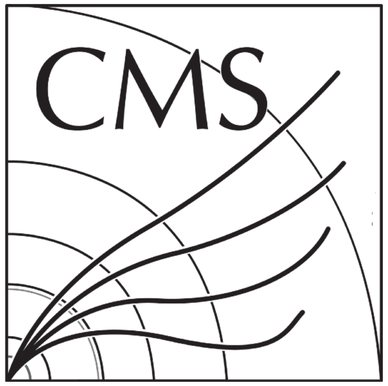

CERN


CMS

Introduction

- High Level Synthesis is a new paradigm in programming FPGAs
  - Write algorithms → synthesis tool determines the hardware
  - Input using C++ — higher level abstraction than HDLs
  - Productivity 📈 — create working designs faster
  - Sophistication 📈 — create advanced designs with complicated algorithms
- But working with HLS still requires expertise, and a foundation in HDL is a great starting point
- Main topics of this part:
  - Number representations and arithmetic
  - Loop Analysis and Optimization

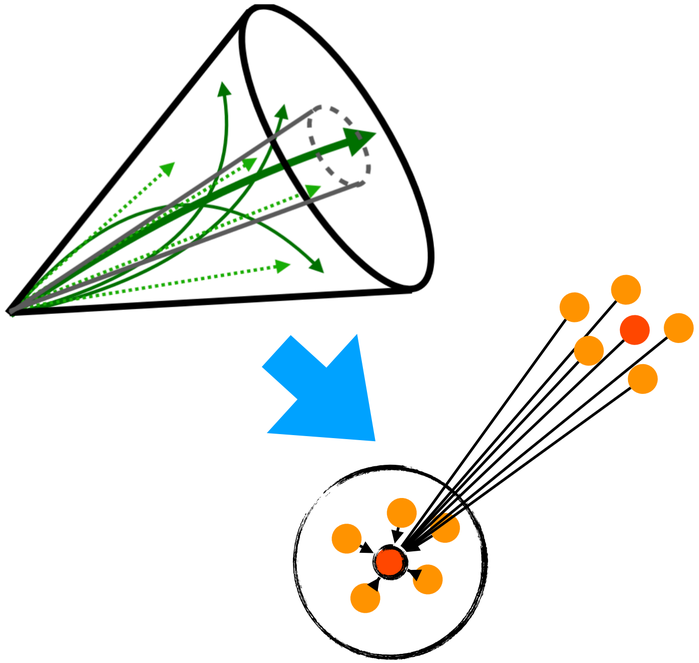
About me

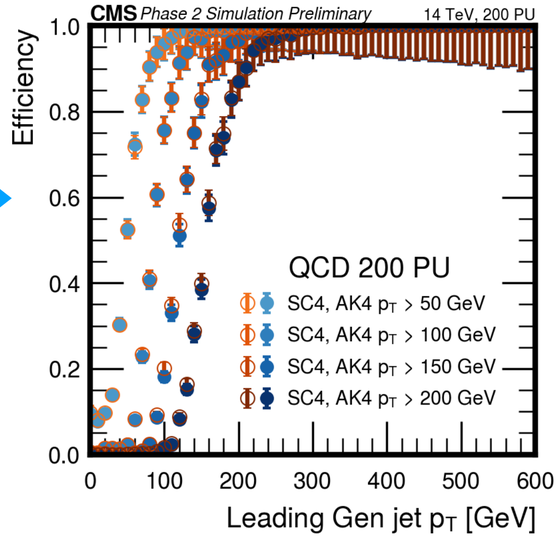
- Staff at CERN working on Level 1 Trigger Upgrade for CMS experiment
  - Mostly designing and implementing detector reconstruction algorithms for Level 1 Trigger
  - Track reconstruction, vertexing, particle flow, jets, jet tagging, ML
  - Task leader in Next Generation Triggers project
- PhD High Energy Physics Imperial College London
  - Thesis: “Applications of FPGAs to triggering in particle physics”
  - Designing physics algorithms with high level languages for FPGAs
- Also deploying Machine Learning into FPGAs for low latency
  - **hls4ml** coordinator 2020-2022, creator and maintainer of **conifer**
- Leading Edge SpAIce project at CERN: ML in FPGA for satellites

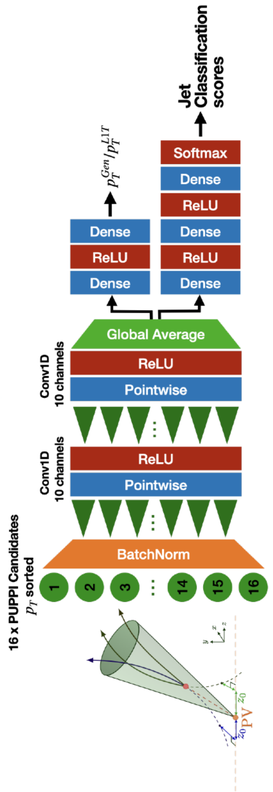
sioni@cern.ch
sioni.web.cern.ch
@thesps
@ssummers

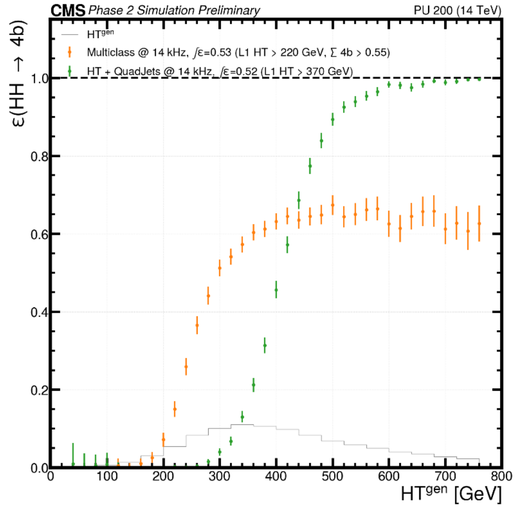
One FPGA Case Study

- For the CMS Phase 2 Upgrade we are designing jet reconstruction and tagging
  - Find cones of particles from the decay of a parent particle, and use ML to predict the parent particle type, make CMS will collect data for important event types (e.g. HH→bbbb)
- Less than 1 μs from particles input to tagged jets output
- We used both HLS and VHDL, and **hls4ml** for the Neural Network

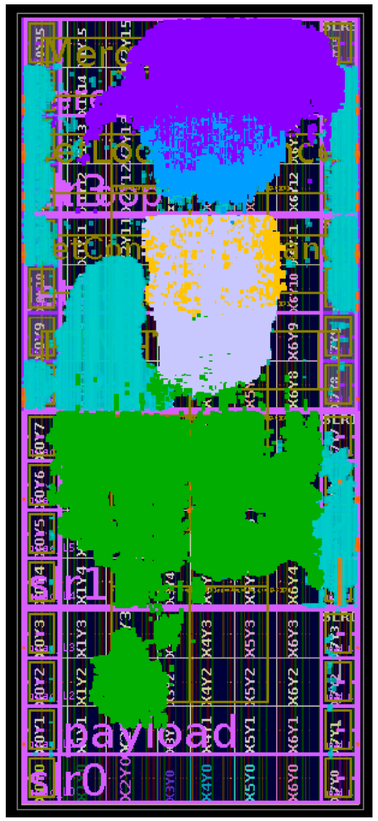






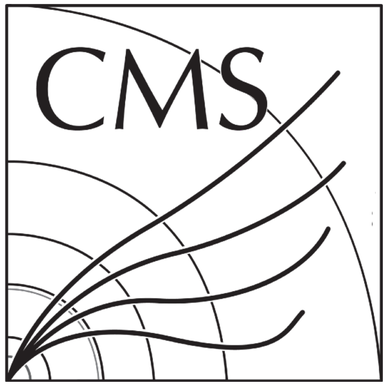


Particle receiving
Jet finding
Jet tagging (NN)



Part 1

Numerics: Fixed Point Arithmetic



- On your CPU (and GPU) you are probably familiar with integer and floating point numerical formats
- Integer represents integer values in a fixed number of bits (e.g. 32, 16, 8)
- Floating point represents values with a mantissa and exponent : $m \cdot 2^e$
  - It's like scientific notation with binary
- Simplified examples, representing the number 113, ignoring sign (positive values only)

Base 10 integer
113_{10}

10^2	10^1	10^0
1	1	3

Base 10 floating point
(3 digit mantissa,
2 digit exponent)
$1.13 \times 10^{2.0}$

10^0	10^{-1}	10^{-2}		10^0	10^{-1}
1 .	1	3	· 10^	2 .	0

Base 2 integer
0b01110001

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	1	0	0	0	1

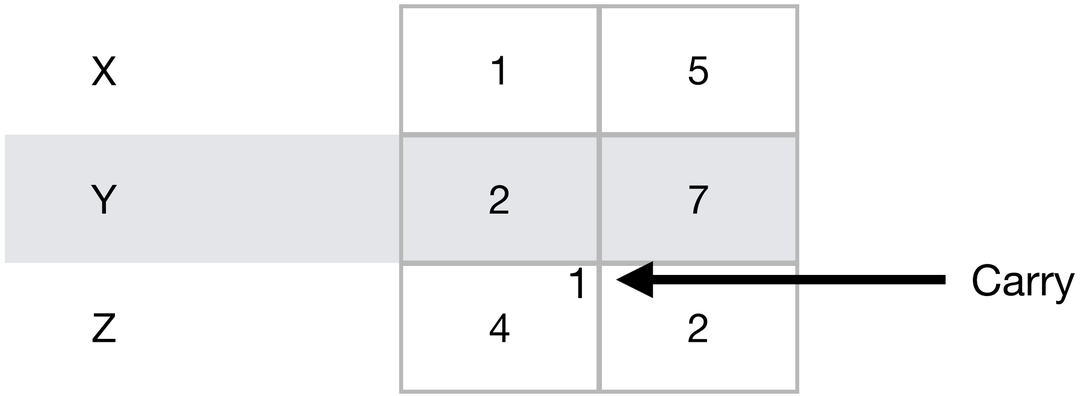
Base 2 floating point
(4 bit mantissa,
4 bit exponent)
$14 \times 2^8 = 112$
(Approximate)

2^3	2^2	2^1	2^0		2^3	2^2	2^1	2^0
1	1	1	0	· 2^	1	0	0	0

Operations

- On your CPU (and GPU) you are probably familiar with integer and floating point numerical formats
- Integer represents integer values in a fixed number of bits (e.g. 32, 16, 8)
- Floating point represents values with a mantissa and exponent : $m \cdot 2^e$

- Demo:
  - Suppose we have decimal integers `x = 15` and `y = 27`
  - Compute `z = x + y` using “long hand”



Integers

- On your CPU (and GPU) you are probably familiar with integer and floating point numerical formats
- Integer represents integer values in a fixed number of bits (e.g. 32, 16, 8)
- Floating point represents values with a mantissa and exponent : $m \cdot 2^e$

- **Exercise**:
  - Suppose we have 4-bit unsigned (positive) integers `x = 0b0011` and `y = 0b0101`
  - Compute `z = x + y` — use the “long hand” method and remember to “carry the 1”

Note: 0b prefix means binary number

Overflow, saturation, truncation

- When working with decimals we usually perform “bit growth” intuitively e.g. `9 + 3 = 12` — one more digit in result
- With computer arithmetic the bit sizes are usually fixed e.g. 4, 8, 16, 32 bits
- When results of operations exceed the constraints of the precision, we can get *overflow*
  - **Exercise**: compute $12_{10} + 5_{10}$ with 4 bit operands and 4 bit result ie ignore anything beyond 4 bits

Bit Growth

- When results of operations exceed the constraints of the precision, we can get *overflow*
- With computer arithmetic the bit sizes are usually fixed e.g. 4, 8, 16, 32 bits
- We can increase the bit precision of the result of an operation in an FPGA to compensate for overflow
  - Provided the result is stored in a different memory/register/logic than the operands of smaller width

- **Exercise**: assuming unsigned integers, what would be the required bit-width for the result of:
  - a 4-bit integer summed with a 4-bit integer?
  - a 4-bit integer summed with a 3-bit integer?
  - a N-bit integer summed with an M-bit integer?
  - a 4-bit integer multiplied with a 4-bit integer?
  - a 4-bit integer multiplied with a 3-bit integer?
  - a N-bit integer multiplied with an M-bit integer?

