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

FPGA module training

Week-12

Lecture-23: 24/04/2025

Varun Sharma

University of Wisconsin – Madison, USA

Content

So far

HLS Pragmas:

- Aggregate
- Expression_balance
- Latency
- Performance
- Protocol
- Stable
- Inline
- Interface
- Stream
- Dataflow
- Pipeline
- Unroll
- Dependence
- Loop_flatten
- Loop_merge
- Loop_tripcount
- Array Partition
- Array reshape
- Allocation
- Bind_op
- Bind_storage
- Function_instantiate

Today

- Some Examples
- HLS Occurrence

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 bind_op:

#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

```
void example(dataIN_t x[N], dataIN_t h[N], dataOUT_t &y) {
  dataOUT_t prod = 0;
  dataOUT_t acc = 0;
#pragma HLS INLINE off
#pragma HLS bind_op op=mul impl=DSP variable=prod latency=2
#pragma HLS pipeline

 for (size_t i = 0; i < N; i++) {
    prod = x[i] * h[i];
    acc += prod;
  }
  y = acc;
}
```

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: 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: Example

```
void example(dataIN_t x[N], dataIN_t h[N], dataOUT_t &y) {
  dataOUT_t prod = 0;
  dataOUT_t acc = 0;
#pragma HLS INLINE off
#pragma HLS bind_storage variable=h type=ram_2p impl=bram  // Bind filter coefficients to BRAM
//#pragma HLS bind_storage variable=x type=ram_2p storage=lutram  // Bind input samples to LUTRAM
#pragma HLS bind_storage variable=x type=ram_2p impl=uram  // Bind input samples to LUTRAM
#pragma HLS pipeline

 for (size_t i = 0; i < N; i++) {
    prod = x[i] * h[i];
    acc += prod;
  }
  y = acc;
}
```

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

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

variable=<variable>Optionally specifies the variable to consider for the dependence

class=[array | pointer]. (Class and variable should not be specified together)

type=[inter | intra]

- **Intra:** dependence within the same loop iteration
- **Inter:** dependence between different loop iterations.

direction=[RAW | WAR | WAW]

- RAW (Read-After-Write - true dependence) The write instruction uses a value used by the read instruction.
- WAR (Write-After-Read - anti dependence) The read instruction gets a value that is overwritten by the write instruction.
- WAW (Write-After-Write - output dependence) Two write instructions write to the same location, in a certain order.

distance=<int>: Specifies the inter-iteration distance for array access. Relevant only for loop-carry dependencies where dependence is set to true.

HLS Dependence

```
#include "example.h"

void example(int *a) {
#pragma HLS pipeline
   for (int i = 1; i < N; i++) {
#pragma HLS dependence variable=a inter RAW distance=2
      a[i] = a[i - 2] + 1;
   }
}
```

```
#include "example.h”

void example(int *a) {
#pragma HLS pipeline
   for (int i = 1; i < N; i++) {
#pragma HLS dependence variable=a inter WAW distance=1
     int temp = a[i];
     a[i%2]=i;
   }
}
```

```
#include "example.h”

void example(int *a) {
#pragma HLS pipeline
   for (int i = 1; i < N; i++) {
#pragma HLS dependence variable=a inter WAR distance=1
     int temp = a[i];
     a[i+1]=temp+1;
   }
}
```

```
void no_dependence_example(int *a) {
#pragma HLS pipeline
   for (int i = 0; i < 10; i++) {
#pragma HLS dependence variable=a inter false
      a[i] = i * 2;
   }
}
```

HLS Occurrence

- In a pipelined function or loop: the **OCCURRENCE** pragma indicates that a region of code runs **less frequently** than the surrounding loop or function
- Allows HLS tools to **pipeline that region at a slower rate**, which can reduce resource usage

Useful when part of a loop body is **conditionally executed**

How to determine the OCCURENCE

- Let the loop iterate <**N**> times.
- If a block inside it runs only <**M**> times, then its occurrence is <**N/M**>
- Example: If a loop runs **10 times**, and a conditionally executed block runs only **2 times**, the occurrence is **10/2 = 5**.
- The OCCURRENCE region can be **shared** in the top-level pipeline and have a **higher initiation interval (II)**.

HLS Occurrence: Syntax

#pragma HLS OCCURRENCE cycle=<int>

```
void example(dataIN_t x[N], dataOUT_t &y) {
#pragma HLS PIPELINE II=2
 y=0;

  dataOUT_t temp=0;
  for (size_t i = 0; i < N; i++) {
    temp += x[i];
    if(i % 5 == 0){
      //this block executes only 10 times out of 50
#pragma HLS OCCURRENCE cycle=5
      for(size_t j=0; j<N/2; ++j)
        temp += x[j+1] + calc(x[j])*3;
    }
  }
  y = temp;
}
```

Cycle = <int>: Specify the occurrence N/M

<N>: number of times the enclosing function or loop is executed

<M>: Number of times the conditional region is executed

Assignment-8:

Q. Use as many pragmas as possible from the list to minimize the latency and resource utilization for following code. Report the new code, latency, timing and resource utilization.

```
#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:

- Aggregate
- Expression_balance
- Latency
- Performance
- Protocol
- Stable
- Inline
- Occurrence
- Interface
- Stream
- Dataflow
- Pipeline
- Unroll
- Dependence
- Loop_flatten
- Loop_merge
- Loop_tripcount
- Array Partition
- Array reshape
- Allocation
- Bind_op
- Bind_storage
- Function_instantiate

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)
- Assignment-8 (24-04-2025)

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