Introduction to HLS

Vitis HLS documentation:
https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Introduction

Code for this tutorial:
https://github.com/gpetruc/GlobalCorrelator_HLS/tree/tutorial-2023

What is HLS

- HLS is a compiler from C++ to FPGA firmware
  - Configurations and #pragma directives in the code tell the compiler what to do
- To get efficient FPGA algorithms, you need to write C++ code with FPGA & HLS in mind

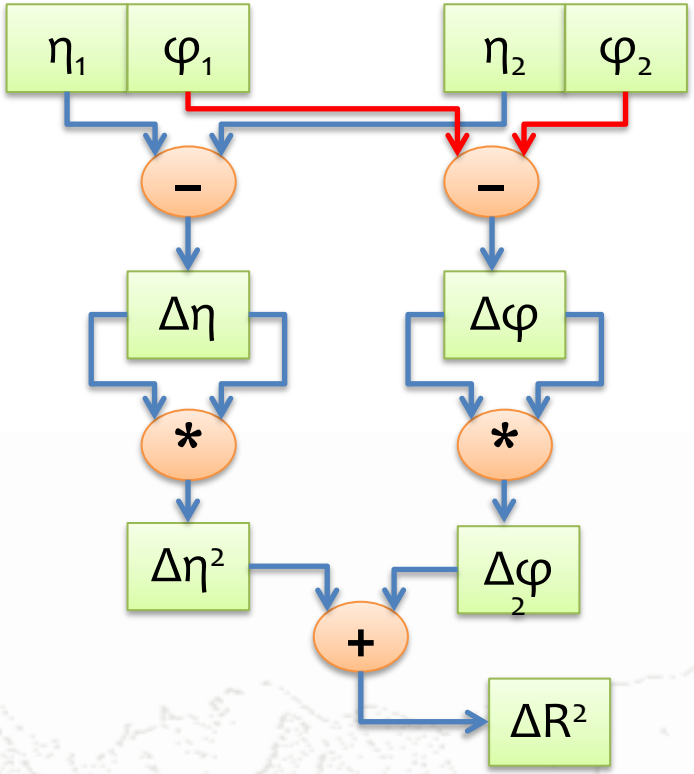
PART I: FPGA BASICS

disclaimer: this will be an oversimplified description

CPU vs FPGA

- An FPGA is an array of configurable logic components and interconnects to route signals across components
  - each component can usually only do simple operations, but they all run in parallel
- Different programming model wrt CPU / GPU
  - A CPU program is a sequence of operations to be executed, i.e. the program extends in time (it's stored in memory).
  - In an FPGA a program defines what fixed operation each component does (at all times), i.e. it extends in space in the FPGA.

Example: computing ΔR^2



Pipelining: the steps of the data processing are done by different components, so a new set of inputs can start to be processed while the older inputs are still being processed.

Characterizing an FPGA algorithm

- A FPGA algorithm (“IP Core”) can be characterized by:
  1. inputs and outputs
  2. clock frequency at which it runs
  3. initialization interval (II): the time after which the algorithm can take in a new set of inputs to process
  4. latency, i.e. the time delay between the input and the corresponding output
  5. FPGA resources used
- In HLS, you define 1 and request (2, 3).
  HLS tries to implement it, and computes (4, 5)

FPGA resources

- There are four main kinds of FPGA resources
  - Flip-flops
  - LUTs
  - Block RAMs
  - DSPs (Digital Signal Processors)
- HLS normally infers from the C++ code what kind of resource to use for each task
  - In some cases, helped by #pragma statements

FPGA resources: Flip-flops

- FFs, or registers, store values used in the computations
  - each flip flop can store **1 bit** of information
- The input signal is captured by the FF from its input pin at the rising edge of the clock.
- During each clock period, signals goes from one or more FF output pins to input pins of the next FFs

FPGA resources: LUTs

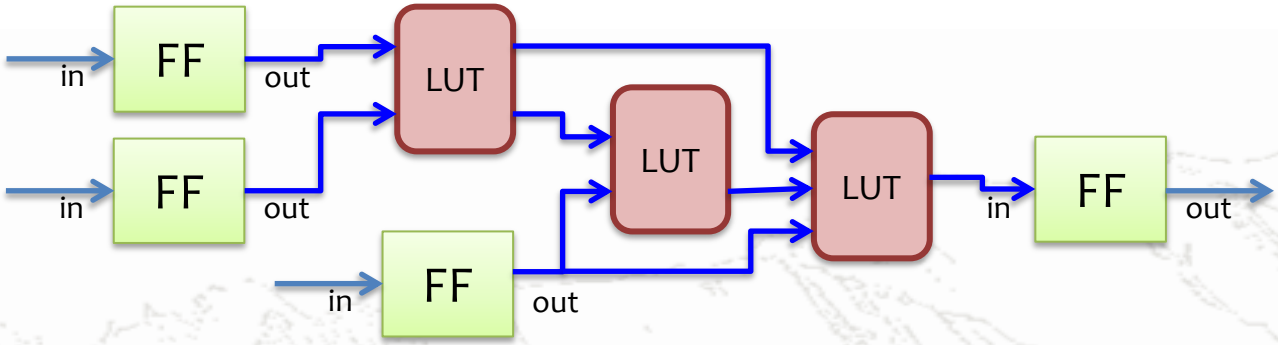
- LUTs are simple components that can compute any **logic function** of 5-6 bits
  - can be combined to get any logic function
- Typical applications of LUTs:
  - Logic gates: `AND`, `OR`, `NOT`, `XOR`, etc.
  - Additions, subtractions, comparisons
  - Selection, e.g. `( a ? b : c )`
  - Can also be used for multiplications when small numbers are involved, or when multiplying by constant factors

