLS loop_merge</code> • <code>pragma HLS loop_tripcount</code>
Array Optimization	<ul style="list-style-type: none"> • <code>pragma HLS array_partition</code> • <code>pragma HLS array_reshape</code>
Structure Packing	<ul style="list-style-type: none"> • <code>pragma HLS aggregate</code> • <code>pragma HLS dataflow</code>
Resource Utilization	<ul style="list-style-type: none"> • <code>pragma HLS allocation</code> • <code>pragma HLS bind_op</code> • <code>pragma HLS bind_storage</code> • <code>pragma HLS function_instantiate</code>

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

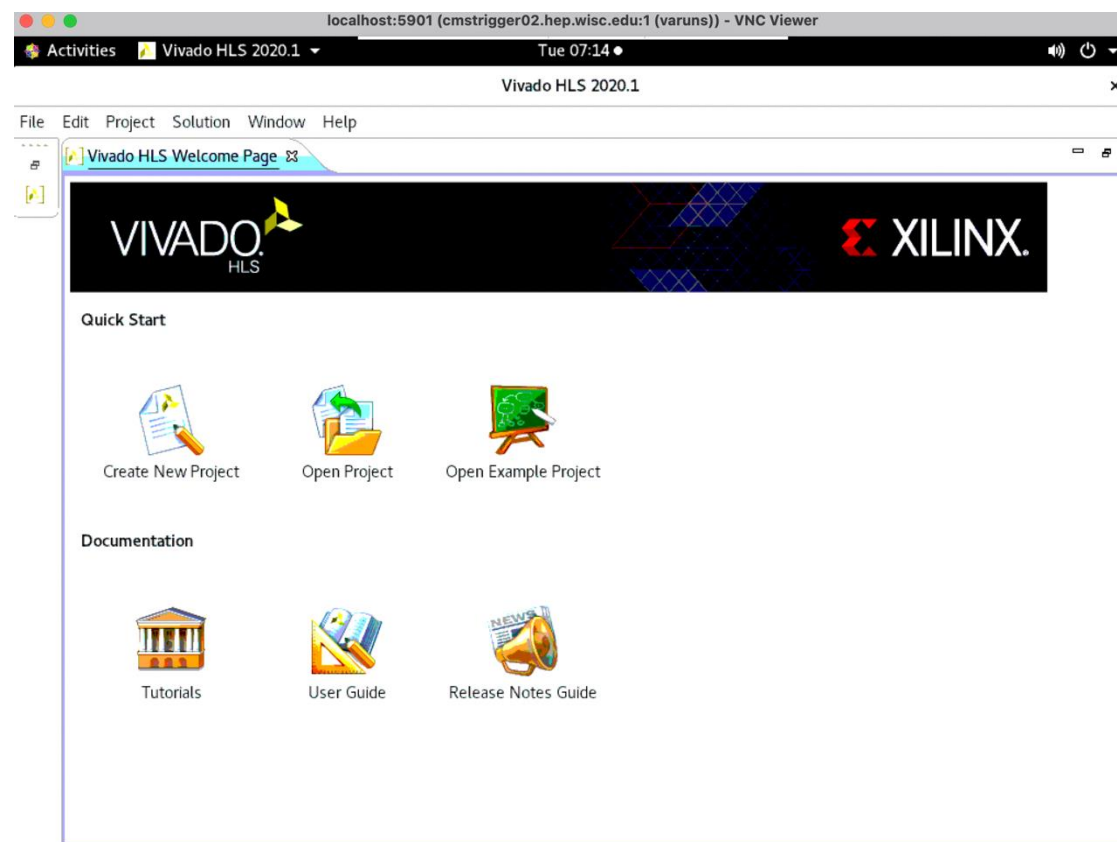
- **Connect to VNC server (remote desktop) client**

- **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

