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

FPGA module training

Lecture-10: April 7th 2026

Varun Sharma

University of Wisconsin – Madison, USA

Content

So far

- Motivation & Introduction
- Comparison: FPGAs/ASICs/GPU/CPU
- Domain specific Accelerators
- HLS setup and first example project
- Unsupported C/C++ constructs
- Data types
- HLS Pragmas
  - Interface
  - Array partition
  - Array Reshape
  - Pipeline
  - DataFlow
  - Allocation
  - Latency
  - Inline
  - Stable

Today

- HLS Pragmas
  - Loop Unrolling
    - Unroll
  - Loop Optimization
    - Loop_flatten
    - Loop_merge
    - Loop_tripcount

HLS Pragmas

"Pragmas": Instructions to tell your compiler how to build the hardware

- HLS tool provides different set of pragmas that can be used to optimize the design, reduce latency, improve performance etc. These pragmas can be directly added to the source code for the kernel.

Type	Attributes
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
Function Inlining	• pragma HLS inline
Interface Synthesis	• pragma HLS interface • pragma HLS stream
Task-level Pipeline	• pragma HLS dataflow • pragma HLS stream
Pipeline	• pragma HLS pipeline • pragma HLS occurrence
Loop Unrolling	• pragma HLS unroll • pragma HLS dependence
Loop Optimization	• pragma HLS loop_flatten • pragma HLS loop_merge • pragma HLS loop_tripcount
Array Optimization	• pragma HLS array_partition • pragma HLS array_reshape
Structure Packing	• pragma HLS aggregate • pragma HLS dataflow
Resource Utilization	• pragma HLS allocation • pragma HLS bind_op • pragma HLS bind_storage • pragma HLS function_instantiate

https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/HLS-Pragmas

TAC-HEP 2026

#pragma HLS UNROLL

Pragma HLS unroll

- Unroll loops to create multiple independent operations rather than a single collection of operations

- **UNROLL** pragma transforms loops by creating multiples hw copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel

- Loops in the C/C++ functions are kept rolled by default
  - When loops are rolled, synthesis creates the logic for one iteration of the loop, and the RTL design executes this logic for each iteration of the loop in sequence

- **UNROLL** pragma allows the loop to be fully or partially unrolled
  - Fully unrolling the loop creates a copy of the loop body in the RTL for each loop iteration, so the entire loop can be run concurrently
  - Partially unrolling a loop lets you specify a factor *N*

Pragma HLS unroll

#pragma HLS unroll factor=<N> skip_exit_check off=true

factor=<*N*>: Specifies a non-zero integer indicating that partial unrolling is requested.

- If factor= is not specified, the loop is fully unrolled.

skip_exit_check: An optional keyword that applies only if partial unrolling is specified with factor=

- **Fixed (known) bounds:** No exit condition check is performed if the iteration count is a multiple of the factor. If the iteration count is not an integer multiple of the factor, the tool:
  - Prevents unrolling.
  - Issues a warning that the exit check must be performed to proceed.
- **Variable (unknown) bounds:** The exit condition check is removed as requested. You must ensure that:
  - The variable bounds is an integer multiple of the specified unroll factor.
  - No exit check is in fact require

Ex: Pragma HLS unroll

#pragma HLS unroll factor=<N> skip_exit_check off=true

- This example fully unrolls the loop
  - Instead of 1 adder used N times, N adders are working together