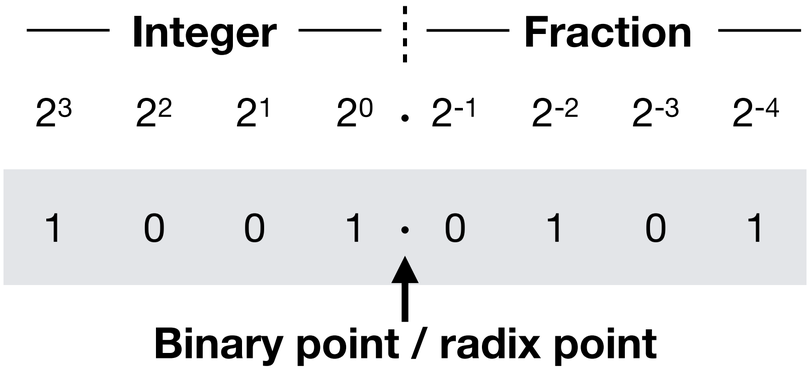
Hint: consider the maximum values of each operand

Two's complement

- So far we used *unsigned* integers, but that's limiting
- How do we represent negative values with fixed size binary numbers?
- Most common method: two's complement
- To work out the two's complement representation of a negative number (e.g. -6_{10} in 4 bits)
  - Start with the binary representation of the *absolute* value: 6_{10} = 0b0110
  - *Invert* all of the bits: 0b0110 → 0b1001
  - Add 1 to the value, *ignoring* overflow: 0b1001 → 0b1010
- Observations:
  - The most significant bit always denotes the sign — leading 0 → positive or zero, leading 1 → negative (but not sign & value)
  - We can do arithmetic with numbers in this representation
  - **Exercise**: take two 4 bit two's complement numbers x = $+3_{10}$, y = -1_{10} and compute x + y in binary

Fixed Point

- On CPU / GPU we need to work with number types that are native to the hardware
  - We can emulate other number types but it arithmetic won't run with high performance
- On FPGA we are designing the hardware itself, so we can use any number representation that we like
- We'll see this for ourselves soon, but integer operations are much less resource and latency intensive than floating point
- But what if we want to represent fractions? Enter fixed point
- Fixed point combines some of the convenience of floating point with the low hardware cost of integers
- Recall: floating point is $m \cdot 2^e \rightarrow$ in fixed point the value of exponent is fixed so it doesn't need to be explicitly represented
- Example: 8 bit unsigned fixed point with 4 integer bits, 4 fractional bits, representing 9.3125_{10}



Fixed Point

- **Exercise**: what are the values in base 10 of these 8 bit unsigned fixed-point numbers?
  - What are the maximum and minimum values of the three fixed-point number formats?

a.

Integer				Fraction			
2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
0	1	1	1	0	0	0	1

b.

Integer						Fraction	
2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}
0	1	1	1	0	0	0	1

c.

Fraction							
2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-11}	2^{-11}
0	1	1	1	0	0	0	1

Example

Integer				Fraction			
2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	0	0	1	0	1	0	1

= 9 .3125$_{10}$

Fixed Point

- **Exercise**: can you generalise the rules for bit-growth of integers to bit-growth of fixed-point?
  - **1)** For addition/subtraction and **2)** multiplication. **Hint**: consider maximum and minimum values
  - Operand 0: F_0 fractional bits, I_0 integer bits (F_0 + I_0 width); Operand 1: F_1 fractional bits, I_1 integer bits (F_1 + I_1 width)
  - Recall the integer bit-growth rules, with operand widths 'N' and 'M'
    - for addition & subtraction the result bitwidth should be 1 + max(N,M)
    - for multiplication the result bitwidth should be N + M

Introduction to High Level Synthesis

- Now we'll get some hands on experience with numerics in High Level Synthesis
- Reminder: why do we use it?
  - Write FPGA designs in C++: lower barrier to entry than HDL
  - Rapid design space exploration (explore resource / latency tradeoffs with ease); realise complex algorithms
- 🛠️ Typical HLS workflow
  - ✏️ Write C/C++ *kernel*: my_function.cpp
  - 🧪 Create *testbench*: my_function_tb.cpp to simulate inputs/outputs
  - ⚙️ Run HLS flow using Vitis HLS:
    - C Simulation: functional correctness check using the testbench — HLS C++ code is executed on the CPU
    - C Synthesis: HLS C++ code is *synthesized* into RTL + resource/latency estimates
    - Co-Simulation: validate generated RTL against testbench — clock cycle accurate
- Optimize latency, throughput, and resource usage (LUTs, FFs, DSPs, BRAM)
  - Explore trade-offs using loop unrolling, pipelining, precision tuning

Introduction to High Level Synthesis

- 📁 Structure of a Typical HLS Project
  - 🔧 my_function.cpp — algorithm to be synthesized to FPGA
  - 📊 my_function_tb.cpp — C++ testbench that drives inputs and checks outputs
  - 📂 hls_prj/ — directory with reports, logs, RTL, etc. produced from the HLS tool
  - 🧾 script.tcl — automates synthesis/verification in batch mode

```
open_project my_proj
add_files my_function.cpp
add_files -tb my_function_tb.cpp
open_solution "solution1"
set_top my_function
create_clock -period 5
csim_design
csynth_design
cosim_design
```

Introduction to High Level Synthesis

- 🔧 my_function.cpp — algorithm to be synthesized to FPGA
  → This is where you describe your logic in standard C++ (with some HLS-specific types)

- 🎯 Define a “top-level” function
  → This is the function that Vitis HLS treats as the hardware module interface
  → Must use scalar or array arguments (no dynamic memory, no STL containers)

- 🧱 Use ap_fixed datatypes for fixed-point arithmetic
  → Provided by the ap_fixed.h header
  → Enables precise control of bit widths for resource and accuracy trade-offs
  - 🔢 **`ap_fixed<16, 5>`**
    → 16-bit signed number: 5 integer bits, 11 fractional bits
  - 🔢 **`ap_fixed<8, 3, AP_RND, AP_SAT>`**
    → 8-bit signed, 3 integer bits, 5 fractional, AP_RND = round to nearest; AP_SAT = saturate on overflow

- 🏗️ Use **`#pragma`** HLS directives to control synthesis behavior
  → Guide unrolling, pipelining, array partitioning, and interface behavior
  → These affect latency, throughput, and resource usage

- 📆 Tomorrow: we'll explore how to use pragmas to tune designs for performance and resource efficiency

Introduction to High Level Synthesis

- Example HLS top-level function, sum two dimension-8 vectors of 16 bit, 8 integer bit values

```
#include “ap_fixed.h”

typedef ap_fixed<16,8> data_t;

void vec_sum(const data_t in1[8], const data_t in2[8], data_t out[8]) {
    VectorLoop:
    for (int i = 0; i < 8; ++i) {
        out[i] = in1[i] + in2[i];
    }
}
```

- When you have all of the ingredients, launch the workflow from the command line with:
  - **`vitis_hls -f csim.tcl ; vitis_hls -f csynth.tcl ; vitis_hls -f cosim.tcl`**
  - This will create a project, run simulation, synthesis, and/or co-simulation as defined in the script

Analyzing Designs

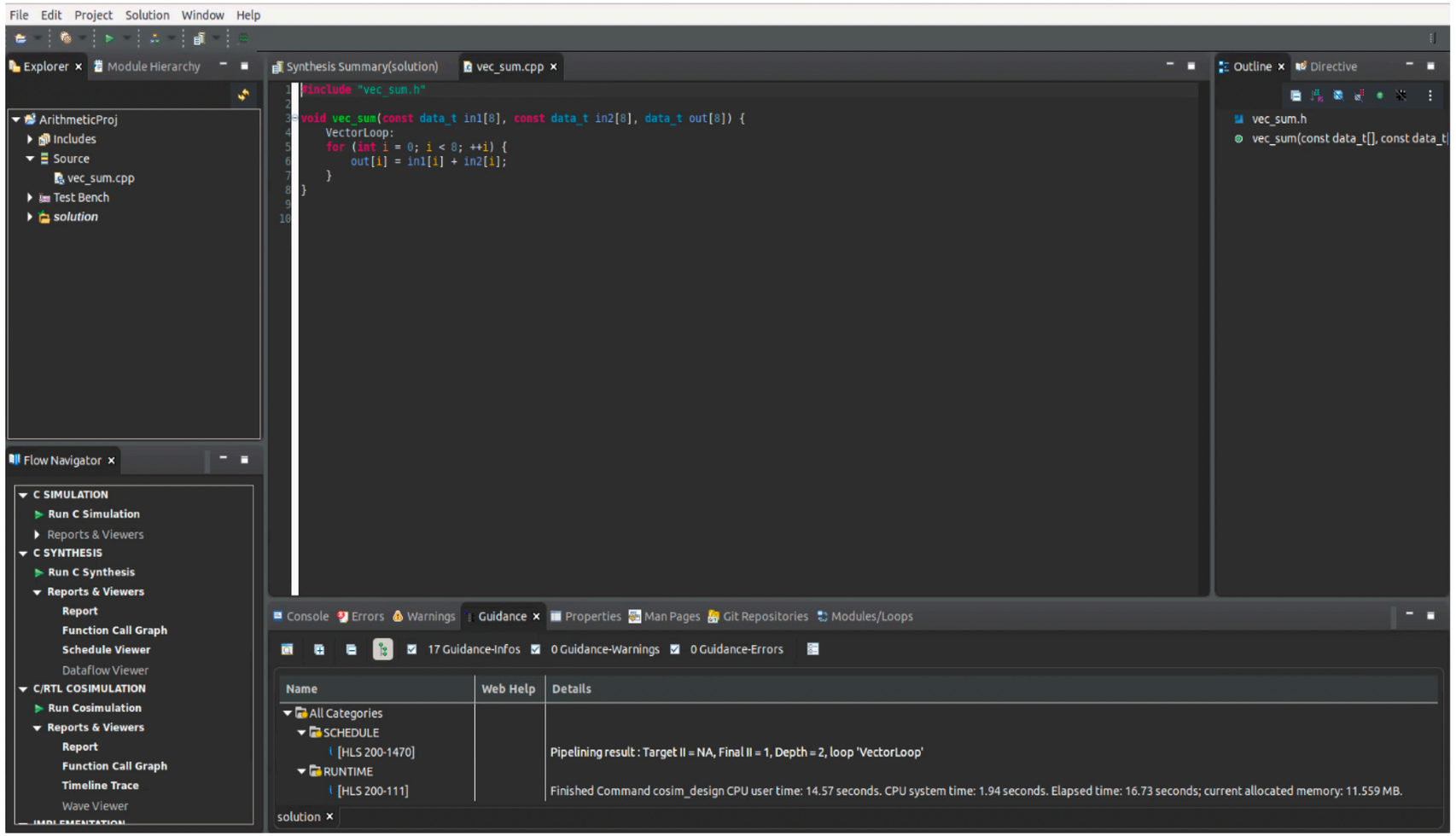
- After we run C Synthesis and Co Simulation, we generate many reports and also an HLS project
  - All of them contain very useful information for analyzing the design
  - In particular:
    - 📄 <project name>/<solution name>/syn/report/csynth.rpt
    - 📄 <project name>/<solution name>/sim/report/<top name>_cosim.rpt
  - These reports can also be viewed in the GUI (next slides)

```
+ Performance & Resource Estimates:

    PS: '+' for module; 'o' for loop; '*' for dataflow
    +---------------+------+------+---------+---------+----------+---------+------+----------+------+----+----------+----------+-----+
    |    Modules    | Issue|      | Latency | Latency | Iteration|         | Trip |          |      |    |          |          |     |
    |    & Loops    | Type | Slack| (cycles)|   (ns)  |  Latency | Interval| Count| Pipelined| BRAM | DSP|    FF    |    LUT   | URAM|
    +---------------+------+------+---------+---------+----------+---------+------+----------+------+----+----------+----------+-----+
    |+ vec_sum      |     -|  0.87|       XX|  XXX.XXX|        XX|       XX|    XX|       y/n|    XX|  XX|        XX|        XX|   XX|
    | o VectorLoop  |     -|  7.30|        X|  XXX.XXX|        XX|       XX|    XX|       y/n|    XX|  XX|        XX|        XX|   XX|
    +---------------+------+------+---------+---------+----------+---------+------+----------+------+----+----------+----------+-----+
```