LUTs and clock

- LUTs are independent from the clock:
  - a signal can pass through many LUTs in one clock cycle, as long as the total time taken by LUTs and routing is below the clock period

$$\max\left(\Sigma\, t_{LUT} + t_{route}\right) \leq t_{clock} - \Delta t_{clock}$$

"clock uncertainty"



FPGA resources: BRAMs

- Block RAMs can store more data than FFs
  - On Xilinx FPGAs, a single block has a capacity of **18k bits**
  - Multiple blocks can be connected to implement larger memories
  - UltraScale+ & Versal devices have also bigger 288k bit UltraRAMs
- Data can be accessed through **2 ports**.
  - At **each clock cycle**, each port can read and/or write a word in the memory at a given address
  - The word size can be configured **to any of 1, 2, 4, 8, 9, or 18 bits**, independently for each port. If one port is used only to read and one only two write, then both can also be 36 bits wide

BRAMs

- BRAMs are good for storing large amount of data that does not need a fast **throughput** in and out of the ram
  - e.g. buffer objects to be later processed sequentially, in any order
- Can also be used for ROM lookup tables, e.g. calibrations or to **tabulate functions** too complex or slow to evaluate
  - HLS allows to define C++ code to initialize the memories
- The ports of BRAM can also operate with separate clocks, effectively transferring data across clock domains
  - However, HLS only works with a single clock

FPGA resources: DSP

- Digital Signal Processors are specialized components that can do maths
- The main use case is integer multiplications:
  - One Xilinx DSP can compute $A_{27} \times B_{18}$, the product of **one 27-bit integer times a 18-bit integer** ($A_{27} \times B_{24}$ on newer Versal FPGAs)
  - DSPs can be combined for bigger multiplications (usually not needed for the L1 trigger precision)
- DSPs have a latency of **up to 4 clock cycles**
  - But are **pipelined with II=1**, i.e. they can take one new input at each clock cycle.

Computation

have you ever programmed for an old 8086 CPU?

- Xilinx FPGAs work well with integers
  - floating point operations are slow & expensive
- Bitwise operations, additions & comparisons are rather cheap & fast (bit shifts are free)
- Multiplications are more expensive
- Divisions and any other mathematical functions are very slow & very expensive
  - often best to implement as ROM lookup tables

HLS: LANGUAGE BASICS

C++ and HLS

A subset of the C++ language can be used efficiently:

- Simple types: custom library for fixed precision integers of any size (floats supported but discouraged)
- Structured data: structs and fixed-size arrays
  - No dynamic memory allocation (e.g. no std::vector, …)
- If blocks, for loops, functions, classes, templates
  - Parameter passing by value, pointers or C++ reference
- Top-level code must be a function
  - Can use function-local static variables for stateful functions

Data types: arbitrary integers

ap_[u]int<N> class: N-bit [unsigned] integers

- support all math operators, and additional bit manipulation
  - bit access var[bit], slicing var(hi-bit,lo-bit), concatenation, reduction
- especially important on input or output
  - Vitis can infer the expected bit range for the outcome of mathematical operations (e.g. the sum of two N-bit numbers is a N+1 bit number)
- Integers can be used for much of the math:
  - to store physics quantities into integers, decide what is your least significant bit, and multiply a large constant, i.e. $x_{int} = \text{int}(x_{flt} \cdot (1/\text{LSB}))$
  - drop some least-significant bits after multiplications ($c = (a \times b) >> n$), to avoid too large integers
  - be mindful of overflows and wrap-around

Data types: fixed precision

- ap_[u]fixed<W,I*[,opts]*>: fixed-point values
  - it's basically an ap_[u]int<W> implicitly multiplied by $2^{-(W-I)}$
  - optionally it can have better overflows & rounding, e.g. ap_ufixed<14, 12, AP_RND_CONV, AP_SAT> : saturates instead of wrap-around for overflows, and rounds instead of truncating (for a slight increase in FPGA logic resources)
- Useful when your have numbers with different precisions or multiplications, since the compiler takes care of bit-shifts
  - E.g. used extensively for NNs in HLS4ML