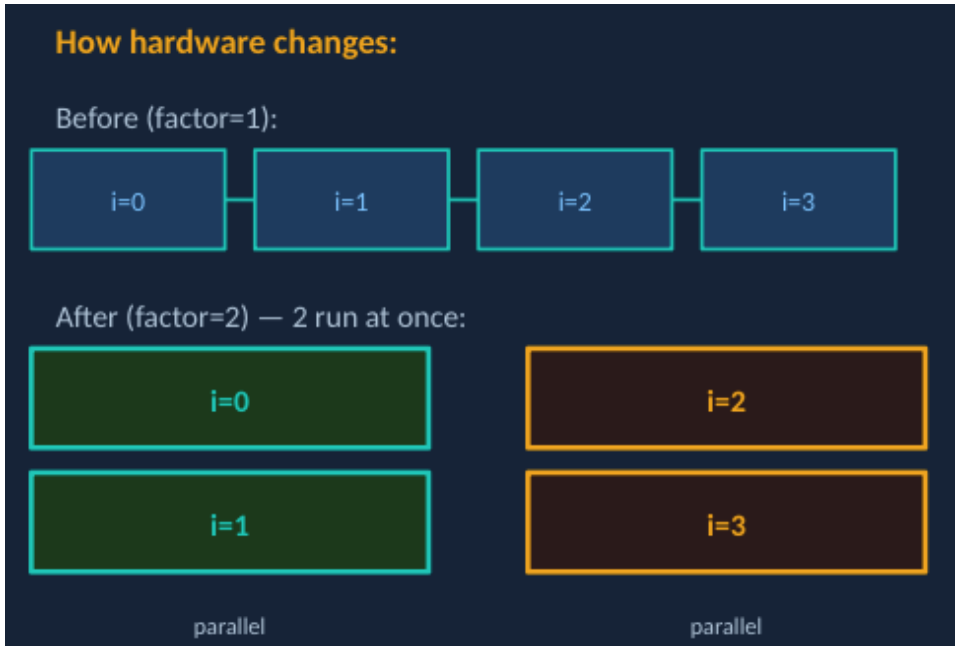
```
void example1(din_t A[N], din_t B[N], din_t C[N]){

 for (size_t i = 0; i < N; ++i) {
 #pragma HLS UNROLL
   C[i] = A[i] + B[i];
  }
}
```

- Patrial UNROLL: N/2

```
void example2(din_t A[N], din_t B[N], din_t C[N]){

 for (size_t i = 0; i < N; ++i) {
 #pragma HLS UNROLL factor = 2
   C[i] = A[i] + B[i];
  }
}
```



Ex: Pragma HLS unroll

#pragma HLS unroll factor=<N> skip_exit_check off=true

The following example fully unrolls loop_1 in function *foo*

Place the pragma in the body of loop_1 as shown:

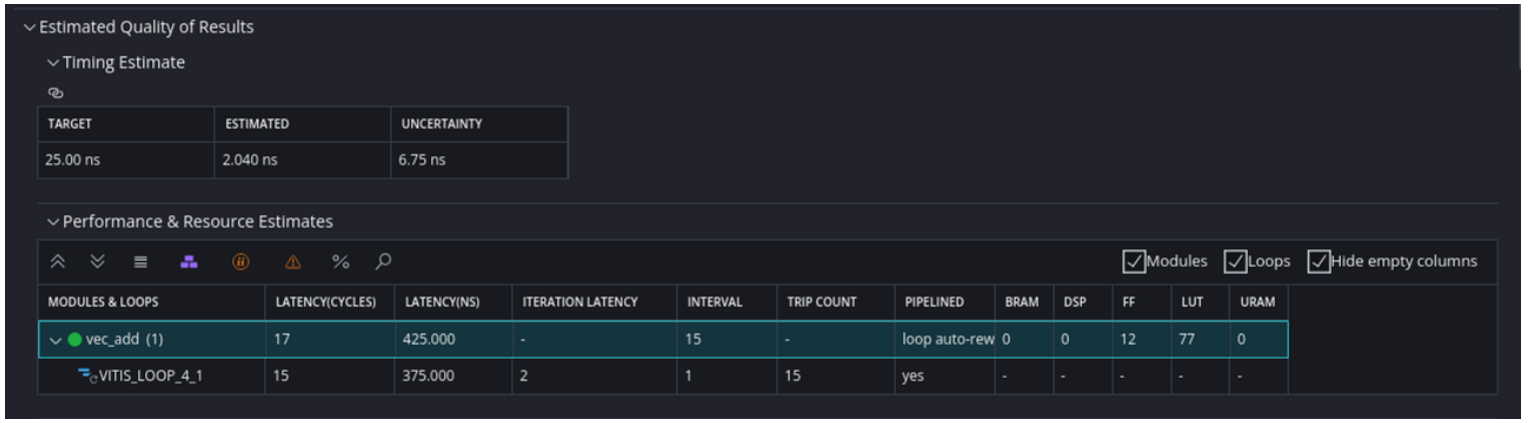
```
void example(din_t A[N], din_t B[N], din_t C[N]) {

 for (size_t i = 0; i < N; ++i) {
#pragma HLS UNROLL factor = 4
   C[i] = A[i] + B[i];
 }
}
```