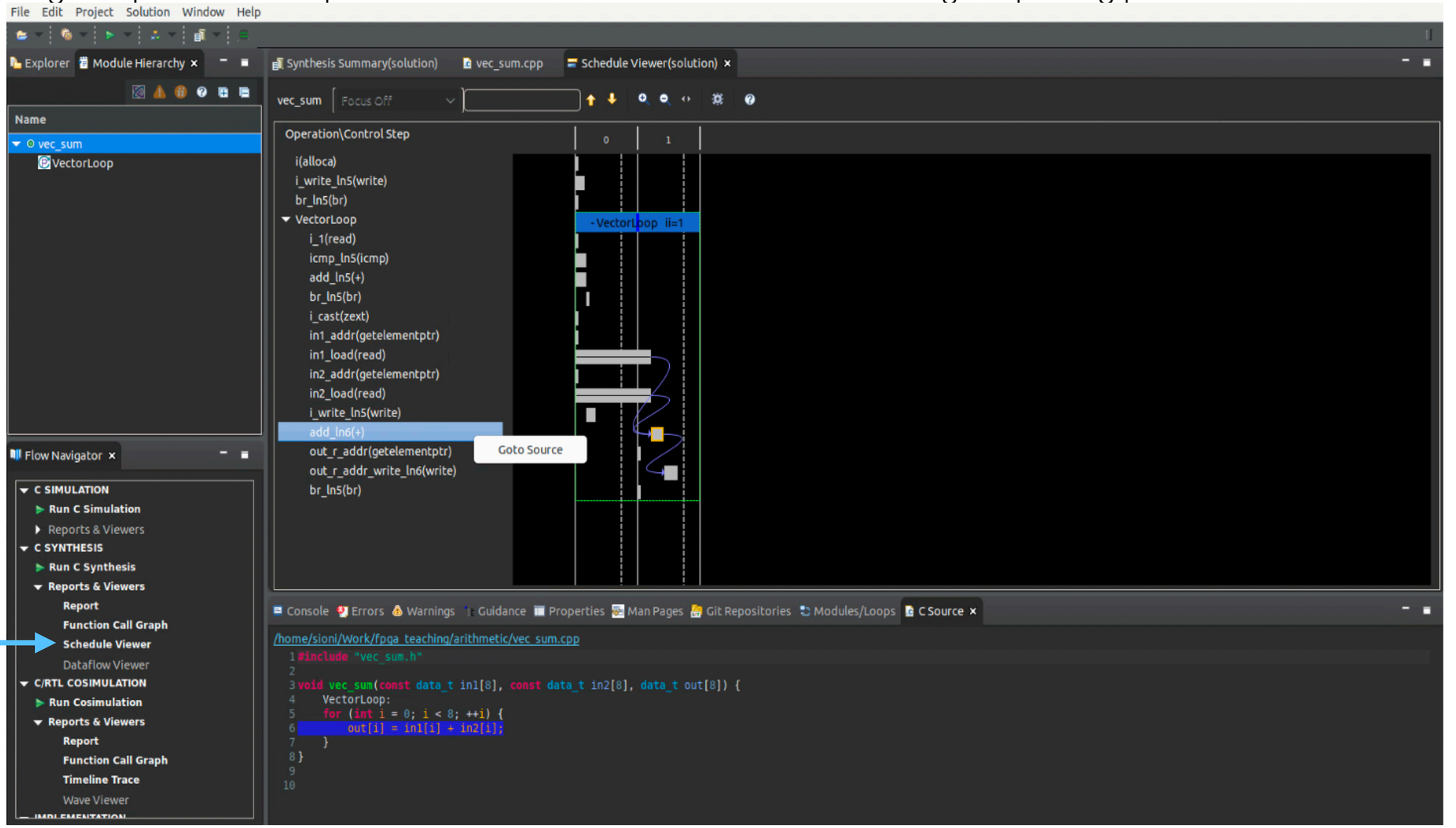
HLS GUI

- We will also use the HLS GUI for some analysis, to open: run **`vitis_hls -classic`** (classic option for ≥ 2023.2)
- You may use whichever file editor you like, but the HLS GUI does provide some useful auto-completion



HLS GUI

- The analysis after running synthesis is especially useful e.g. schedule viewer
  - did the design map to HW as expected? Where are the bottlenecks in the design impacting performance?



Exercise 1

- We'll work with the project that sums together 8-dimension vectors
- 📁 go to directory arithmetic/
- Find all of the necessary files for a first HLS project:
  - 🔧 vec_sum.h & vec_sum.cpp — algorithm to be synthesized to FPGA
  - 📊 testbench.cpp — C++ testbench that drives inputs and checks outputs
  - 🧾 csim.tcl, csynth.tcl, cosim.tcl— automates synthesis/ verification in batch mode
- **Exercise**: run the scripts, and browse the reports and Vitis HLS GUI
  - What is the latency and resource usage of this design?

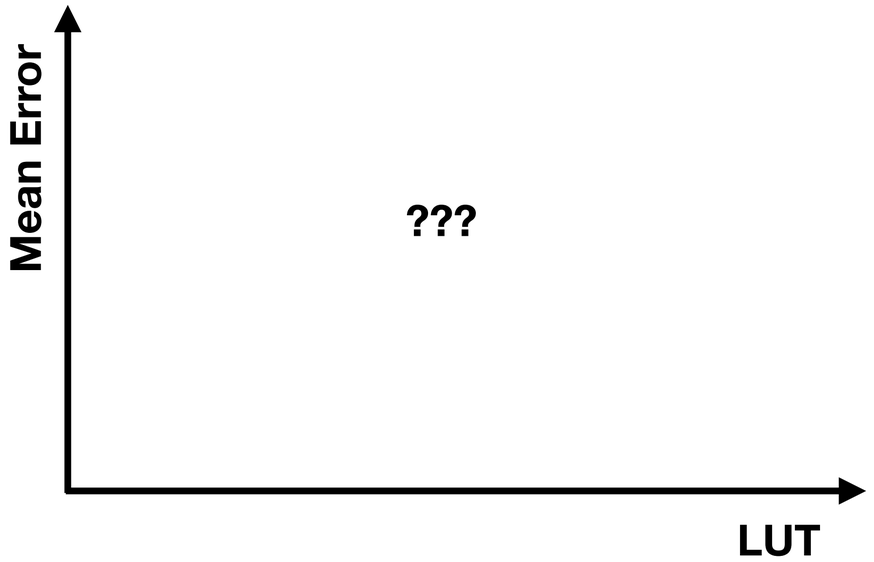
```
void vec_sum(const data_t in1[8],
             const data_t in2[8],
             data_t out[8]) {
    for (int i = 0; i < 8; ++i) {
        out[i] = in1[i] + in2[i];
    }
}
```

DSP	LUT	FF	BRAM	Latency	Worst Error	Mean Error
?	?	?	?	?	?	?

Exercise 2 & 3

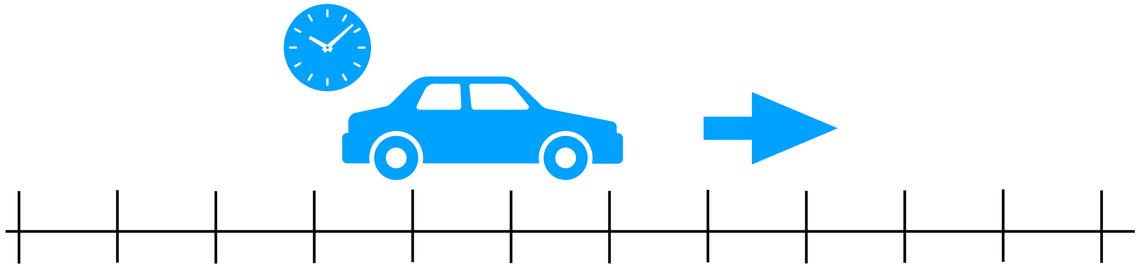
- vec_sum.h has a line:
  - typedef float data_t; that defines that we used floating point for all of the variables
- **Exercise**: adapt the function to use fixed point instead
  - Find the necessary number of integer bits to avoid overflow
  - Find the smallest total bit width that keeps the worst error smaller than 0.1
  - Track your experiments in a results table, and plot LUT usage vs error
  - **Note** running the script overwrites the existing project, so log your results after each run

	DSP	LUT	FF	BRAM	Latency	Worst Error	Mean Error
Width 1	?	?	?	?	?	?	?
Width 2	?	?	?	?	?	?	?
Width 3	?	?	?	?	?	?	?
…	…	…	…	…	…	…	…



Fixed Point

- The rules for bit-growth are important for understanding the types you give to variables in HLS
  - But it's not the only factor!
- If you have a long sequence of arithmetic operations the bit-growth can result in very wide data types
  - Consider that one DSP has one 25 and one 18 bit input
- It can be useful to use some domain knowledge about the realistic values to constrain the types
  - i.e. grow then shrink



- Example: d = v · t distance: meters; time: seconds; velocity: meters per second
  - For a car, suppose v ∈ [0, 50] m/s (50 m/s ≃ 180 km/h) ; t ∈ [0, 20] s sampling device
  - Then v → `ap_ufixed<10,6>` and t → `ap_ufixed<10,6>` — values up to 63, steps of 1/16 ~ 0.06
  - Bitgrowth multiplication rule yields `ap_ufixed<20, 12>` — values up to 4095 m, steps of 1/256 ~ 0.004
  - But given our constraints the real maximum value could be 1000 m, and 0.004 m is too precise
  - Could clip to `ap_fixed<14, 10, AP_RND, AP_SAT>` for our real range and a reasonable precision, with rounding and saturation for safety — values up to 1023 m, steps of 1/16 ~ 0.06 m

Long Exercise 1: Missing Transverse Energy

- Now that you've learned the essentials of fixed point arithmetic, it's time to put it to practice with an extended exercise

- One important quantity that we compute in FPGAs in the CMS Level 1 Trigger system is Missing Transverse Energy (MET)

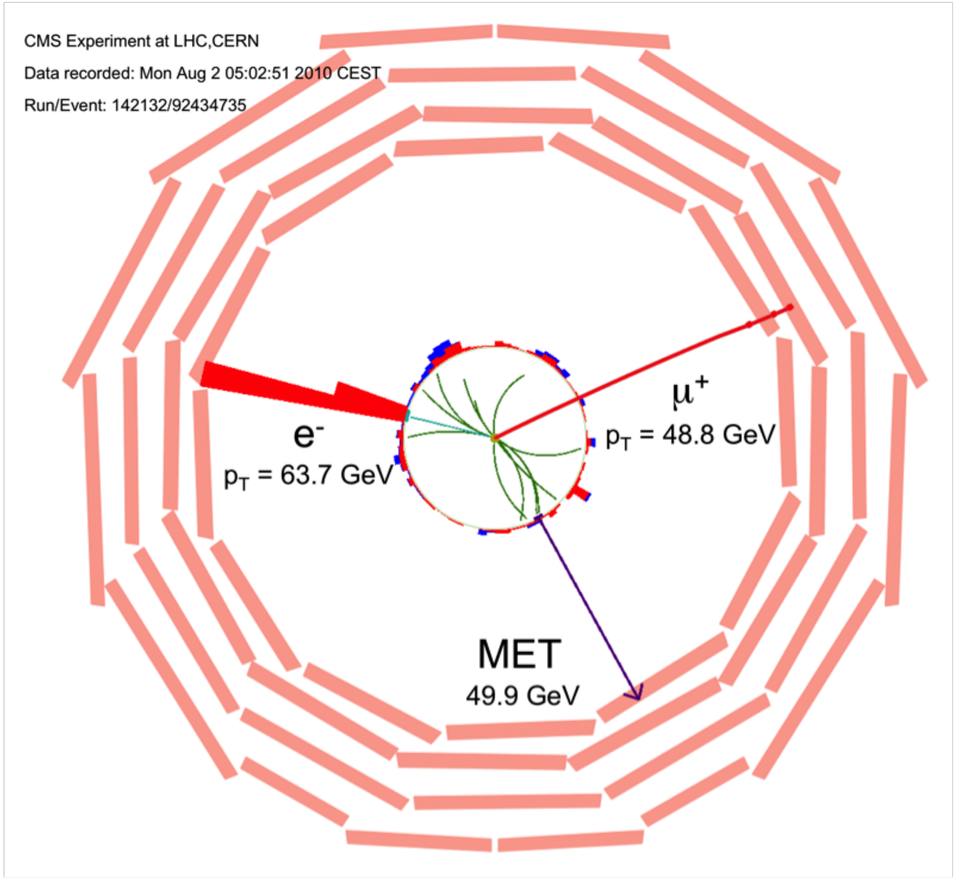
  - Due to momentum conservation, the vector sum over particle momenta must be zero*