Data types: structs & arrays

- Arrays can be implemented as BRAM, FIFO queues or just registers (FFs) for each element
  - which choice is better depends on needed access pattern, can be specified via #pragma's
  - for not too large arrays, best latency & throughput usually achieved by fully splitting into FFs with this directive

#pragma HLS array_partition variable=<*name*> complete

- structs or simple classes are supported
  - array of structs may be implemented by HLS as also as structure of arrays, if not fully split (#pragmas can be used to configure it)

“if” block, “for” loops

- **if**: most likely, HLS will have to instantiate FPGA resources to evaluate both branches of the block:
  - resources will be sum of the two branches
  - latency will be the worst of the two branches, plus some more
- **for**: can be sequential or parallel (**unrolled**); unrolling possible under some conditions:
  - all inputs must be readable simultaneously (e.g. they must be in FFs, not in a BRAM)
  - no complex inter-dependencies across iterations (but e.g. `for (i) { sum += a[i] * b[i]; }` may be ok)
  - Unrolling can be requested explicitly (`#pragma HLS unroll`) or inferred by the HLS tool

Functions

- Functions can be used, with inputs & outputs by value or reference (or pointer)
  - the top-level entity to be compiled to firmware must be a function
- The requested pipelining can be specified with

  #pragma HLS pipeline II=<N>
- Functions can be inlined to allow more scheduling optimization (#pragma HLS inline)
  - the price is longer compilation and less clear accounting of resources and latency

HLS: TOOLS

Disclaimer: something may be a bit obsolete as the new Vitis 2023.2 introduced a new GUI and integrated Vitis+Vivado IDE that I couldn't test yet.

An HLS project

An HLS project contains:

- C++ sources for the function to synthetize
- C++ source files for the testbench:
  - an executable that calls the function to be synthetized, and validates the output
- Configuration for the project:
  - Name of the function to synthetize
  - Target FPGA model and clock speed
- A project can be created from a TCL script
  - easier to store in git than the project itself

Vitis HLS: steps

There's four steps that Vitis HLS can run:

- C/C++ simulation: compile the code to be synthetized and the test bench, and run it
  - purpose: check you didn't introduce bugs when optimizing the C++ for synthesis
- C/C++ synthesis: compile C++ to firmware
  - this is the most important step
- C/RTL co-simulation: run side-by-side the C++ and the firmware
  - May be useful e.g. for complex algorithms with an internal state
- Running Implementation (RTL Synthesis, Place & Route)
  - Provides a more accurate estimate of resource usage and of whether the RTL will meet timing

Running Vitis

- Run Vivado in batch from a TCL file:

  `vitis_hls -f <script.tcl>`

  – to create the project from a tcl file

  – to run repetitive tasks

- Open a project in the Vitis HLS GUI

  `vitis_hls -p <project_dir>`

  – to browse detailed reports

  – for a faster cycle of edit / run sim / run synthesis

Minimal Vitis TCL project

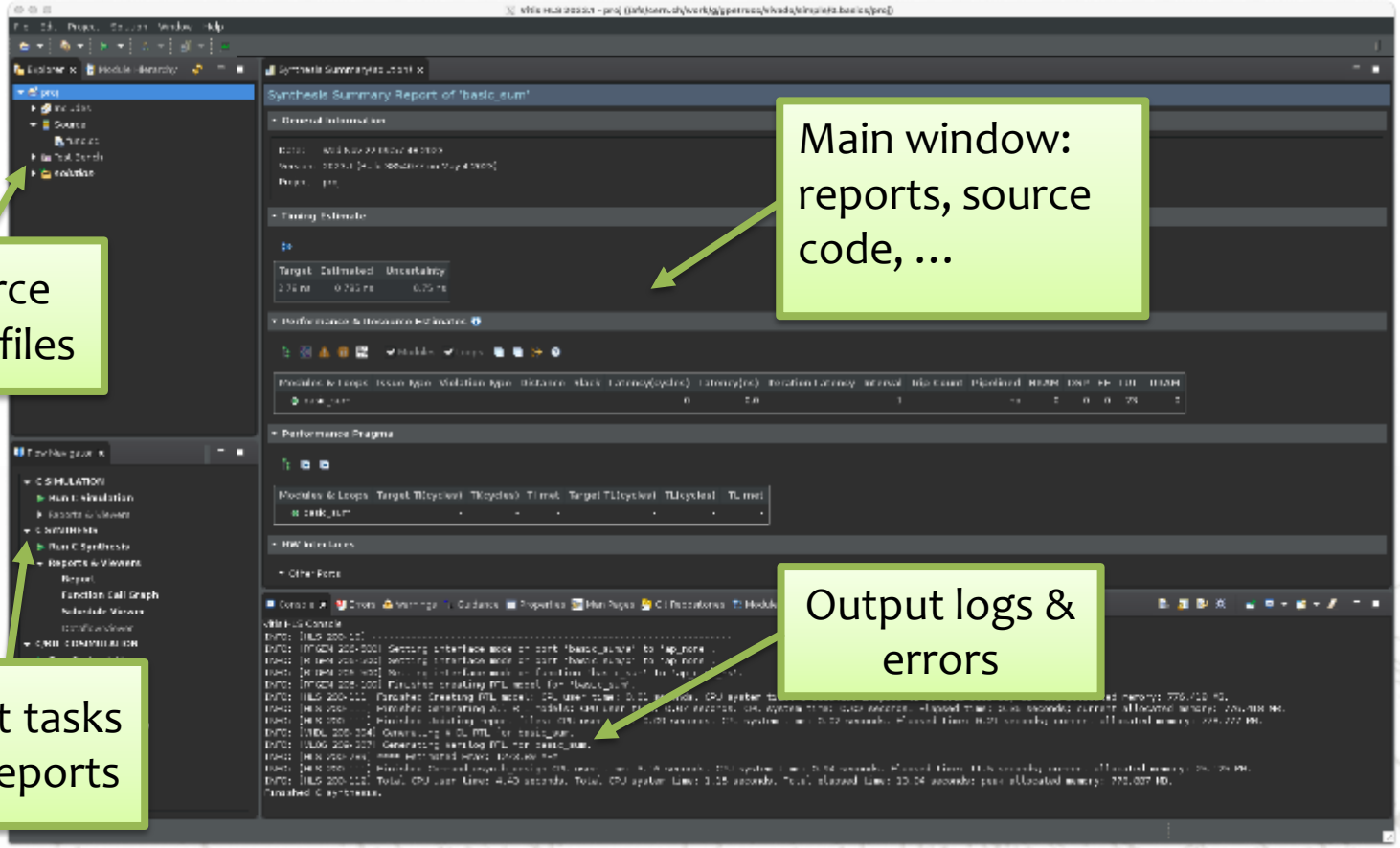
```
open_project -reset "proj"
set_top myfunc
add_files src/func.cc
add_files -tb testbench.cc

open_solution -reset "solution"
set_part {xcvu13p-flga2577-2-e}
create_clock -period 2.777

#csim_design
#csynth_design
exit
```

- create a project
- specify the function to synthetize
- source code for synthesis
- source code for the testbench

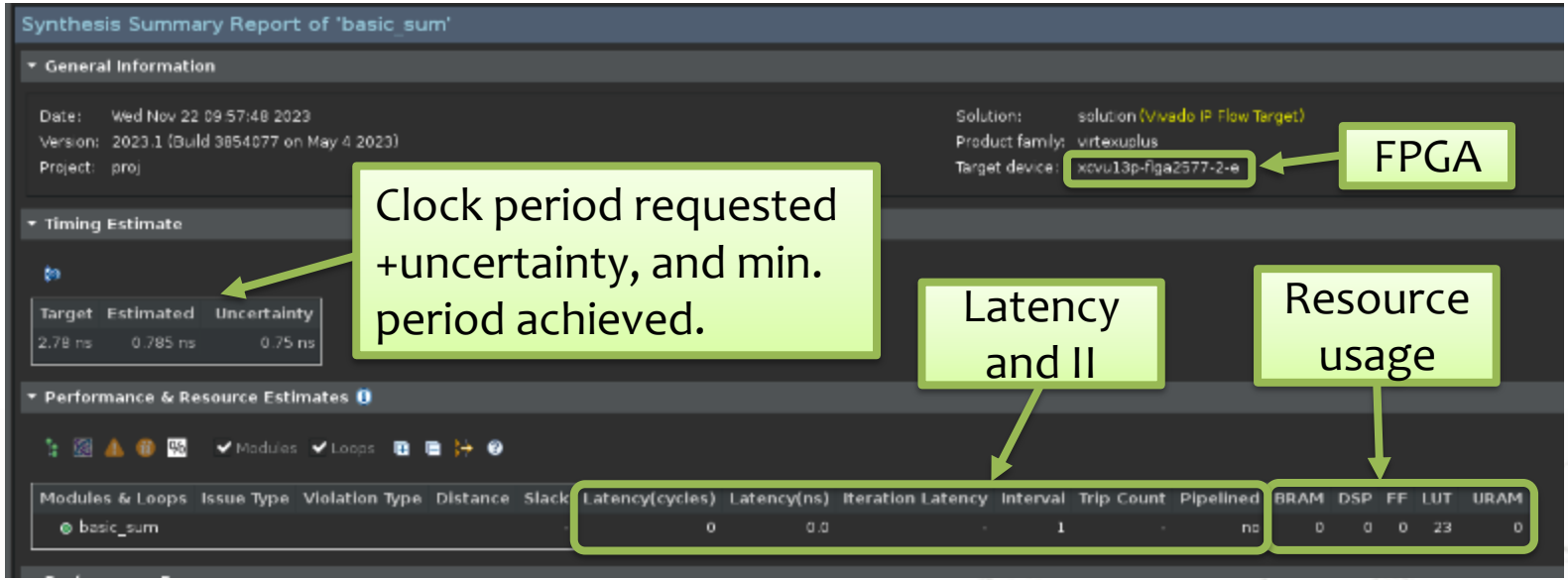
- create a solution, i.e. a hardware configuration for synthesis: FPGA (VU13P), clock (360 MHz)

