VITIS HLS TOOLS & OPTIMISATION

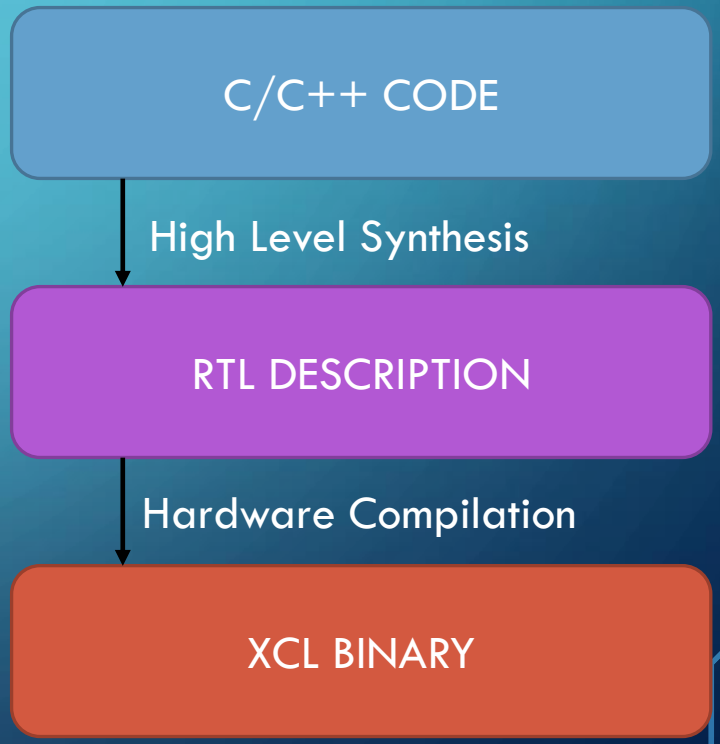
A GENTLE INTRODUCTION TO FPGA PROGRAMMING

FPGA DEVELOPMENT WORKFLOW

- Kernels written in a restricted subset of C++
  - No recursion
  - No function pointers
  - No dynamic memory
- Optimisation /resource management directives added & tested
- Kernels will still need to be written in a hardware-friendly way
- Hardware compilation for FPGA usable xclbin file
- Test with: C Simulation, RTL/C Co-Simulation, Software Emulation, Hardware Emulation, Hardware

WHAT IS HIGH LEVEL SYNTHESIS (HLS)?

- High Level Synthesis is a process which takes a higher level description (in our case C/C++ code) and converts it to a Register Transfer Level (RTL) description
- RTL is the level of abstraction occupied by Hardware Description Languages (HDLs) like VHDL and Verilog
- Vitis HLS is guided by the use of special directives: #pragma HLS …
- The RTL description can later be compiled into the explicit circuit representation needed to encode the hardware



VITIS HLS TOOLS

The Vitis HLS application allows us to do a number of things

- C Simulation: If you write a test suite which calls the kernel function (as a regular C function – no OpenCL required!) you can test that the C function is correct using C Simulation
- C Synthesis: Builds a model which includes scheduling and resource usage estimates. This is the most important step for applying and inspecting optimisations
- RTL / C Co-Simulation: Simulates execution of generated RTL code to check that the test results as the same as the C code

USING THE VITIS HLS APPLICATION

- Access one of the login nodes using X2GO or other remote desktop software (you will need visual information & GUI)
- Open a terminal and source the FPGA modules
- Run the command `vitis_hls`
- In the application, select “Create Project”
- Select the folder you want you project to go into, and name the project
- Add your kernel source file(s) and select the top level function
- (Optional) Add a test suite if you have one
- Select “Vitis Kernel Flow Target”, and in hardware “Alveo U280” board
- Run C-Synthesis