This example specifies an unroll factor of 4 to partially unroll loop_2 of function foo, and removes the exit check:

```
void foo (...) {
  int8 array1[M];
  int12 array2[N];
  ...
  loop_2: for(i=0;i<M;i++) {
    #pragma HLS unroll skip_exit_check factor=4
    array1[i] = ...;
    array2[i] = ...;
    ...
  }
  ...
}
```

Reports: Can you guess the Unrolled?



Estimated Quality of Results

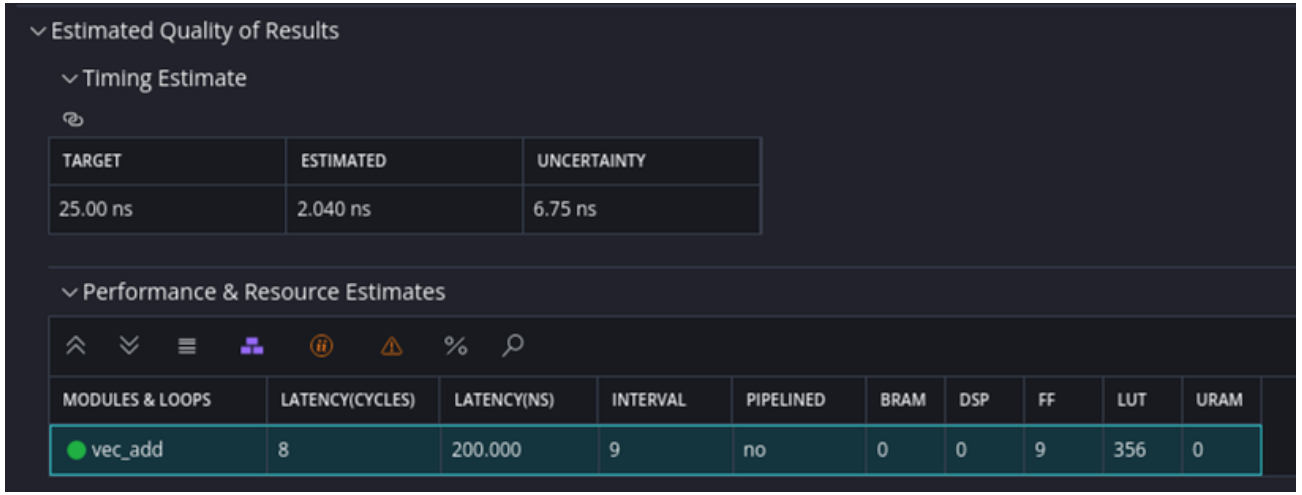
Timing Estimate

TARGET	ESTIMATED	UNCERTAINTY
25.00 ns	2.040 ns	6.75 ns

Performance & Resource Estimates

Modules Loops Hide empty columns

MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	ITERATION LATENCY	INTERVAL	TRIP COUNT	PIPELINED	BRAM	DSP	FF	LUT	URAM
vec_add (1)	17	425.000	-	15	-	loop auto-rew	0	0	12	77	0
VITIS_LOOP_4_1	15	375.000	2	1	15	yes	-	-	-	-	-



