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

FPGA module training

Week-11

Lecture-20: 15/04/2025

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

- Kernel Optimzation
- HLS Pragmas
  - Aggregate, Expression_balance, performance, protocol

Assignment-6a

- 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: Remove usage of DSP**
- Refer to ex-all folder for example with pragmas

Assignment-7

- Example: https://github.com/varuns23/TAC-HEP-FPGA/tree/main/tutorial/wk11lec20/ex-matmul
- Run code for N=64 and use the sorter get a sorted output
  - May require restructuring of C-code
  - Add pragmas in order to minimize the resource utilization & timing constraints and synthesize the code

Kernel in HLS

- Specific, reusable module or function that performs computational tasks on FPGA
- Naïve code often leads to poor hardware designs

- **Kernel optimization is essential to**
  - Reach timing & resource constraints
  - Maximize performance of the hardware accelerator
  - Minimize power and area for embedded or constrained environments
  - Exploit the parallelism and pipelining possible in hardware

Kernel Optimization Pragmas

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

Pragma HLS AGGREGATE

Collects and groups struct data fields into a single wide scalar (wider word width)

Uses the **AGGREGATE pragma** to enable simultaneous read/write of all struct members

Bit alignment is based on the order of struct elements:

- First element aligns with the LSB.
- Last element aligns with the MSB.

Arrays within the struct are:

- Fully partitioned
- Reshaped similarly to ARRAY_RESHAPE
- Packed into the wide scalar along with other elements

Pragma HLS AGGREGATE: Syntax

#pragma HLS AGGREGATE variable=<var> compact=<arg>

- **variable = <variable>:** Specifies the variable to be grouped
- **compact = [bit | byte | none | auto]:** Specifies the alignment of the aggregated struct
  - Alignment can be on the bit-level, the byte-level, none, or automatically determined by the tool which is the default behavior

Caution: Careful while using AGGREGATE optimization on struct objects with large arrays. If an array has 4096 elements of type int, this will result in a vector (and port) of width 4096×32=131072 bits. The Vitis HLS tool can create this RTL design, however it is very unlikely logic synthesis will be able to route this during the FPGA implementation.

HLS AGGREGATE: Example-1

```
#include <ap_int.h>

typedef struct {
    ap_uint<8>  a;
    ap_uint<16> b;
    ap_uint<32> c;
} my_struct;

void process(my_struct in[4], my_struct out[4]) {
#pragma HLS AGGREGATE variable=in compact=bit
#pragma HLS AGGREGATE variable=out compact=bit
#pragma HLS PIPELINE

    for (int i = 0; i < 4; i++) {
        out[i].a = in[i].a + 1;
        out[i].b = in[i].b + 2;
        out[i].c = in[i].c + 3;
    }
}
```

HLS combines all fields into a **single wide port**, improving efficiency on AXI streams or memory-mapped interfaces

HLS AGGREGATE: Example-1

```
#include <ap_int.h>

typedef struct {
    ap_uint<8>  a;
    ap_uint<16> b;
    ap_uint<32> c;
} my_struct;

void process(my_struct in[4], my_struct out[4]) {
#pragma HLS AGGREGATE variable=in compact=bit
#pragma HLS AGGREGATE variable=out compact=bit
#pragma HLS PIPELINE

    for (int i = 0; i < 4; i++) {
        out[i].a = in[i].a + 1;
        out[i].b = in[i].b + 2;
        out[i].c = in[i].c + 3;
    }
}
```

HLS combines all fields into a **single wide port**, improving efficiency on AXI streams or memory-mapped interfaces

Without AGGREGATE, the struct might use separate interface ports.