- Run C-simulation
- Run Synthesis
- Exit from the TCL prompt

Vitis GUI


Synthesis Summary Report of 'basic_sum'
Main window: reports, source code, …
Source code files
Output logs & errors
Select tasks and reports

Vitis: synthesis report



https://github.com/gpetruc/GlobalCorrelator_HLS/tree/tutorial-2023

HANDS-ON PART

https://github.com/gpetruc/GlobalCorrelator_HLS/tree/tutorial-2023/0.basics

EXAMPLE 0 : VERY BASIC STUFF

Very basic examples

```
ap_int<16> basic_sum(ap_int<16> a,
                     ap_int<16> b) {
    return a+b;
}


ap_int<24> basic_mul(ap_int<16> a,
                     ap_int<10> b) {
    return a*b;
}


ap_int<24> pipeline_mul(ap_int<16> a,
                        ap_int<10> b) {
    #pragma HLS pipeline II=1
    return a*b;
}
```

Latency	II	DSP	FF	LUT
0	1	0	0	23

Estimated min clock period: 0.79ns

Latency	II	DSP	FF	LUT
0	1	1	0	5

Estimated min clock period: 1.94ns

Latency	II	DSP	FF	LUT
0	1	1	0	5

Estimated min clock period: 1.94ns

Parallelization & switches

```
void
sum_and_mul(ap_int<16> a, ap_int<10> b,
     ap_int<16> &sum, ap_int<24> &prod) {
  #pragma HLS pipeline II=1
  sum  = a + b;
  prod = a * b;
}


ap_int<24> sum_or_mul(bool want_sum,
           ap_int<16> a, ap_int<10> b) {
    #pragma HLS pipeline II=1
   return want_sum ?
       ap_int<24>(a + b):
       ap_int<24>(a * b);
}
```

Latency	II	DSP	FF	LUT
0	1	1	0	28

Estimated min clock period: 1.94ns

Latency: worse of sum & product

Resources: sum of sum & product

Latency	II	DSP	FF	LUT
1	1	1	44	55

Estimated min clock period: 1.94ns

More resources & latency

Divide et despera

```
ap_int<16> basic_div(ap_int<16> a,
                     ap_int<10> b) {
    #pragma HLS pipeline II=1
    return a / b;
}
```

Latency	II	DSP	FF	LUT
19	1	0	655	459

Just don't do it

Fixed point and saturation

```
ap_fixed<16,12> fix_sum(
    ap_fixed<16,12> a, ap_fixed<10,6> b)
{
     #pragma HLS pipeline II=1
    return a + b;
}

ap_fixed<16,12,AP_TRN,AP_SAT>
fix_sum_sat(
  ap_fixed<16,12,AP_TRN,AP_SAT> a,
  ap_fixed<10,6,AP_TRN,AP_SAT> b) {
    #pragma HLS pipeline II=1
    return a + b;
}
```

Latency	II	DSP	FF	LUT
0	1	0	0	23

Estimated min clock period: 0.79ns

Same as ap_int<16> + ap_int<10>

Latency	II	DSP	FF	LUT
0	1	0	0	86

Estimated min clock period: 1.03ns

More logic, for the saturation check (in this very simple case, a lot more; usually less so…)

https://github.com/gpetruc/GlobalCorrelator_HLS/tree/tutorial-2023/1.arrays

EXAMPLE 1: ARRAYS

Arrays

```
#define NDATA 12
ap_int<24> mul_add_basic(
      const ap_int<16> a[NDATA],
      const ap_int<16> b[NDATA]) {
  ap_int<24> sum = 0;
  for (int i = 0; i < NDATA; ++i) {
    ap_int<24> prod = (a[i] * b[i]) >> 8;
    sum += prod;
  }
  return sum;
}
```