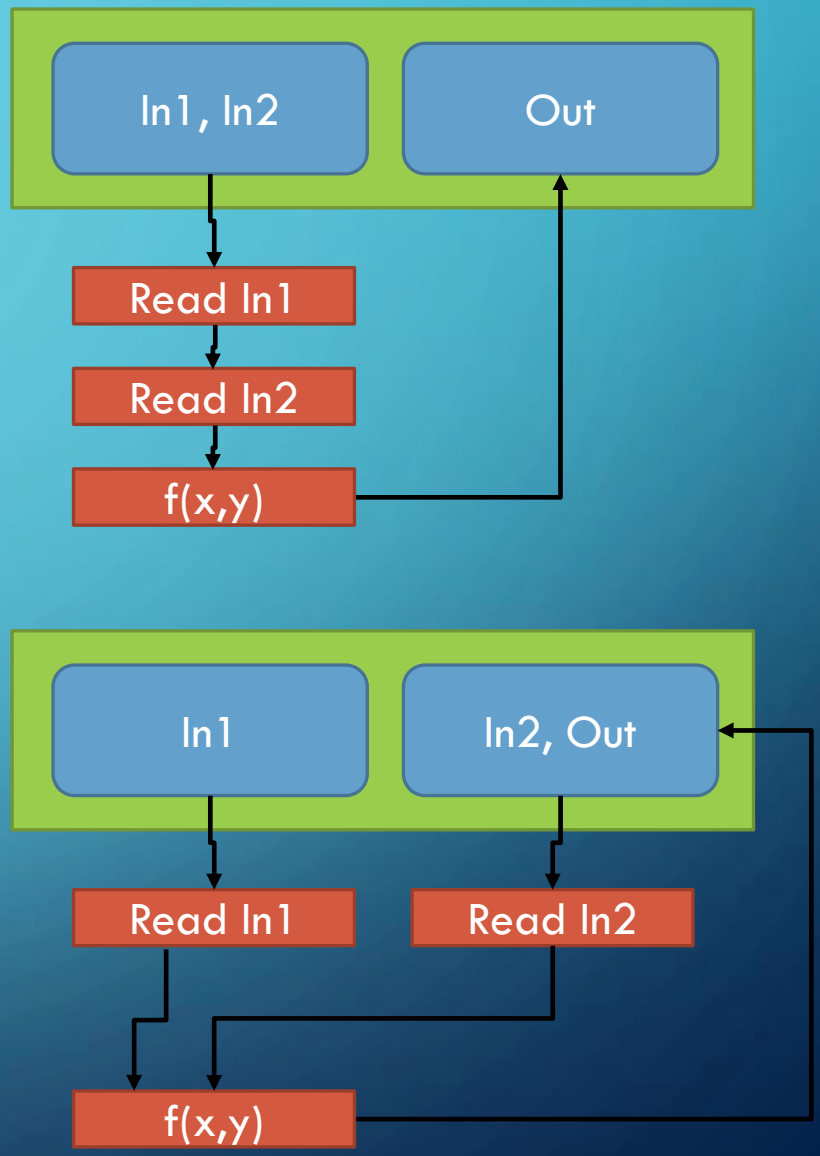
```
48  double sum(double *array)
49  {
50      double sum = 0.0;
51      double sum_buffer[4], sum_buffer_2[2];
52  #pragma HLS array_partition variable=sum_buffer complete
53  #pragma HLS array_partition variable=sum_buffer_2 complete
54      sum_array_loop:
55      for(int i = 0; i < 8; i+=4)
56      {
57  #pragma HLS pipeline II=2
58          for(int j = 0; j < 4; j++)
59          {
60              // Do these four in parallel
61  #pragma HLS UNROLL
62              sum_buffer[j] += array[i + j];
63          }
64      }
65      for(int j = 0; j < 2; j++)
66      {
67  #pragma HLS UNROLL
68          sum_buffer_2[j] = sum_buffer[j] + sum_buffer[j+2];
69      }
70      sum = sum_buffer_2[0] + sum_buffer_2[1];
71
72      return sum;
73  }
```

HLS DIRECTIVES

- #pragma HLS <directive> <options>
- HLS directives control optimisations and hardware level implementations that do not exist regular C code
- Can write them directly in your source code or add them in the Vitis HLS application
- There are many such pragmas, but we shall cover a few common ones.

INTERFACE

- #pragma HLS interface <mode> port=<argument name> [bundle=<string>]
- Determines use of global memory or external streams
- If you want to read arguments in parallel then you need separate interfaces (bundles), but the data must also be stored in separate banks (4 x DDR, 32 x HBM)
- Multiple interfaces reading from the same memory banks will be sequentialised
- Don't jump around in memory! Read contiguous blocks whenever you can (similar to caching strategies)
- This pragma has lots of other options and applications!