Estimated Quality of Results

Timing Estimate

TARGET	ESTIMATED	UNCERTAINTY
25.00 ns	2.040 ns	6.75 ns

Performance & Resource Estimates

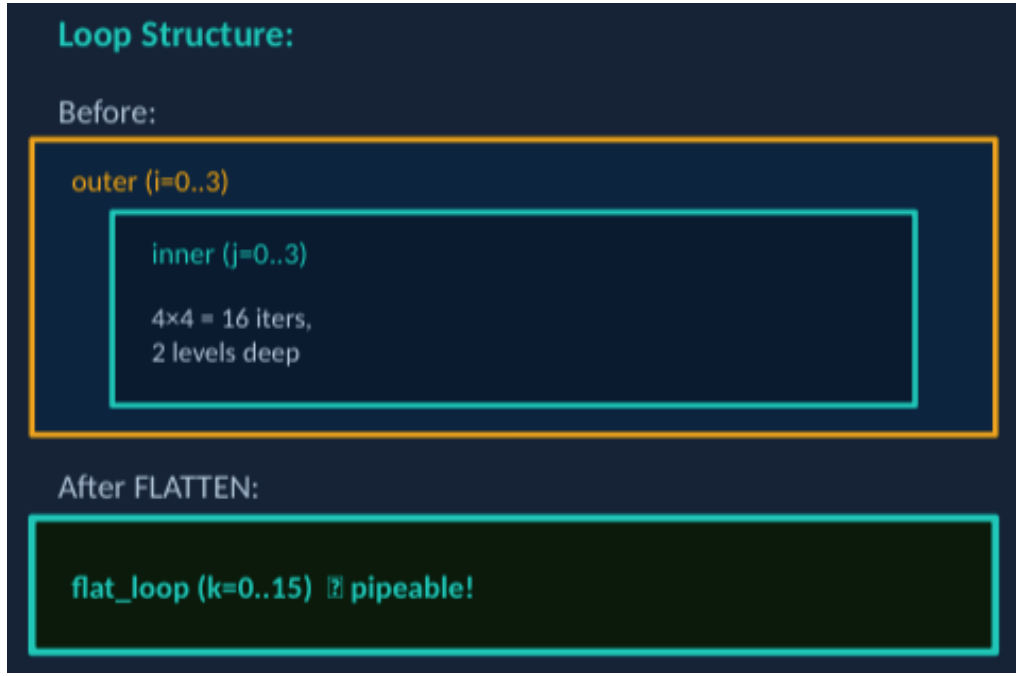
MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	INTERVAL	PIPELINED	BRAM	DSP	FF	LUT	URAM
vec_add	8	200.000	9	no	0	0	9	356	0

TAC-HEP 2026

#pragma HLS LOOP_FLATTEN

HLS Loop_flatten

- Allow nested loops to be flattened into a single loop hierarchy with improved latency
- In RTL implementation, it takes one clock-cycle to move from outer loop to the inner loop & vice versa
- Flattening nested loops allows them to be optimized as a single loop
  - Saves clock cycles
  - Allows for greater optimization of the loop body logic
- *LOOP_FLATTEN* pragma is applied to the loop body of the inner-most loop in the hierarchy
  - Only perfect & semi-perfect loops can be flattened



Different for-loops

- Perfect loop nest:
  - Only innermost loop body has content
  - No logic between loop statements
  - All loop bounds are constant

```
// Perfect Loop Example
for (int i = 0; i < 10; i++) {
   for (int j = 0; j < 20; j++) {
      #pragma HLS LOOP_FLATTEN
      A[i][j] = B[i][j] + 5;
}}
```

- Semi-perfect loop nest:
  - Only innermost loop body has content
  - No logic between loop statements
  - Outermost loop bound can be a variable

```
// Semi-Perfect Loop Example
for (int i = 0; i < 10; i++) {
   for (int j = 0; j < i; j++) { // Variable bound
      #pragma HLS LOOP_FLATTEN
      process(i, j);
}}
```

- Imperfect loop nest:
  - Loop body is not exclusively inside the inner loop
  - Try restructure or UNROLL

Pragma Loop_flatten Syntax

#pragma HLS loop_flatten off

```
void foo (num_samples, ...) {
  int i;
  ...
  loop_1: for(i=0;i< num_samples;i++) {
   #pragma HLS loop_flatten
   ...
    result = a + b;
  }
}
```

```
loop_1: for(i=0;i< num_samples;i++) {
  #pragma HLS loop_flatten off
  ...
```