Latency	II	DSP	FF	LUT
16	17	1	93	120

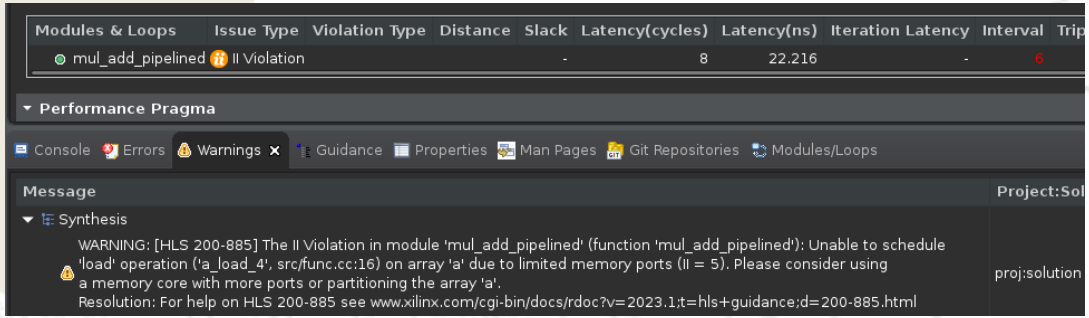
"a" and "b" implemented as memories, and read sequentially

Arrays input/output limitations

```
#define NDATA 12
ap_int<24> mul_add_basic(
      const ap_int<16> a[NDATA],
      const ap_int<16> b[NDATA]) {
  #pragma HLS pipeline II=1
  ap_int<24> sum = 0;
  for (int i = 0; i < NDATA; ++i) {
    ap_int<24> prod = (a[i] * b[i]) >> 8;
    sum += prod;
  }
  return sum;
}
```

Latency	II	DSP	FF	LUT
8	6	12	432	536

"a" and "b" implemented as memories, Vitis needs at least 6 clock cycles to read all the 12 values from the 2 BRAM ports



Arrays partitioning

```
#define NDATA 12
ap_int<24> mul_add_basic(
      const ap_int<16> a[NDATA],
      const ap_int<16> b[NDATA]) {
  #pragma HLS pipeline II=1
  #pragma HLS array_partition variable=a complete
  #pragma HLS array_partition variable=b complete
  ap_int<24> sum = 0;
  for (int i = 0; i < NDATA; ++i) {
    ap_int<24> prod = (a[i] * b[i]) >> 8;
    sum += prod;
  }
  return sum;
}
```

Latency	II	DSP	FF	LUT
2	**1**	12	363	333

"a" and "b" now are each implemented 12 individual integer inputs a_0, a_1, a_2, …

So, they can all be read at once and the function can be pipelined at ii=1

https://github.com/gpetruc/GlobalCorrelator_HLS/tree/tutorial-2023/2.simple_ht

EXAMPLE 2: H_T COMPUTATION

Structures, loops, understanding latency & dependency on clock speed, recusive templates, running implementation

Compute H_T with an $|\eta| < 2.4$ cut

```
typedef ap_uint<14>  pt_t;
     // 1 unit = 0.25 GeV
     // max = 2 TeV
typedef ap_int<10>  etaphi_t;
    // 1 unit = 0.01;
    // max = 5.12

struct Particle {
    pt_t hwPt;
    etaphi_t hwEta;
};

#define NPARTICLES 20

pt_t algo(Particle articles[NPARTICLES]) {

}
```

