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

FPGA module training

Week-12

Lecture-22: 22/04/2025

Varun Sharma

University of Wisconsin – Madison, USA

Content

So far

- HLS Pragmas:
  - Aggregate
  - Expression_balance
  - Latency
  - Performance
  - Protocol
  - Stable
  - Inline
  - Interface
  - Dataflow
  - Pipeline
  - Unroll
  - Loop_flatten
  - Loop_merge
  - Loop_tripcount
  - Array Partition
  - Array reshape
  - Allocation

Today

- HLS Pragmas
  - HLS bind_op
  - HLS bind_storage
  - HLS function_instantiate
  - HLS Stream
  - HLS Dependence

HLS bind_op

- Vitis HLS assigns specific operations (*e.g., mul, add, div*) to specific device resources for RTL implementation
- If BIND_OP is **not** used, Vitis HLS automatically selects appropriate resources for operations.
- DSP multi-operation matching enables optimized hardware mapping of multiple operations into a single DSP block
- 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;
}
```

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

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;
}
```

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);
}
```

HLS Stream

By default, array variables are implemented as RAM:

- Top-level function array parameters are implemented as a RAM interface port
- General arrays are implemented as RAMs for read-write access.
- Arrays involved in sub-functions, or loop-based DATAFLOW optimizations are implemented as a RAM ping pong buffer channel.

- If the data stored in the array is consumed or produced in a sequential manner, a more **efficient** communication mechanism is to **use streaming data** as specified by the **STREAM** pragma, where **FIFOs** are used instead of **RAMs**.

#pragma HLS Stream variable=<variable> type=<type> depth=<int>

#pragma HLS STREAM variable=B depth=12 type=fifo

HLS Dependence

Vitis HLS detects dependencies within loops:

- dependencies within the same iteration of a loop are loop-independent dependencies
- dependencies between different iterations of a loop are loop-carried dependencies.

The DEPENDENCE pragma allows you to provide additional information to define, negate loop dependencies, and allow loops to be pipelined with lower intervals

Loop-independent dependence

```
for (i=1;i<N;i++) {
 A[i]=x;
 y=A[i];
}
```

The same element is accessed in a single loop iteration

Loop-carried dependence

```
for (i=1;i<N;i++) {
 A[i]=A[i-1]*2;
}
```

The same element is accessed from a different loop iteration

HLS Dependence

- **Dependencies affect operation scheduling**, especially during **function and loop pipelining**.
- Automatic dependence analysis may be too conservative, especially in cases like:
  - Variable-dependent array indexing.
  - External requirements (e.g., two inputs never having the same index).
- DEPENDENCE pragma allows explicit definition of dependencies, helping:
  - **Eliminate false dependencies**.
  - Improve pipelining and scheduling.
- Applies to function call arguments as well:
  - **Inter-dependence**: Between successive function calls
  - **Intra-dependence**: Within a single function call

HLS Dependence

**#pragma HLS dependence variable=<variable> <class> <type> **
<direction> distance=<int> <dependent>

```
 void foo(int a[3], int x) {
  #pragma HLS pipeline
  #pragma HLS dependence variable=a inter true distance=3
  a[x] += ...;
}

void dut(int a[3], int x, ...) {
  #pragma HLS dataflow
  foo(a, x);
  bar(...);
}
```

HLS Dependence

```
void foo(int rows, int cols, ...){
  for (row = 0; row < rows + 1; row++) {
    for (col = 0; col < cols + 1; col++) {
    #pragma HLS PIPELINE II=1
    #pragma HLS dependence variable=buff_A type=inter false
    #pragma HLS dependence variable=buff_B type=inter false
      if (col < cols) {
         buff_A[2][col] = buff_A[1][col]; // read from buff_A[1][col]
         buff_A[1][col] = buff_A[0][col]; // write to buff_A[1][col]
         buff_B[1][col] = buff_B[0][col];
      }
    }
  }
}
```

What all pragma can we use?

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

#include "example.h"

void example(din_t A[N][N], din_t B[N][N], din_t C[N][N]) {

 for (size_t i = 0; i < N; ++i) {
  for (size_t j = 0; j < N; ++j) {
   C[i][j]=0;
//#pragma HLS UNROLL factor = 4
   for (size_t k = 0; k < N; ++k) {

    C[i][j] += A[i][k] * B[k][j];
   }
  }
 }
}
```

HLS Pragmas:

- Interface
- Array Partition
- Array reshape
- Pipeline
- Dataflow
- Latency
- Allocation
- Stable
- Inline
- Unroll
- Aggregate
- Expression_balance
- Performance
- Protocol

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)
- Assignment-6 (27-03-2025)
- Assignment-6a (15-04-2025)
- Assignment-7 (15-04-2025)

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