Examples

#pragma HLS loop_flatten off

Perfect for loop

```
void example (
   int A[N],
   int B[N]
   ) {
 int sum;

Loop_I: for (size_t i=0 ; i < N; i++) {
  Loop_J: for (size_t j=0 ; j < N; j++) {
#pragma HLS loop_flatten
          if(j==0) sum = 0;
          sum  += A[j] *j;
          if(j==19){
            if(i%2 == 0)
              B[i] = sum/20;
            else
              B[i] =0;
          }
        }
      }
}
```

Imperfect for loop

```
void example (
   int A[N],
   int B[N]
   ) {
 int sum;

Loop_I: for (size_t i=0 ; i < N; i++) {
          sum = 0;
Loop_J: for (size_t j=0 ; j < N; j++) {
#pragma HLS loop_flatten
          sum  += A[j] *j;
        }
        if(i%2 == 0)
          B[i] = sum/20;
        else
          B[i] =0;
        }
}
```

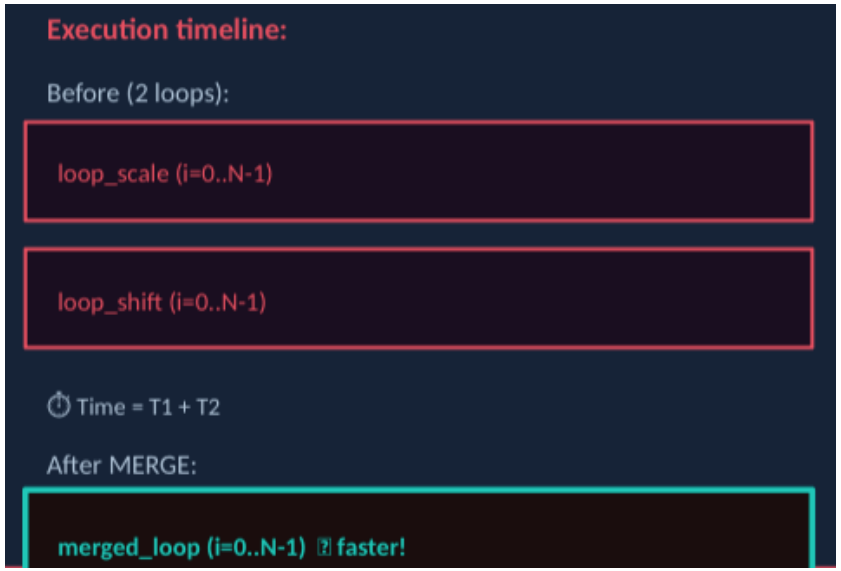
TAC-HEP 2026

#pragma HLS LOOP_MERGE

HLS LOOP_MERGE

- Merge consecutive loops into a single loop to reduce overall latency, increase sharing, and improve logic optimization

- Merging loops:
  - Reduces the number of clock cycles required in the RTL to transition between the loop-body implementations
  - Allows the loops be implemented in parallel (if possible)



- *force*: optional keyword to force loops to be merged even when HLS tool issues a warning

HLS LOOP_MERGE

- **Some rules before thinking to merge:**
  - For variable loop bounds, must have same value (# of iterations)
  - For constant loop bounds, the max. constant value is used as the bound of the merged loop

  - Loops with both variable bounds and constant bounds CANNOT be merged.

  - The code between loops to be merged cannot have side effects
    - Multiple execution of this code should generate the same results (a=b is allowed, a=a+1 is not).

  - Loops cannot be merged when they contain FIFO reads
    - Reads from a FIFO or FIFO interface must always be in sequence

HLS Loop_merge

#pragma HLS loop_merge force

```
void example (
   int A[N],
   int B[N],
   int C[N]
   ) {
  int sum;

//#pragma HLS LOOP_MERGE
Loop_I: for (size_t i=0 ; i < N; i++) {
//#pragma HLS PIPELINE II=1
         B[i] = A[i] + 1;
       }

Loop_J: for (size_t j=0 ; j < N; j++) {
         C[i] = A[i] * 1;
}

}
```