Define integer datatypes for p_T and η

Define a struct for a Particle

Task: compute sum p_T for particles with $|\eta| < 2.4$

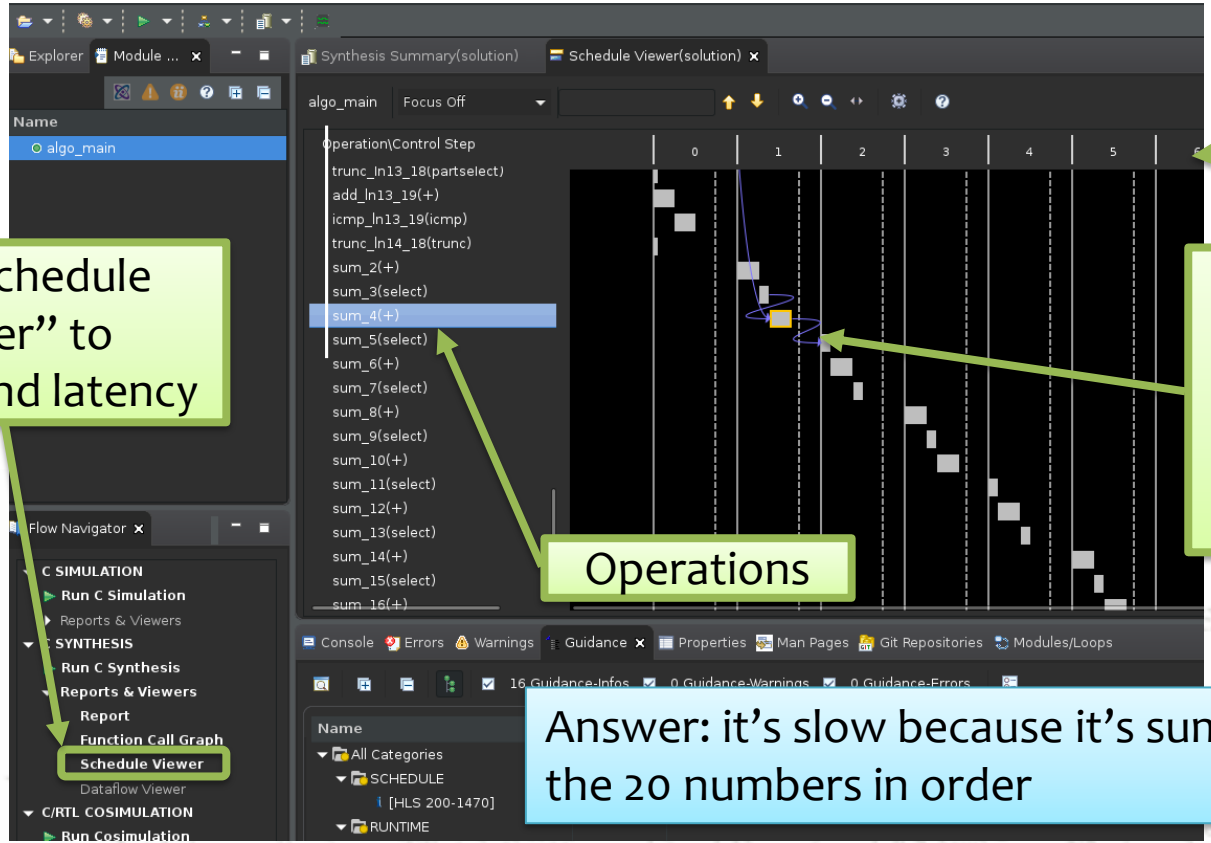
Compute H_T with an $|\eta|$ cut / 1

```
pt_t algo_main(Particle
          particles[NPARTICLES])
{
    #pragma HLS ARRAY_PARTITION \
        variable=particles complete
    #pragma HLS pipeline II=1
    pt_t sum = 0;
    for (unsigned int i = 0;
         i < NPARTICLES; ++i) {
       if (-240 <= particles[i].hwEta &&
            particles[i].hwEta <= 240) {
         sum += particles[i].hwPt;
       }
    }
    return sum;
}
```

Latency	II	DSP	FF	LUT
13	1	0	2629	2385

Why is it taking so much time just to add up 20 numbers?

Schedule Viewer



Clock cycles

Use “Schedule Viewer” to understand latency

Time for that operation, and scheduling dependencies (arrows)

Operations

Answer: it’s slow because it’s summing all the 20 numbers in order

Rewritten loop

```
pt_t algo_main(Particle
          particles[NPARTICLES]) {
    […]
    pt_t sum = 0;
    for (unsigned int i = 0;
         i < NPARTICLES; ++i) {
       bool central =
          (-240 <= particles[i].hwEta &&
           particles[i].hwEta <= 240);
       sum += (central ?
               particles[i].hwPt :
               pt_t(0));
    }
    return sum;
}
```

Latency	II	DSP	FF	LUT
2	1	0	375	1271

In this case, rewriting the loop so that the sum is always performed allows Vitis HLS to understand that the order doesn't matter, and the sum can be done more efficiently

Recursive template version

```
// process each half and combine
template<unsigned int N>
pt_t partial_ht(const Particle particles[N]) {
   return partial_ht<N/2>(particles) +
          partial_ht<N-N/2>(&particles[N/2]);

}

// tail case: a single item
template<>
pt_t partial_ht<1>(
    const Particle particles[1]) {
    if (-240 <= particles[0].hwEta &&
particles[0].hwEta <= 240) {
        return particles[0].hwPt;
    } else {
        return pt_t(0);
    }
}
```

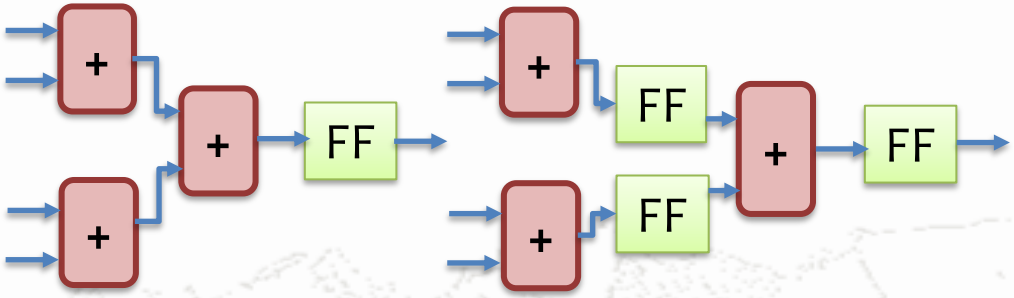
Latency	II	DSP	FF	LUT
2	1	0	375	1271

In this approach we explicitly tell HLS to compute the H_T from the two halves of the particle array and combine them