  - If we find a significantly non-zero MET it's of interest

    - It could be mismeasurement, or a known particle that doesn't interact with the detector e.g. neutrino

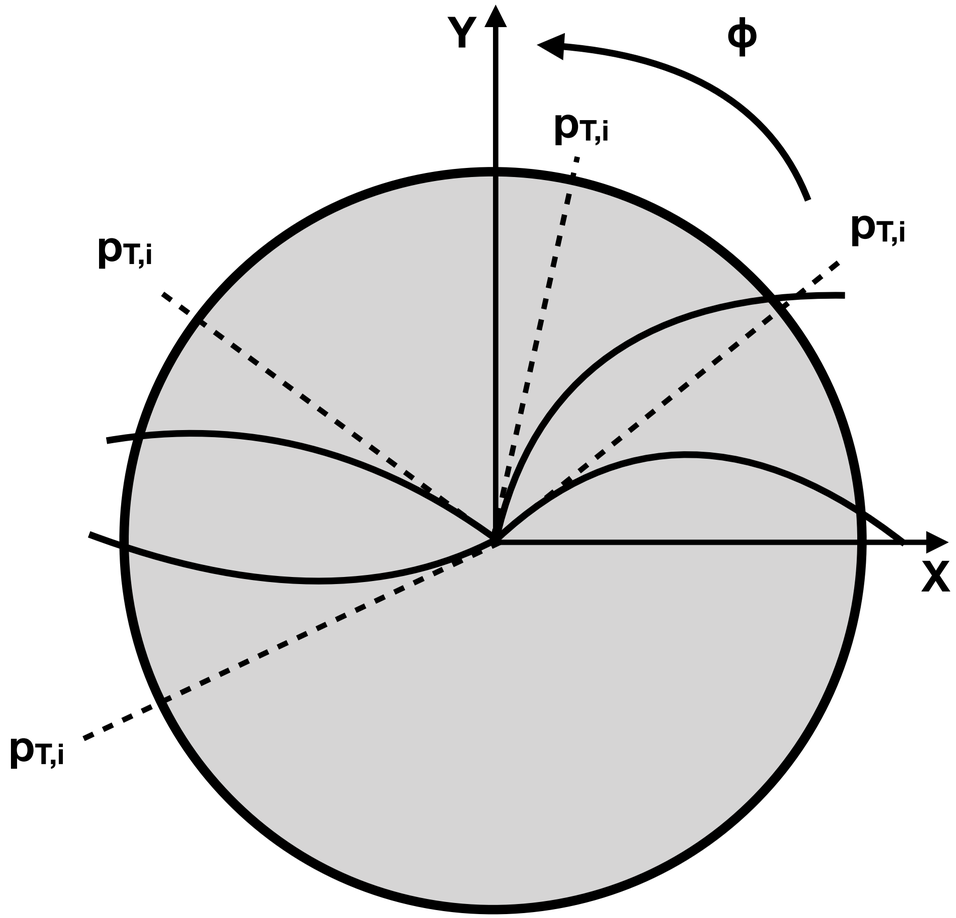
    - Or it could be a new type of particle that doesn't interact with the detector

  - * it's only true in the transverse plane since the colliding particles are constituents of the proton, potentially carrying different momenta in the longitudinal direction



Exercise

- You need to compute the MET (vector sum) from all of the *particles* in each *event*
  - An event refers to all of the particles produced from one collision at the LHC
- A particle is represented as an object with three properties:
  - Transverse momentum — p_T
  - Angle at vertex — ϕ
  - ‘Angle’ in longitudinal plane — η
- MET is the magnitude of the vector sum of particles
- You are given:
  - a file with particles from 1000 simulated events of a process with real MET*
  - a reference implementation in Python



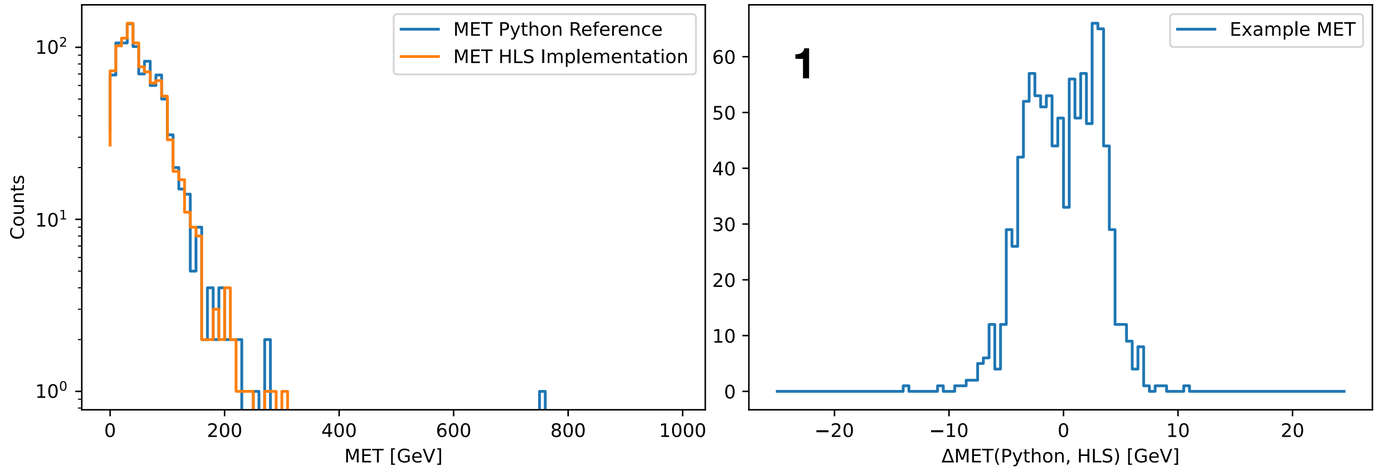
Maths functions

- There is a library of math functions for HLS available as #include “hls_math.h”
  - This include implementations of the trigonometric functions you’ll need e.g. `hls::cos` , `hls::sin` , `hls::sqrt`
  - Use them for your first attempt
  - Make a plot to validate the HLS trigonometry on your fixed point against the C++ math trigonometry on floating point
- In some cases we can be more efficient by preparing a Look Up Table that we can read with an address
  - Fill the table with the ‘ideal’ floating point function during compile / synthesis time
  - Read the table using the runtime data in the FPGA
  - If you have time, see if you can reduce the latency and resource usage of the trigonometry functions by replacing the HLS functions with a precomputed table
  - Validate your tables with a plot to compare

Evaluating your results

- In this exercise we will look at three metrics:
  1. Accuracy of the MET calculation against the Python reference
     - Aiming for 10 GeV absolute and 2% relative maximum difference
     - Use plots and other tools in the provided Python notebook to judge
  2. Resources of MET calculation HLS function
     - Use synthesis reports
  3. Latency of MET calculation HLS function
     - Use synthesis reports and cosimulation

1



2

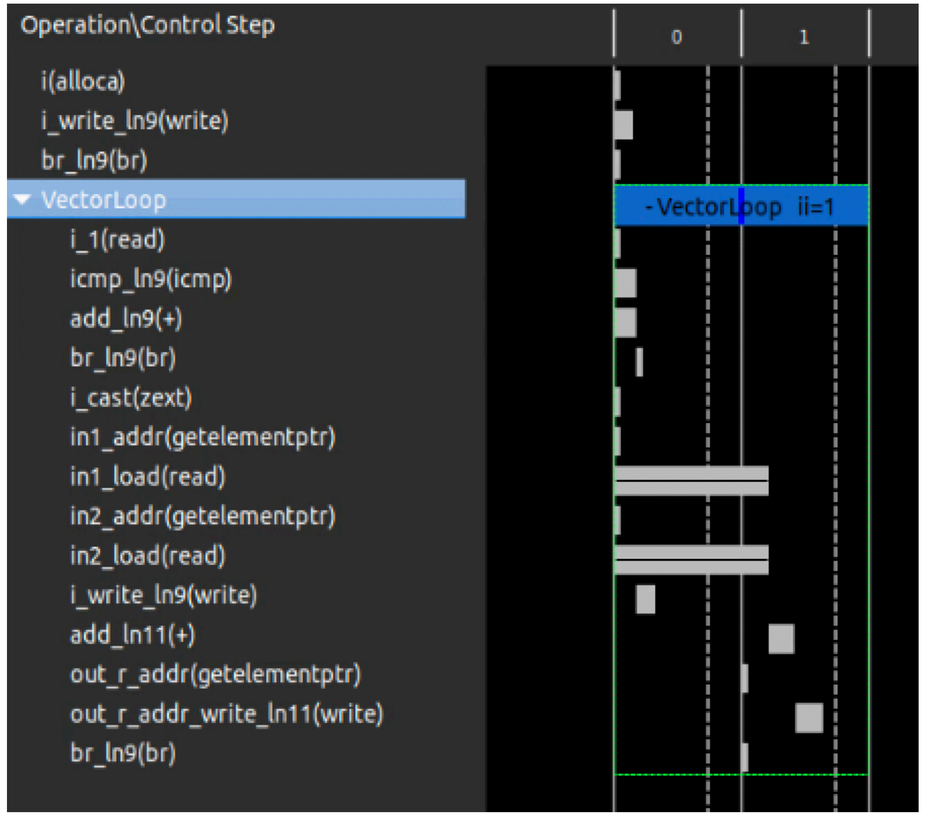
BRAM	DSP	FF	LUT	URAM
4 (1%)	6 (2%)	2139 (1%)	4364 (6%)	-
-	-	1834 (1%)	3614 (5%)	-
-	-	-	-	-
4 (1%)	-	90 (~0%)	311 (~0%)	-
-	-	-	-	-

3

RTL	Status	Latency(Clock Cycles)			Interval(Clock Cycles)			Total Execution Time (Clock Cycles)
		min	avg	max	min	avg	max	
VHDL	NA	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	428	1312	2756	429	1313	2757	1313303

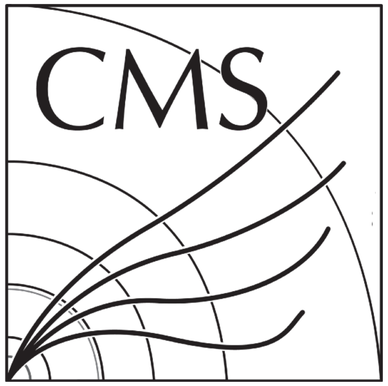
Improving your results

- Accuracy of the MET calculation
  - Explore the dataset and reference calculation in Python to choose appropriate precision
  - Remember that every operation and variable can have a different precision!
  - The HLS testbench from C Simulation writes a CSV file that the Python script can read to compare
  - Use the plots and “np.testing.assert_allclose” cell to evaluate and improve
  - When you change something, save the results to a different filename so that you can compare and improve
- Resources and Latency
  - Use the Schedule viewer and analysis view in Vitis HLS GUI
  - Giving labels to for-loops in HLS C++ can help identify them in reports
  - Defining functions for computation blocks can help identify them in reports