HLS AGGREGATE: Example-2

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

typedef struct {
   ap_uint<8>  id;
   ap_uint<16> val;
} packet_t;

typedef ap_axiu<32, 0, 0, 0> axis_t;

void pack_stream(hls::stream<packet_t> &in, hls::stream<axis_t> &out) {
#pragma HLS AGGREGATE variable=in compact=bit
#pragma HLS INTERFACE axis port=in
#pragma HLS INTERFACE axis port=out
#pragma HLS PIPELINE

   packet_t pkt = in.read();
   axis_t out_pkt;
   out_pkt.data = (pkt.val, pkt.id);  // concat into 24 bits
   out_pkt.last = 0;
   out.write(out_pkt);
}
```

```
typedef struct{
unsigned char R, G, B;
} pixel;

pixel AB[17];
#pragma HLS aggregate variable=AB
```

Pragma HLS Expression_balance

- Sequential C/C++ operations can form long RTL operation chains, increasing latency with small clock periods.
- Vitis HLS rearranges operations using associative and commutative properties to reduce latency
- This rearrangement forms a balanced tree of operations, known as **expression balancing.** Defaults:
  - **Enabled for integer operations**
  - **Disabled for floating point operation**

#pragma HLS Expression_balnce

Expression_balance: Example

```
void top_function(int A[SIZE], int B[SIZE], int C[SIZE]) {
#pragma HLS DATAFLOW
#pragma HLS expression_balance

    hls::stream<int> a_s, b_s, sum_s;

#pragma HLS STREAM variable=a_s depth=2
#pragma HLS STREAM variable=b_s depth=2
#pragma HLS STREAM variable=sum_s depth=2

    load(A, B, a_s, b_s);
    add(a_s, b_s, sum_s);
    multiply_by_two(sum_s, C);
```

Pragma HLS Performance

The PERFORMANCE pragma/directive sets lets you specify a high-level constraint

- loop execution timing (target_ti or target_tl)
- Helps guide the tool to apply lower-level optimizations like UNROLL, PIPELINE, ARRAY_PARTITION, and INLINE

The directive does **not guarantee** the specified performance — it's only a goal.

target_ti: target transaction interval - number of clock cycles for the function, loop or region of code to complete an iteration
target_tl: target latency **-** number of clock cycles for the loop to complete all iterations

#pragma HLS performance target_ti=<value> target_tl=<value> unit=[sec | cycle]

HLS Performance: Example

```
 for (int i =0; i < 1000; ++i) {

#pragma HLS performance target_ti=1000

   for (int j = 0; j < 8; ++j) {
      int tmp = b_buf[j].read();
      b[i * 8 + j] = tmp + 2;
    }
  }
```

Pragma HLS Protocol

- Defines a **protocol region** where **no clock operations** are inserted by Vitis HLS unless explicitly coded
- Ensures no clocks between operations, including reads/writes to function arguments
- Maintains the exact order of reads and writes in the synthesized RTL
- A protocol region is defined using **braces {}** and a name
- Ensures that **reads and writes happen in the exact order** written in the C/C++ code

#pragma HLS protocol [floating | fixed]

Pragma HLS Protocol

#pragma HLS protocol [floating | fixed]

Floating: Lets code statements outside the protocol region overlap and execute in parallel with statements in the protocol region in the final RTL. The protocol region remains cycle accurate, but outside operations can occur at the same time. This is the default mode

Fixed: The fixed mode ensures that statements outside the protocol region do not execute in parallel with the protocol region.

Protocol: Example

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

void stream_modifier(hls::stream<ap_uint<32>> &in_stream,
                     hls::stream<ap_uint<32>> &out_stream) {
    #pragma HLS interface axis port=in_stream
    #pragma HLS interface axis port=out_stream
    #pragma HLS interface ap_ctrl_none port=return

    {
        #pragma HLS protocol fixed  // ensures tight control over read-modify-write

        ap_uint<32> data_in = in_stream.read();  // AXI handshaking happens here
        ap_uint<32> data_out = data_in + 1;
        out_stream.write(data_out);              // AXI handshaking happens here
    }
}
```

Protocol: when to use

- **Cycle-accurate control** over I/O or memory accesses
  - Allows to define exact timing and handshaking signals for function interface – crucial when interacting with external hardware or memory interfaces with strict timing requirement
- Custom protocols (e.g., GPIO, SPI-like behavior)
  - Interfacing with custom hardware using protocols not directly supported by standard HLS interfaces
  - Define sequence of reads, writes and control signal transitions
- **Low-latency pipelines** with precise read/write sequencing
- To override default behavior where HLS inserts waits or splits across cycles

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