This approach can be used for many other tasks, e.g. selection, sorting …

Changing clock frequency

Clock (MHz)	Latency		FFs	LUTs
240 MHz	1 clk	4.2 ns	156	1271
360 MHz	2 clk	5.6 ns	375	1271
480 MHz	3 clk	6.2 ns	580	1271



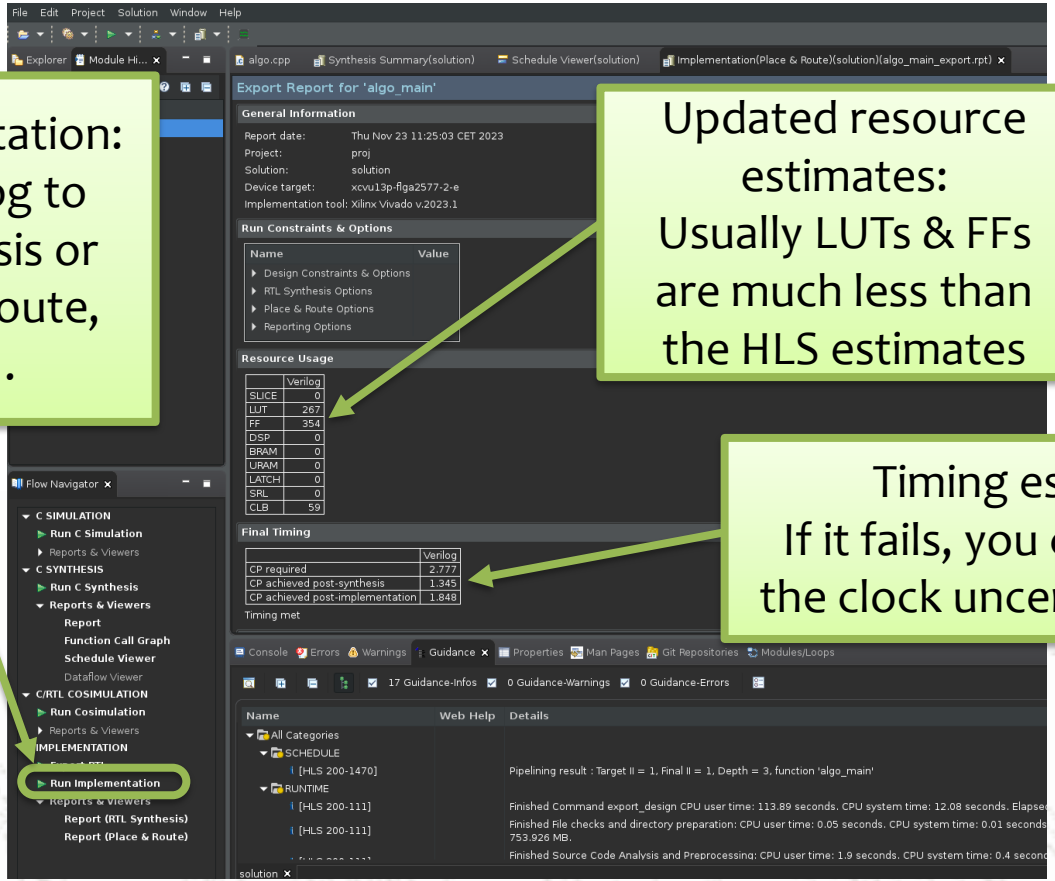
- Same number of computations in any case: same number of LUTs
- Higher frequency, less time per clock cycle, can do less operations per clock, need more clock cycles
  - Latency in ns is more similar, but higher for faster clocks (need to leave more margin)
  - Higher usage of FFs (and of FPGA interconnect, but it's not reported by HLS)
  - Higher throughput

Running implementation

Run implementation: Opens a dialog to select Synthesis or also Place + Route, options, …

Updated resource estimates: Usually LUTs & FFs are much less than the HLS estimates

Timing estimate. If it fails, you can increase the clock uncertainty in HLS



https://github.com/gpetruc/GlobalCorrelator_HLS/tree/tutorial-2023/3.lookup_table

EXAMPLE 3: E_T^{MISS} USING LOOKUP-TABLES

Using lookup tables to speed-up complex functions (e.g. sin, cos, sqrt),

https://github.com/gpetruc/GlobalCorrelator_HLS/tree/tutorial-2023/4.stateful

EXAMPLE 4: A STATEFUL ALGORITHM

An algorithm to deserialize and unpack muons

https://github.com/gpetruc/GlobalCorrelator_HLS/tree/tutorial-2023/a0.alveo_50_basics

EXAMPLE A0: RUNNING ON ALVEO U50

EXERCISES