BIND STORAGE

- #pragma HLS bind_storage variable=<variable> type=<type> impl=<implementation>
- Allows us to select the type of memory used for a given variable
- Types include FIFO, RAM_1P, RAM_2P, RAM_T2P etc.
- Implementations include BRAM (block RAM), URAM (ultra-RAM), LUTRAM (look-up table RAM) and AUTO
- Latency is lower if resources are localised
- Make sure memory structures match read/write patterns

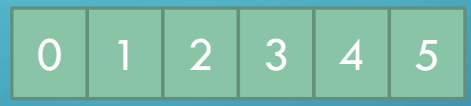
PIPELINE

- #pragma HLS pipeline II=<num>
- Pipelining is a key aspect of FPGA optimisation
- Pipelining is a form of parallelism which is ideal for FIFO data-structures
- II – 'Initiation Interval' – refers to the number cycles between starting working on one data element and the next
- Vitis compiler will often try to do this automatically and use II=1 default; you may have to manually set II to avoid problems

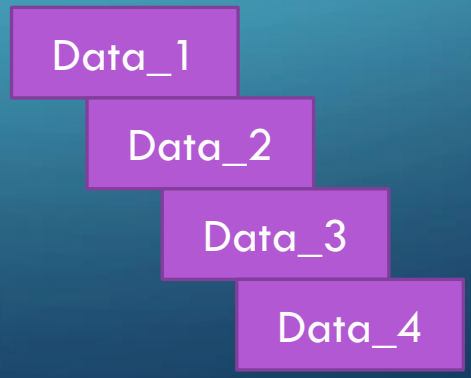
…

…

$$t_{tot} = N_{data} \times t_0$$



…



…

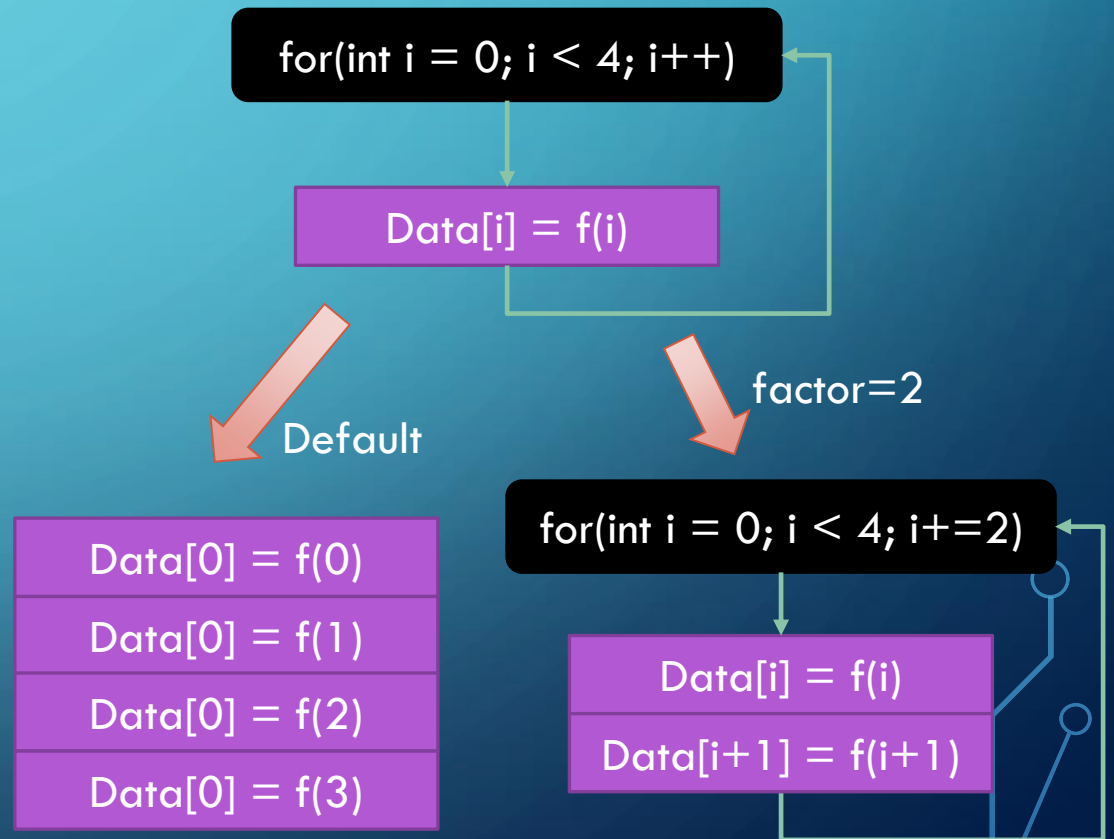
$$t_{tot} = (N_{data} \times II) + t_0 - 1$$

ARRAY PARTITION

- #pragma HLS array_partition <variable> <type> <factor>
- Breaks up an array in memory into multiple blocks with separate read/write ports
- Allows us to access these blocks of memory in parallel
- Type can be: complete, cyclic, block
- In the code you continue to refer to the array and indexing as usual!