Part 1.1

Interfaces



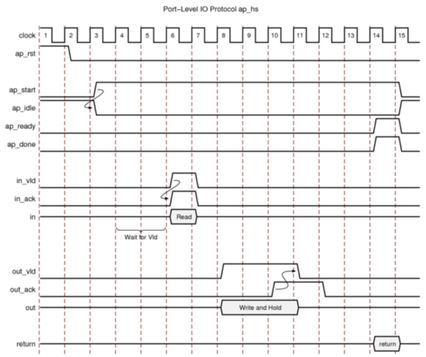
Introduction

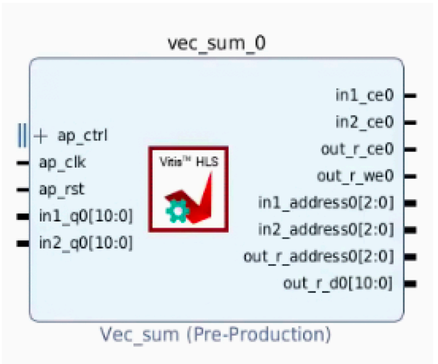
- In the first HLS exercises, we used:
  - C Simulation to interpret the numerical results
  - C Synthesis to evaluate the resource usage and latency
- But we neglected something important: how the HLS module will communicate with the outside world
- This section will give a brief overview and an example on the Arty 100T
  - Just to give you a flavour of the options

The basics

- When we run the C Synthesis we produce some RTL output: VHDL or Verilog, can be packaged as an “IP”
  - This RTL has some specific interface and an expected handshaking mechanism
  - Two things define the RTL interface:
    - The function signature of the C++
    - Any interface directives inside the function

```
void vec_sum(const data_t in1[8], const data_t in2[8],
data_t out[8]) {
    VectorLoop:
    for (int i = 0; i < 8; ++i) {
        out[i] = in1[i] + in2[i];
    }
}
```

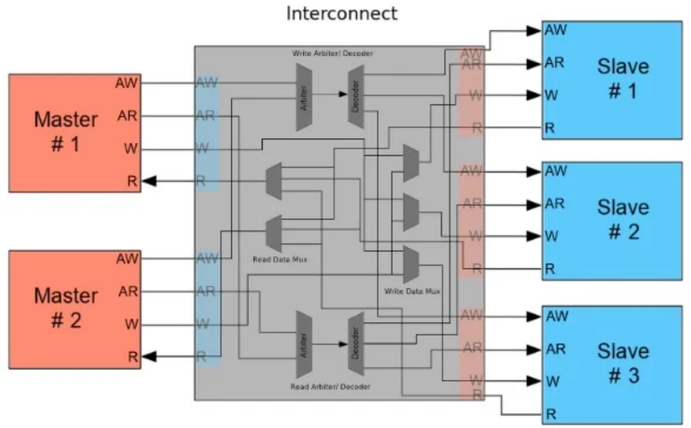





```
entity vec_sum is
port (
    ap_clk : IN STD_LOGIC;
    ap_rst : IN STD_LOGIC;
    ap_start : IN STD_LOGIC;
    ap_done : OUT STD_LOGIC;
    ap_idle : OUT STD_LOGIC;
    ap_ready : OUT STD_LOGIC;
    in1_address0 : OUT STD_LOGIC_VECTOR (2 downto 0);
    in1_ce0 : OUT STD_LOGIC;
    in1_q0 : IN STD_LOGIC_VECTOR (10 downto 0);
    in2_address0 : OUT STD_LOGIC_VECTOR (2 downto 0);
    in2_ce0 : OUT STD_LOGIC;
    in2_q0 : IN STD_LOGIC_VECTOR (10 downto 0);
    out_r_address0 : OUT STD_LOGIC_VECTOR (2 downto 0);
    out_r_ce0 : OUT STD_LOGIC;
    out_r_we0 : OUT STD_LOGIC;
    out_r_d0 : OUT STD_LOGIC_VECTOR (10 downto 0) );
end;
```

Interface types

- HLS abstracts huge flexibility about the interface — full guide from Xilinx “Interfaces of the HLS Design”
  - You can make very big changes about the interface type with small code changes
  - Even for an array, the access pattern of the array in the code will impact the ports
- The main types of interface:
  - Bare registers, memory ports, streams
  - AXI Protocol (Advanced eXtensible Interface) : AXI4-Lite, AXI4-Stream, AXI4-Master

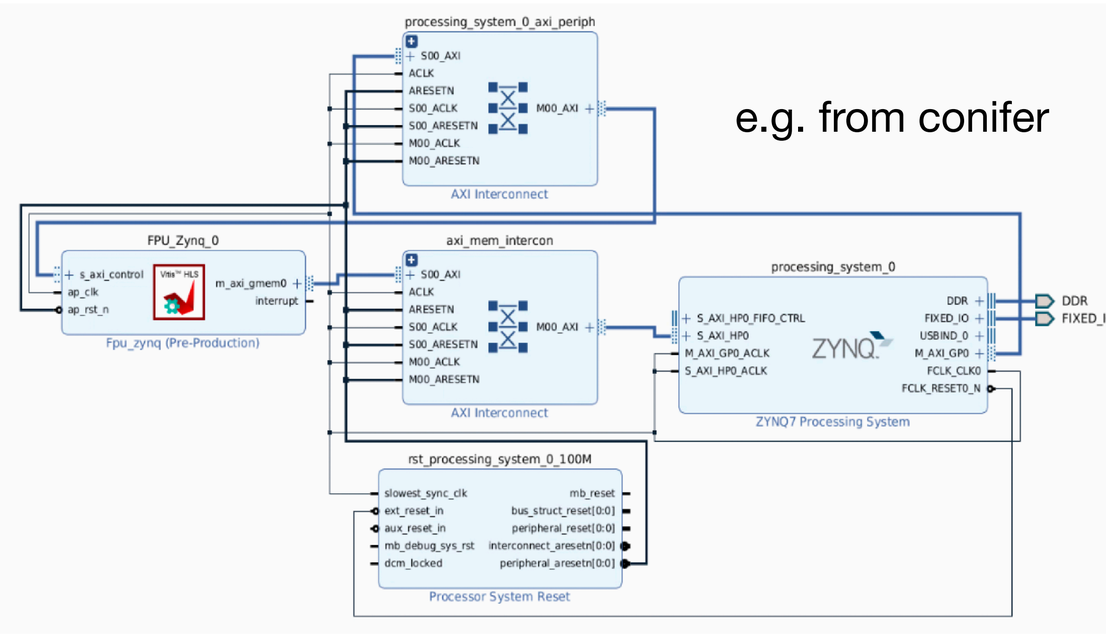


- Bare registers, memory ports, streams:
  - Low level, low overhead, full control
  - Manual integration
- We use this for designs at the CMS Level 1 Trigger

- AXI Protocol
  - Flexible communication and topology
  - Easy integration with other AMD IP, full board designs, AMD Software
- We use this for accelerator designs in hls4ml and conifer

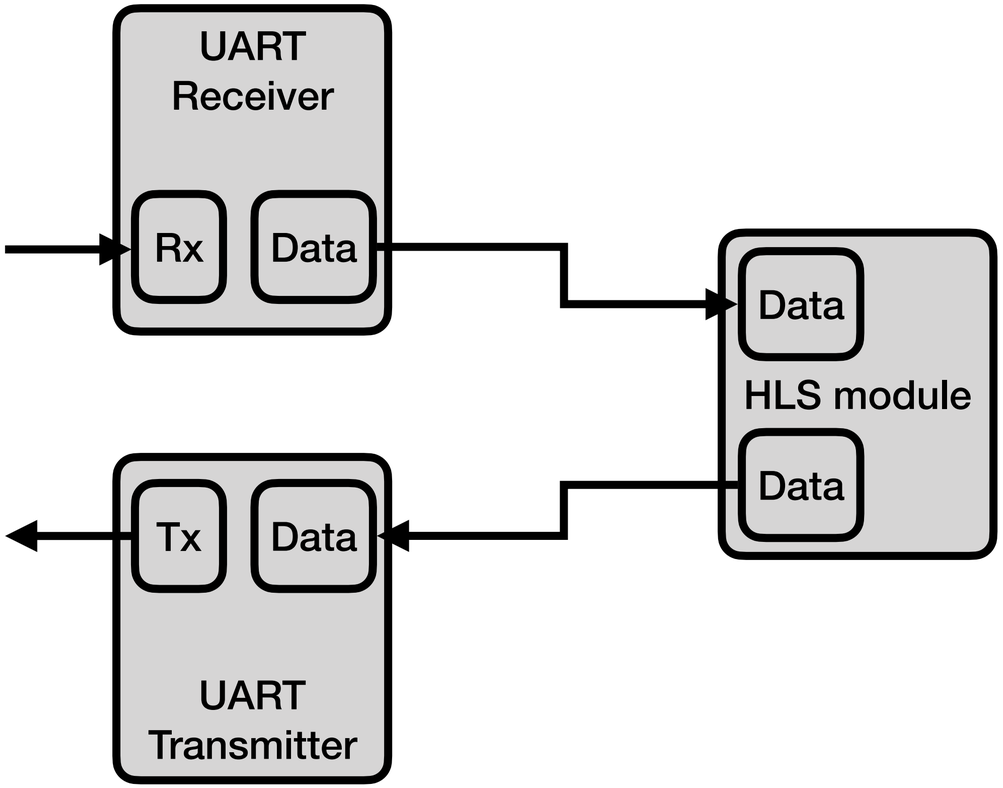
Making a complete design

- After defining how the HLS will module will communicate, we need to integrate it into a full design
- There are a few main ways, all useful for different contexts:
  - Using the ‘vitis’ tool (formerly SDAccel) to target boards like Alveo
    - e.g. `v++ -t hw --platform xilinx_u250_gen3x16_xdma_4_1_202210_1 -o hls_kernel.xo`
      - One command to create a full design (connecting AXI ports in the HLS kernel to PCIe) and run synthesis & implementation with Vivado
  - Using Vivado and ‘IP Integrator’
    - Connect together blocks with some abstraction and automation
      - Tool recognises common interface types that can be connected
  - Using Vivado and VHDL or Verilog
    - Make your own top-level design and instantiate the HLS RTL inside



Simple HLS Integration Example

- We will use the UART design from Day 3
  - Use your own implementation or the solution
- Instead of looping the RX data back to the TX output, we connect to an HLS module
  - HLS module will keep a rolling sum of the received data and send the current sum after every update
- The design will use:
  - stream interfaces in HLS
  - VHDL integration in Vivado



Simple HLS Example

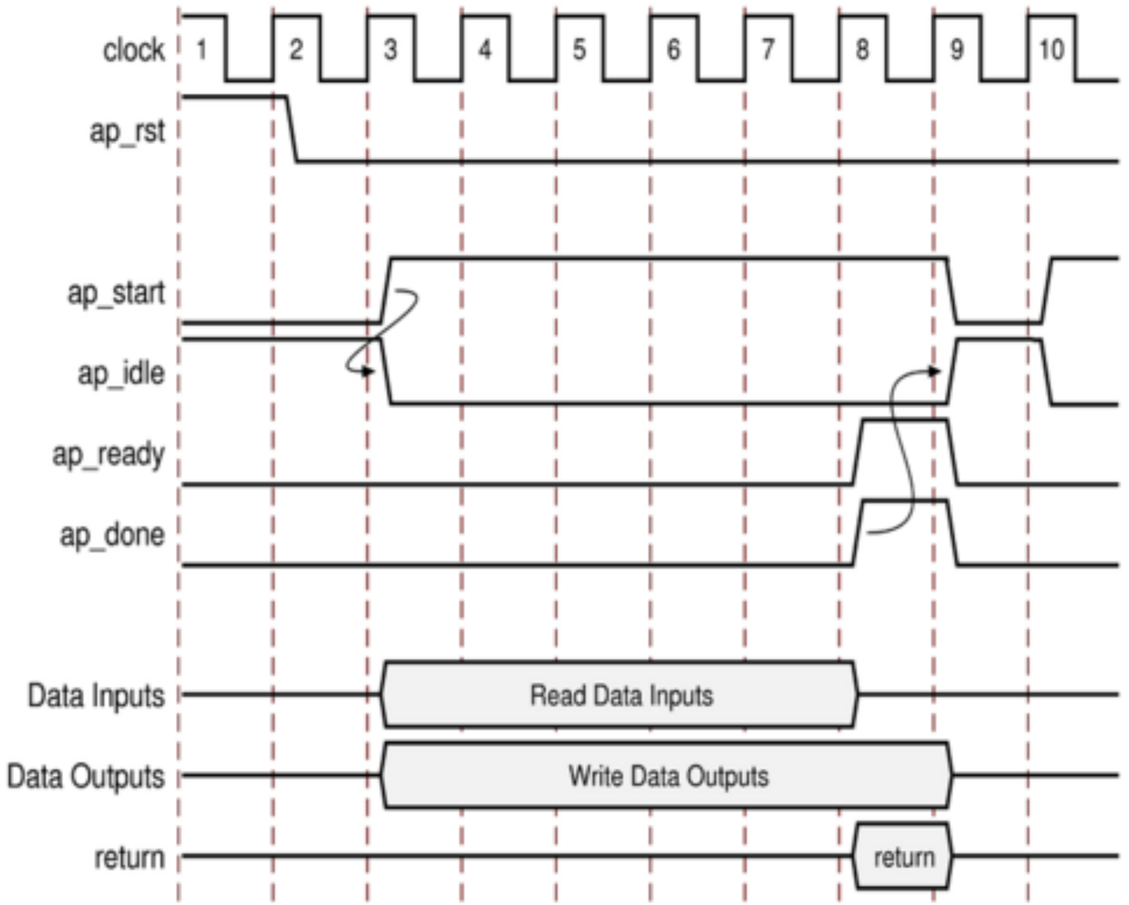
- Accumulator the value received from UART and send the accumulator value
- Static variables
  - After we program the FPGA, the design stays running there indefinitely
  - We “call” the HLS function in the FPGA by sending a start signal
  - In this design we want to accumulate the value every time we send new data
  - We need to persist the value in the device: static variables do that
  - In C++: static variables maintain their value between function calls
  - In FPGA: the static variable will be represented with some register / LUT that is not reset after function completion

```
void accumulator(ap_uint<8> in, ap_uint<8>& out) {
    static ap_uint<8> sum = 0;
    sum += in;
    out = sum;
}
```

```
entity accumulator is
port (
    ap_clk : IN STD_LOGIC;
    ap_rst : IN STD_LOGIC;
    ap_start : IN STD_LOGIC;
    ap_done : OUT STD_LOGIC;
    ap_idle : OUT STD_LOGIC;
    ap_ready : OUT STD_LOGIC;
    in_r : IN STD_LOGIC_VECTOR (7 downto 0);
    out_r : OUT STD_LOGIC_VECTOR (7 downto 0);
    out_r_ap_vld : OUT STD_LOGIC );
end;
```