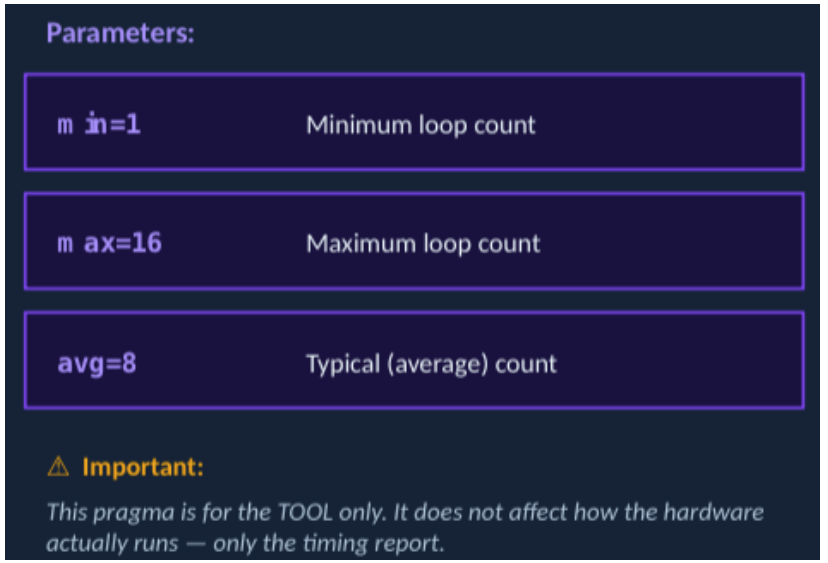
TAC-HEP 2026

#pragma HLS LOOP_TRIPCOUNT

HLS LOOP_TRIPCOUNT

- Loops: Known iterations, specify the total number of iterations performed by a loop.

- HLS reports the total latency of each loop, which is the number of clock cycles to execute all iterations of the loop
  - Loop latency is a function of the number of loop iterations, or trip count.
  - Includes cases in which the variables used to determine the trip count are:
    - Input arguments or
    - Variables calculated by dynamic operation



- In cases where the loop latency is unknown or cannot be calculated
  - The **LOOP_TRIPCOUNT pragma** lets you specify **minimum**, **maximum**, and **average** iterations for a loop
  - Let the tool analyze how the loop latency contributes to the total design latency in the reports, and helps you determine appropriate optimizations for the design

HLS LOOP_TRIPCOUNT : Syntax

#pragma HLS loop_tripcount min=<int> max=int avg=<int>

- **max= <int>** Specifies the maximum number of loop iterations.
- **min=<int>** Specifies the minimum number of loop iterations.
- **avg=<int>** Specifies the average number of loop iterations.

```
void example (
   int A[N],
   int B[N],
   int C[N]
   ) {
  int sum;

//#pragma HLS LOOP_MERGE
Loop_I: for (size_t i=0 ; i < N; i++) {
//#pragma HLS LOOP_TRIPCOUNT min=1 max=N avg=5
//#pragma HLS PIPELINE II=1
        B[i] = A[i] + 1;
      }

Loop_J: for (size_t j=0 ; j < N; j++) {
        C[i] = A[i] * 1;
}

}
```

Easy comparison

Pragma	Idea behind	Hardware effect
UNROLL	Parallel iterations	More area (resources) for higher speed
LOOP_FLATTEN	Remove nested loop control	Slight area increase; gain pipeline; Lower latency
LOOP_MERGE	Combine adjacent loops	Lower control overhead
LOOP_TRIPCOUNT	Help reporting	No hardware change

Homework: lecture 10

Homework # 6 (lecture 10)

Q9. Run code "hw6q9" and try to find the reason for why C-Simulation failed? Fix it and report it what was used to make it working. (Please don't use CHAT GPT or any help here). Its very trivial but important logic to understand.

Q10.Use matrix multiplication example used in Q5. for complete array partitioning and compare the results after using HLS UNROLL pragram and then using both HLS Array Paritioning and UNROLL together. Report your observation.