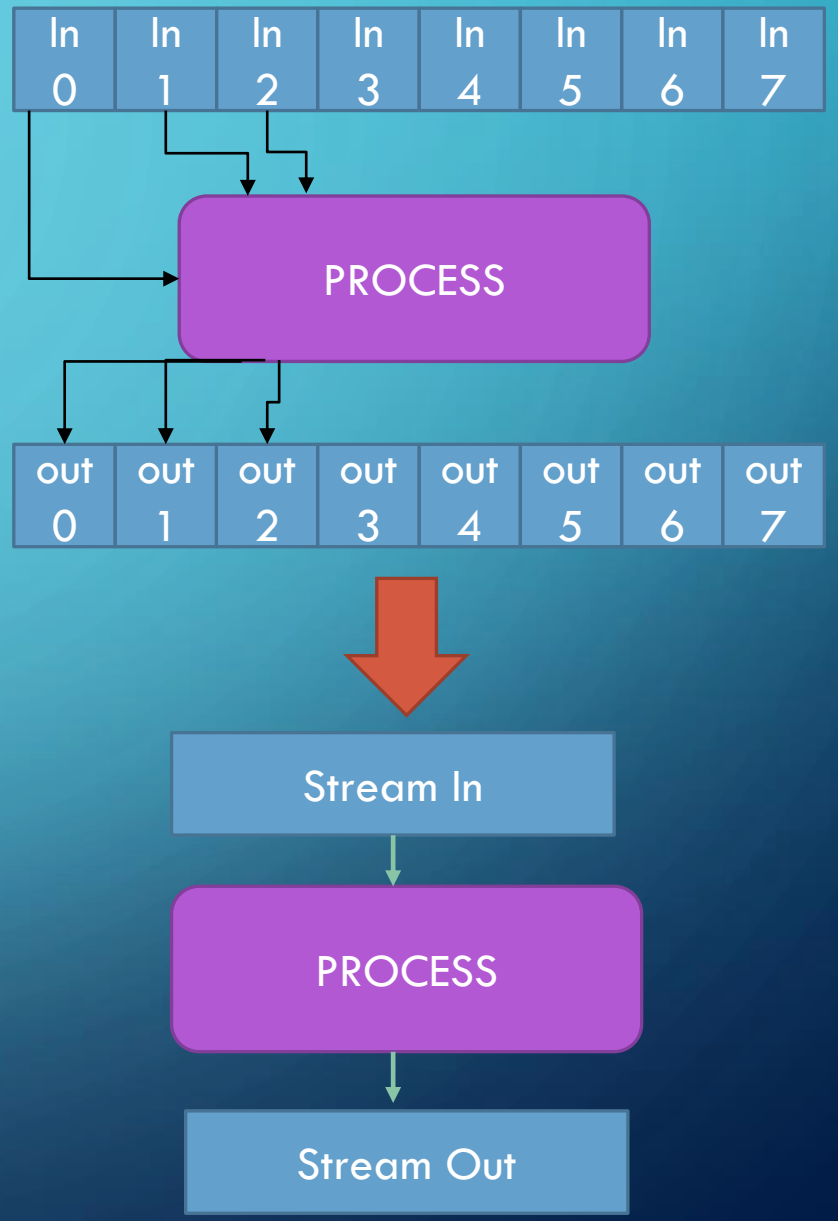
UNROLL

- #pragma HLS unroll factor=<number>
- Unrolling converts a loop to a sequence of instructions
- Utilises more space on the board
- Saves time on loop overheads
- Complete unroll requires loop iterations to be known at compile time