HLS Integration Exercise

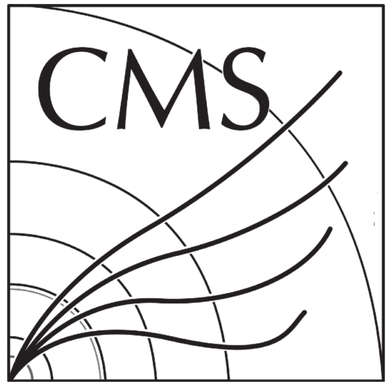
- Exercise:
  - Connect the ports of the HLS entity in the top.vhd design to the correct signals of the UART RX and TX entities
  - Hints:
    - use the diagram to the right for guidance
    - you won't need additional logic besides connecting signals
  - Synthesize, Implement the design in Vivado and program the Arty
  - Test with the Serial console or provided Python driver



AMD: Port level protocol with default synthesis

Part 2

Loops: analyzing and optimizing

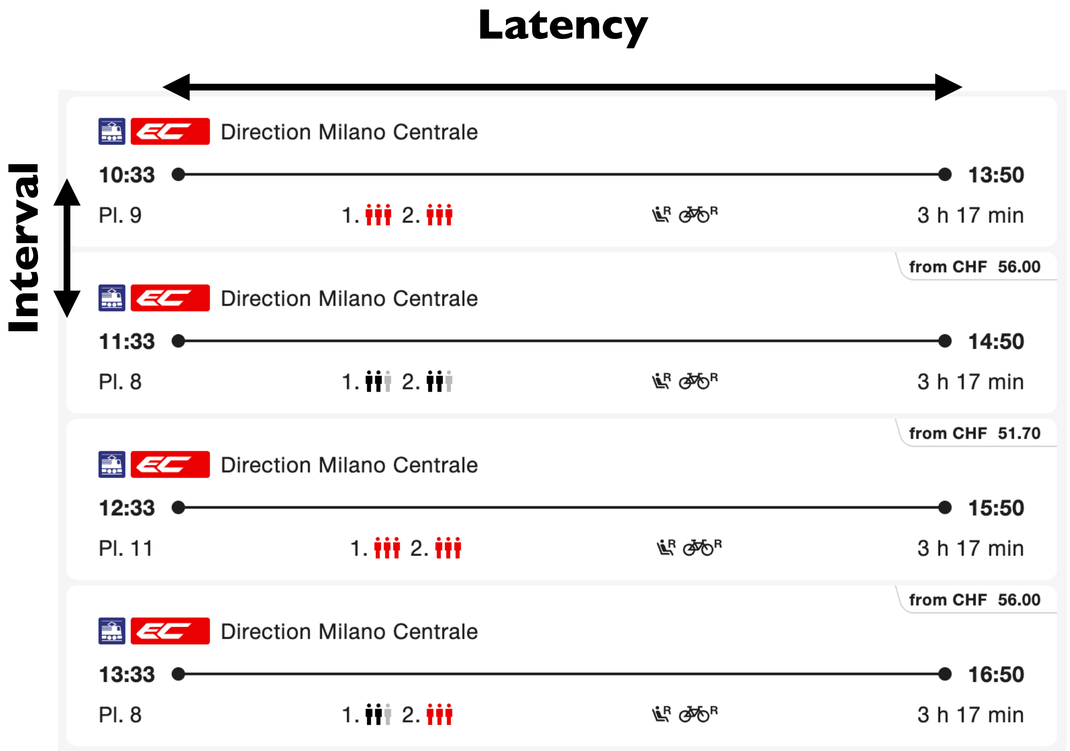


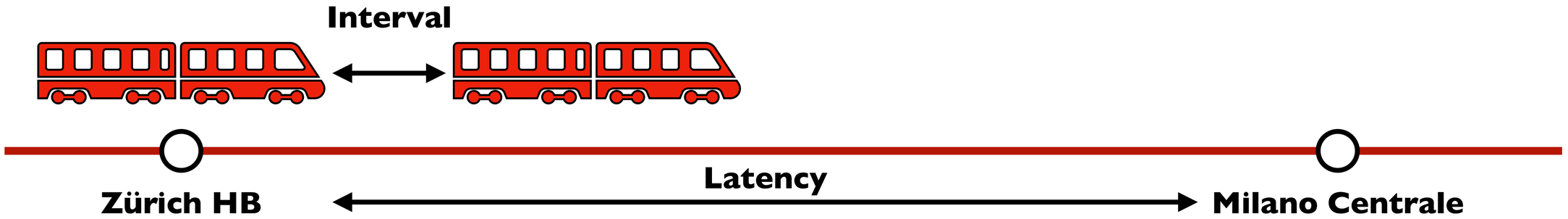
Introduction

- Loop optimization is the core concept to efficient HLS design
- In this section we'll go over the fundamental principles and explore how to control loops in HLS code
- Basics:
- Loop bounds: are they constant (at synthesis time) or variable (data dependent)?
  - e.g. the vector addition example had constant loop bounds (dimension 8) but MET exercise had variable loop bounds (N particles)
- Memory: arrays are implemented in "memory" in the FPGA
  - It's difficult to think about loops without learning about more about memory
  - Arrays can be mapped to any memory type: LUTs & FFs, Block RAM (BRAM), Ultra RAM (URAM), External DDR
  - The different types have different attributes in terms of size — performance tradeoff

Pipeline

- With FPGAs we can take advantage of pipeline processing
- We need to work to keep the pipeline filled with data
- Depends on the loops of our algorithm and their inter-dependencies
- First some terminology:
  - ‘Interval’ or ‘Initiation Interval’ : gap between *start* of subsequent executions of a process
    - How often to trains depart the station? (Once per hour)
  - ‘Latency’ : delay between start of execution of a process, and output of results
    - How long does it take to get from A to B? (3 hours 17 minutes)
- Main advantage of pipelining: latency and interval are not coupled!





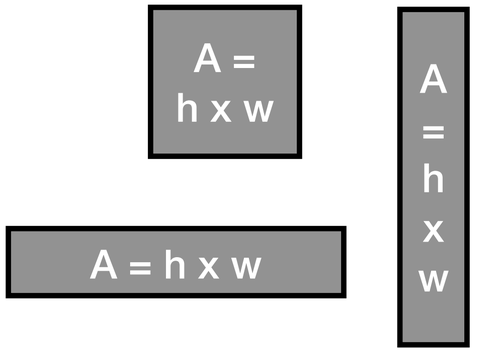
Memory

- Here are the main types of memory available on FPGAs
  - Note that DDR is an external component — the others are inside the FPGA

Memory Type	Max Size (per block)	Access Ports	Latency	Typical Use	Notes
Register / Logic	~bits	unlimited	1 clk	Scalars, small arrays	Inferred from variables
LUTRAM	~bits	1R1W	1 clk	Small, distributed structures	Uses LUTs → consumes logic
BRAM	18K–36K bits	1R1W or 2R	1 clk	Medium arrays, buffers	Parallel access requires partitioning
URAM	288K bits	1R1W	2 clk	Large buffers, high reuse	Only on large devices (e.g. Ultrascale+)
External DDR	GBs	AXI burst	100+ clk	Large datasets, software-like	Sequential access only; not suitable for pipelining

LUT RAM vs Block RAM

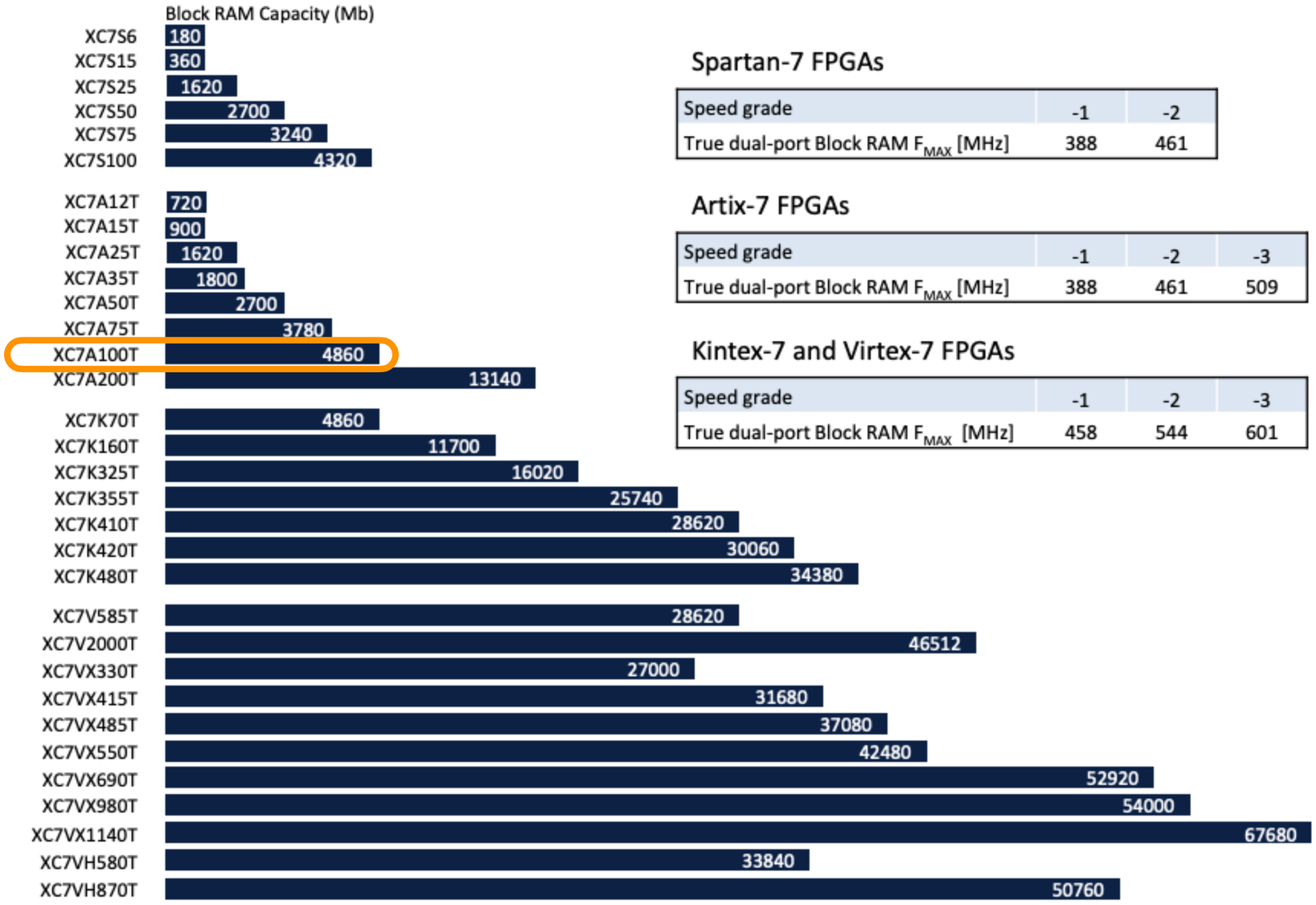
- LUT RAM (a.k.a. Distributed RAM)
  - Built from LUTs in the logic fabric
- Small and fast, ideal for:
  - Small lookup tables, FIFOs, temporary storage
- *Inferred* when arrays are very small (e.g. <32 elements)
  - i.e. don’t have to explicitly write code for it
- Advantage: low latency and logic proximity
- Tradeoff: consumes LUT resources, which may be needed for logic