Q11. Make a very simple neural network, starting with a single dense layer which is nothing but a weighted Sum. You can assume that you are already have the weight matrix.

$$y_j = \sum_i x_i w_{ij}$$

- y_j = Sum_i {x_i*w_ij}, where, x_i is vector, w_ij are weights (ixj matrix).
- Use a combination of pragma discussed in class so far and get the best optimized results in terms of timing and resource utilization.
- Pragma's to choose from: Interface, Array parition, Array reshape, pipeline, dataflow, allocation, latency, inline, stable, loop unroll, loop_flatten, loop_merge, loop_tripcount

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

TAC-HEP 2026

Questions?

TAC-HEP 2026

Extra Slides

TAC-HEP 2026

HLS Setup on cmstrigger02

https://github.com/varuns23/TAC-HEP-FPGA/blob/main/hls-setup/readme.md

Setup

Setup Port Forwarding for RDP

- From your local machine, run:
  - `ssh -L 3389:localhost:3389 -J <username>@login.hep.wisc.edu <username>@cmstrigger02.hep.wisc.edu`
- Keep this terminal open - it maintains the RDP tunnel

Connect Using Remote Desktop (RDP Client)

- Use Microsoft's RDP client (called `Windows App`) available for macOS and Windows.
- **Setup:** Download & install the `Windows App`, Open the app and click the **+** icon, then select `Add PC`
- **Configure:** Enter the IP address: `localhost:3389` or `127.0.0.1:3389`
- **Connect:** Double-click the PC icon, enter your UW computing and , and click **Connect**
- You should now see the remote desktop of `cmstrigger02`

https://github.com/varuns23/TAC-HEP-FPGA/blob/main/hls-setup/readme.md

Setup workign Directory

After logging in via RDP, open a terminal inside the remote desktop

- `mkdir -p /scratch/`whoami`` (if not there already)
- `cd /scratch/`whoami``
  - **For Vitis HLS:**
    - `Source /opt/Xilinx/Vitis/2020.1/settings64.sh`
    - `cd /scratch/`whoami``
    - `vitis_hls`
  - **For Vivado HLS:**
    - `Source /opt/Xilinx/Vivado/2020.1/settings64.sh`
    - `cd /scratch/`whoami``
    - `vivado_hls`

Jargons

- **ICs - Integrated chip:** assembly of hundreds of millions of transistors on a minor chip
- **PCB:** Printed Circuit Board
- **LUT - Look Up Table aka 'logic'** - generic functions on small bitwidth inputs. Combine many to build the algorithm
- **FF - Flip Flops** - control the flow of data with the clock pulse. Used to build the pipeline and achieve high throughput
- **DSP - Digital Signal Processor** - performs multiplication and other arithmetic in the FPGA
- **BRAM - Block RAM** - hardened RAM resource. More efficient memories than using LUTs for more than a few elements
- **PCIe or PCI-E - Peripheral Component Interconnect Express**: is a serial expansion bus standard for connecting a computer to one or more peripheral devices
- **InfiniBand** is a computer networking communications standard used in high-performance computing that features very high throughput and very low latency
- **HLS** - High Level Synthesis - compiler for C, C++, SystemC into FPGA IP cores
- **HDL** - Hardware Description Language - low level language for describing circuits
- **RTL** - Register Transfer Level - the very low level description of the function and connection of logic gates
- **FIFO** – First In First Out memory
- **Latency** - time between starting processing and receiving the result
  - Measured in clock cycles or seconds
- **II - Initiation Interval** - time from accepting first input to accepting next input

Terminology

- **HLS file:** C/C++ code that will be synthesised and run on FPGA
- **Test bench (TB) file:** C/C++ code that is run to test the HLS code. It calls the HLS functions and can run tests on their output, e.g. C asserts
- **Tcl scripts:** set of tcl instructions executed by the Vivado HLS shell

- **Synthesis:** C/C++ ➞ HDL lang (VHDL/Verilog)
- **Project:** Collection of HLS and test bench (TB) files
  - Has a top-level function name that is the starting point for synthesis
- **Solution:** specific implementation of project
  - Runs on a specific device at a specific clock frequency
- **C simulation:** HLS+TB files are compiled with gcc against HLS headers and lib and plainly run as any other executable
- **C/RTL cosimulation:** synthesized HLS code is run on simulator and results tested on the C/C++ test bench