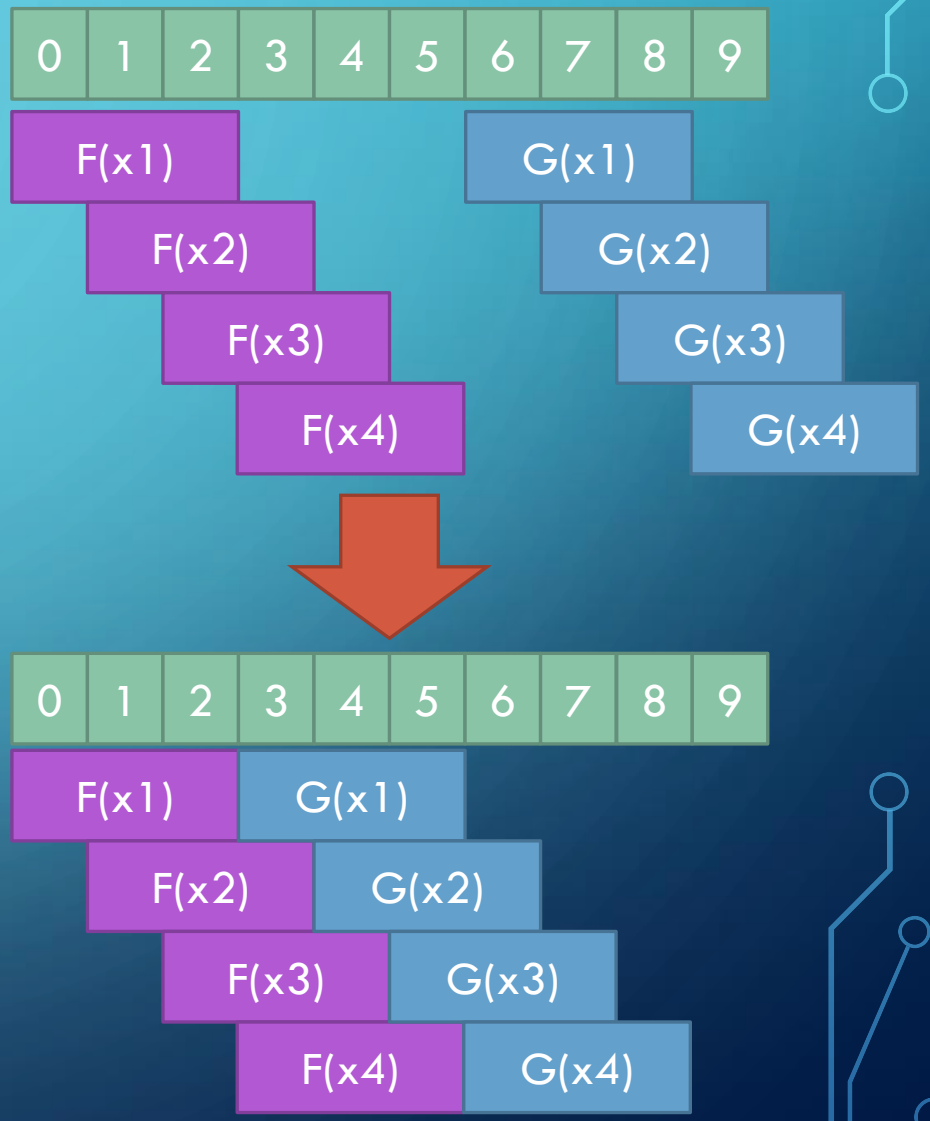
STREAM

- #pragma HLS stream variable=<variable name> type=<type> depth=<number>
- Streams are usually FIFO data structures
- FIFO places restrictions on data access patterns to which one must adhere
- Can save significant memory space
- Streaming is quicker than accessing RAM
- Can also use declare variables of the type hls::stream<datatype> and use read() / write()
- Use hls::stream to enforce semantics



DATAFLOW

- #pragma HLS DATAFLOW
- Task level parallelism
- Allows functions or loops to overlap / execute in parallel
- Vitis compiler will attempt to analyse the program for dependencies
- Dataflow regions allow for the use of efficient streaming but require a single-producer-consumer model



EXAMPLES AND EXERCISES

- You can find examples and exercises in HLS_examples
- Each folder contains
  - An example kernel (in a .cpp file) which can be improved by the use of the relevant pragma. The top level function will always have kernel in the name.
  - A markdown file which describes the pragma(s) used and the problem to be solved
- Use the C Synthesis in the Vitis HLS application to analyse the examples and your solutions
- Remember to use features like the schedule viewer to see how the solution changes when you make your alterations!