- Block RAM (BRAM) — FPGA's Dedicated Memory Blocks
  - Fixed capacity per block (typically 18K or 36K bits)
- Multiple configurations: e.g. 512×36, 1K×18, 2K×9
  - Configurable address & data width
  - Total size is fixed, but width/height can vary
- Wider data bus = fewer addressable locations
- Typically 1 or 2 access ports
  - 1R1W (1 read, 1 write) or 2R (2 reads)
- Shared between loop iterations → can limit pipelining
- Used automatically by HLS for medium arrays
- View usage and mapping in synthesis reports (.rpt)

Memory

- Memory capacity is an important consideration when choosing a device
- e.g. Xilinx 7-series Product Select Guide
- FPGAs enable acceleration by combining parallelism with an application-specific memory hierarchy
  - explicitly controlling where and how data is stored and accessed
  - unlike CPUs with general-purpose caches



Spartan-7 FPGAs

Speed grade	-1	-2
True dual-port Block RAM F_{MAX} [MHz]	388	461

Artix-7 FPGAs

Speed grade	-1	-2	-3
True dual-port Block RAM F_{MAX} [MHz]	388	461	509

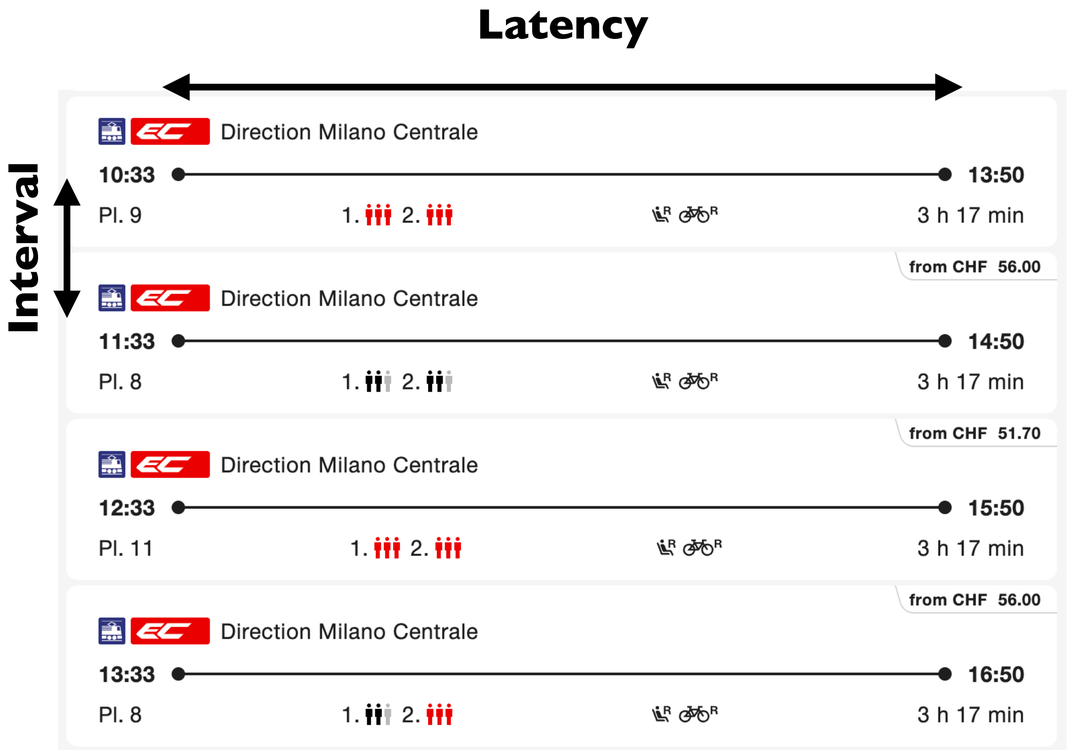
Kintex-7 and Virtex-7 FPGAs

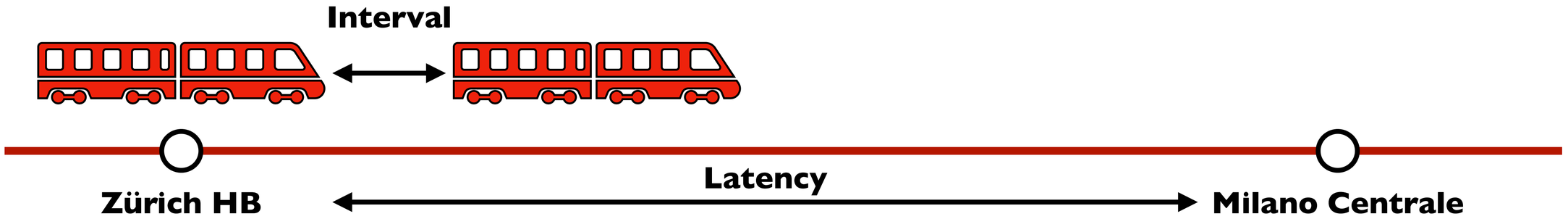
Speed grade	-1	-2	-3
True dual-port Block RAM F_{MAX} [MHz]	458	544	601

For more information, refer to: UG473, *7 Series FPGAs Memory Resources User Guide*

Loop Analysis

- With FPGAs we can take advantage of pipeline processing
- We need to work to keep the pipeline filled with data
- Depends on the loops of our algorithm and their inter-dependencies
- First some terminology:
  - ‘Interval’ or ‘Initiation Interval’ : gap between *start* of subsequent executions of a process
    - How often to trains depart the station? (Once per hour)
  - ‘Latency’ : delay between start of execution of a process, and output of results
    - How long does it take to get from A to B? (3 hours 17 minutes)
- Main advantage of pipelining: latency and interval are not coupled!

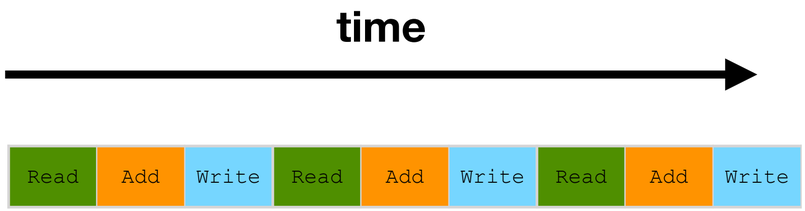




Loop Analysis

- Loops can have dependencies that impacts scheduling, unrolling, and interval
- Consider this loop executed sequentially

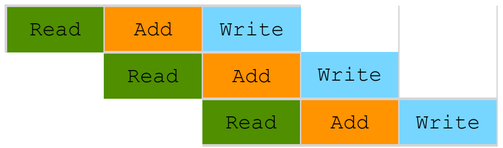
```
for(i = 0; i < 3; i++)
    a[i] = a[i] + 1;
```



  - The loop has Latency 3 cycles, Interval 3 cycles

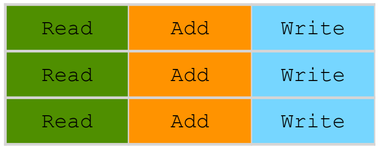
- This loop has no iteration dependence (iteration `i` does not depend on any other iteration)
  - It can be pipelined: loop has Latency 3 cycles, Interval 1 cycle

```
for(i = 0; i < 3; i++)
    a[i] = a[i] + 1;
```



- If all of `a[i]` can be read simultaneously (e.g. it's in FPGA registers not BRAMs), the loop can be *unrolled*

```
for(i = 0; i < 3; i++)
    a[i] = a[i] + 1;
```



Loop Analysis

- Some loops have dependencies (loop-carried dependence)

```
for(i = n; i > 0; i--)
    a[i] = a[i-1] + x[i];
```

- We can’t pipeline or unroll this loop since the `read` of iteration `i` depends on the `write` of iteration `i-1`

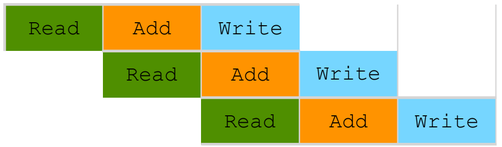
- For best performance with parallel architectures, we need to understand and optimise our loops
  - Defines how we can distribute loop iterations across different processing units
  - Merge loops where possible
  - Break dependencies by reordering loops

Loops Optimizing

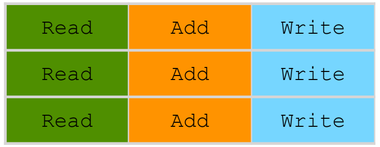
- The way we control loops in HLS is partly through how we write the C++ (more on this later) and partly through *pragmas*
  - Directives that the synthesis tool uses to map the code to RTL

```
for(i = 0; i < 3; i++)
    a[i] = a[i] + 1;
```


```
for(i = 0; i < 3; i++)
#pragma hls pipeline
    a[i] = a[i] + 1;
```



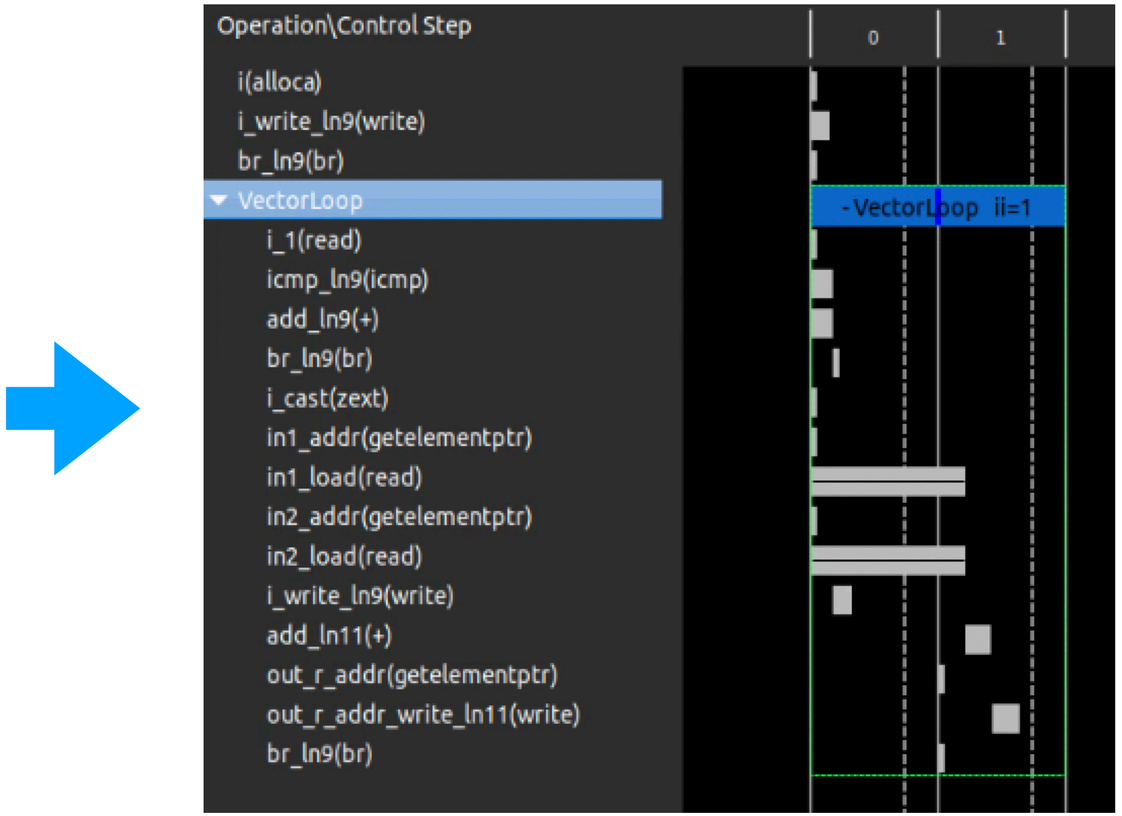

```
for(i = 0; i < 3; i++)
#pragma hls unroll
    a[i] = a[i] + 1;
```



Loop Optimizing

- The way we control loops in HLS is partly through how we write the C++ (more on this later) and partly through *pragmas*
  - Directives that the synthesis tool uses to map the code to RTL
  - Recall our vector sum example from the arithmetic section
  - The provided implementation was scheduled like this:

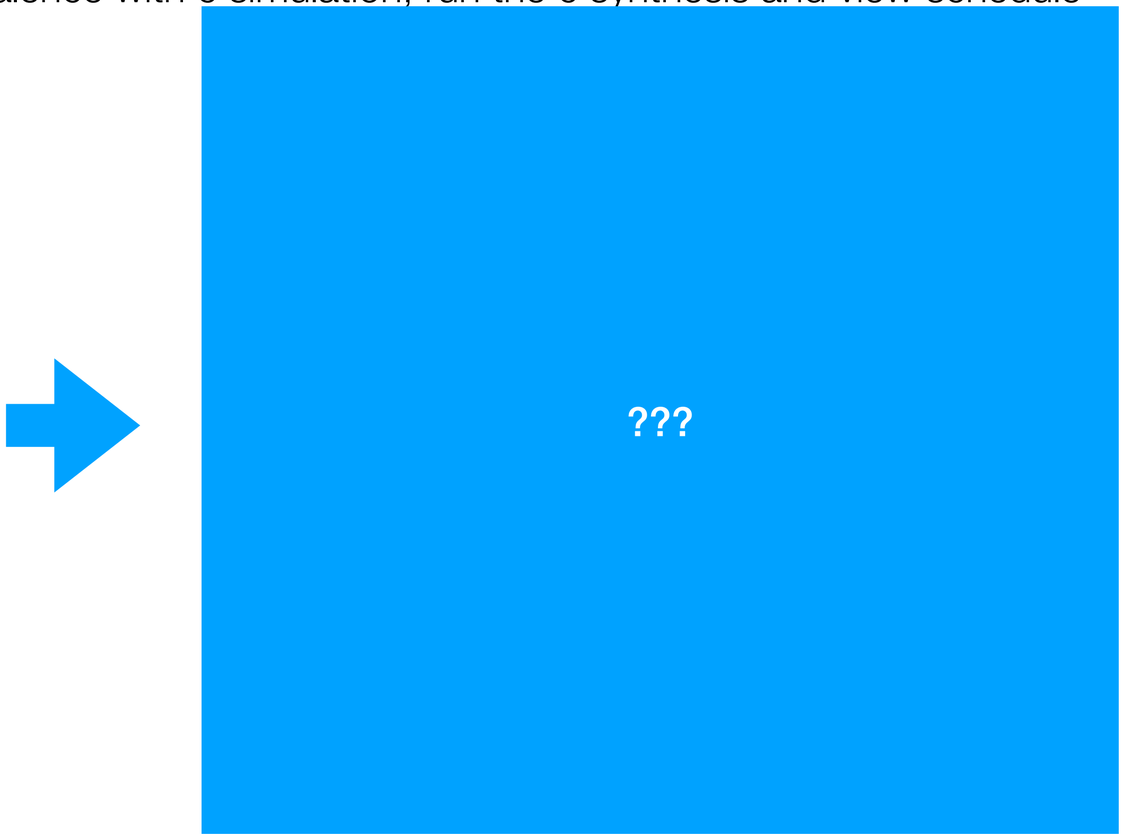
```
void vec_sum(const data_t in1[8],
             const data_t in2[8],
             data_t out[8]) {
   for (int i = 0; i < 8; ++i) {
       out[i] = in1[i] + in2[i];
    }
}
```



Loop Optimizing

- The way we control loops in HLS is partly through how we write the C++ (more on this later) and partly through *pragmas*
  - Directives that the synthesis tool uses to map the code to RTL
  - Recall our vector sum example from the arithmetic section
  - **Exercise**: now add these *pragmas* to the code, verify the equivalence with c simulation, run the c synthesis and view schedule

```
void vec_sum(const data_t in1[8],
             const data_t in2[8],
             data_t out[8]) {
    #pragma HLS array_partition variable=in1
    #pragma HLS array_partition variable=in2
    #pragma HLS array_partition variable=out
    #pragma HLS pipeline
    for (int i = 0; i < 8; ++i) {
        #pragma HLS unroll
        out[i] = in1[i] + in2[i];
    }
}
```

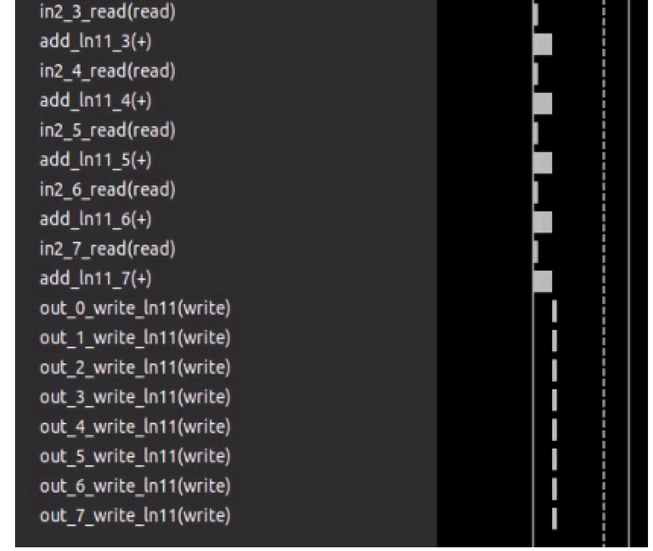


Skip

Loop Optimizing

- The way we control loops in HLS is partly through how we write the C++ (more on this later) and partly through *pragmas*
  - Directives that the synthesis tool uses to map the code to RTL
  - Recall our vector sum example from the arithmetic section
  - The order and parallelization of operations has completely changed

```
void vec_sum(const data_t in1[8],
             const data_t in2[8],
             data_t out[8]) {
    #pragma HLS array_partition variable=in1
    #pragma HLS array_partition variable=in2
    #pragma HLS array_partition variable=out
    #pragma HLS pipeline
    for (int i = 0; i < 8; ++i) {
        #pragma HLS unroll
        out[i] = in1[i] + in2[i];
    }
}
```



Pragma details

- What did each pragma do?

```
void vec_sum(const data_t in1[8],
             const data_t in2[8],
             data_t out[8]) {
    #pragma HLS array_partition variable=in1
    #pragma HLS array_partition variable=in2
    #pragma HLS array_partition variable=out
    #pragma HLS pipeline
    for (int i = 0; i < 8; ++i) {
        #pragma HLS unroll
        out[i] = in1[i] + in2[i];
    }
}
```

← Partition: make all of the array elements simultaneously readable

← Pipeline: enable concurrent operation execution

← Unroll: execute each loop iteration in parallel

Pragmas / loops

- **Exercise:** Try each of the pragmas in isolation, and log some key metrics:
  - Look at the schedule viewer for each one and reason what constraints are impacting the behaviour

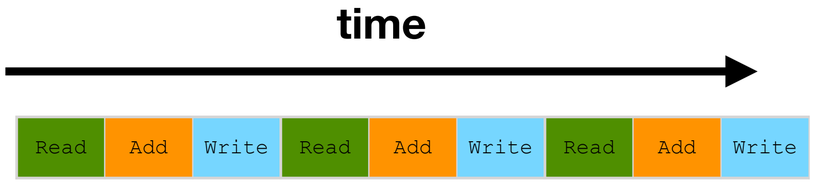
```
void vec_sum(const data_t in1[8],
             const data_t in2[8],
             data_t out[8]) {
    for (int i = 0; i < 8; ++i) {
        #pragma HLS unroll
        out[i] = in1[i] + in2[i];
    }
}
```

	LUT	Latency	Interval
No pragmas			
Unroll			
Array Partition			
Pipeline			
All pragmas			

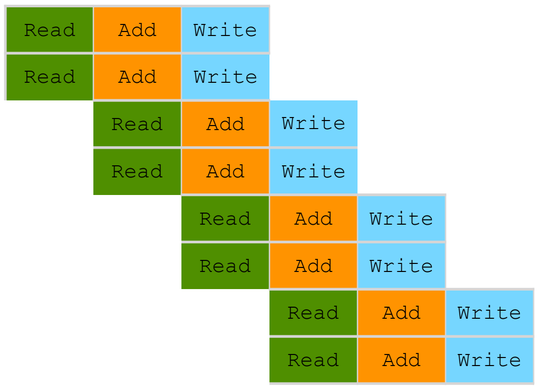
Partial unrolling

- We can also *partially* unroll a loop and *partially* partition an array
  - Useful when fully unrolling a loop will consume more resources than the chip has available
  - Example: partially unrolled with “factor = 2”

```
for(i = 0; i < 3; i++)
    a[i] = a[i] + 1;
```




```
for(i = 0; i < 8; i++)
#pragma hls unroll factor=2
    a[i] = a[i] + 1;
```



Loop bounds

- Some of the loop optimizations are only valid when the loop bounds are known at C Synthesis time
  - Consider: unrolled loop create *N* copies of the loop body in hardware → not possible if *N* is a variable
  - Recall our MET example top function: `T_met compute_met(unsigned short n_particles, T_pt* pt, T_pt* phi)`
  - The HLS Synthesis assumes the maximum value of `unsigned short` iterations for the loop over n_particles

```
WARNING: [HLS 200-936] Cannot unroll loop 'LOOP_X' (loop_var.cpp:22) in function 'loop_var':
cannot completely unroll a loop with a variable trip count.
```

- Sometimes this is unavoidable, but sometimes small changes can enable access to loop optimizations
  - Tell HLS the real limits: `#pragma HLS loop_tripcount min=<int> max=<int> avg=<int>`
  - Change a variable iteration loop to a fixed size one
    - Can then apply any unrolling, pipelining
    - May need to handle edge cases e.g. with conditional execution if out of loop bounds

Merging loops

- Code that seems well organized and natural in regular C++ for CPU may be suboptimal for HLS: merged loops
  - Example from HLS documentation

```
void top(a[4], b[4], c[4], d[4]){

  Add:
  for(i=3; i>=0; i++){
    if(d[i])
      a[i] = b[i] + c[i];
  }

  Sub:
  for(i=3; i>=0; i++){
    if(!d[i])
      a[i] = b[i] - c[i];
  }

}
```

- Left: HLS will execute the full Add loop then the full Sub loop
  - Plus a ‘control’ cycle to enter and exit each loop
  - Latency will be 11 cycles
- Right: HLS will execute the merged loop
  - Latency will be 6 cycles

```
void top(a[4], b[4], c[4], d[4]){

  AddOrSub:
  for(i=3; i>=0; i++){
    if(d[i])
      a[i] = b[i] + c[I];
    else
      a[i] = b[i] - c[i];
  }


}
```

Loops Exercise: faster MET

- **Exercise**: improve the performance of the MET computation from the first session
  - Use all of the loop analysis and optimization strategies from this section
  - Reorganize the code, use pragmas
- **Challenge**: achieve the lowest latency for the MET computation
  - Constraints: the resource usage must be less than 100% of the Arty 100 after C Synthesis individually for all resource types
  - The MET numerical result must satisfy the same limits: 10 GeV absolute and 2% relative maximum difference vs the floating point

	DSP	LUT	FF	BRAM	Latency	Interval	Worst Error	Mean Error
Original	?	?	?	?	?	?	?	?
Optimized	?	?	?	?	?	?	?	?