WRITING FPGA KERNELS

GETTING STARTED WITH FPGA KERNEL WRITING

STRUCTURE OF AN FPGA KERNEL

- Kernel top level functions require extern “C”{} declaration to prevent name mangling
- Top level function should be void
- Outputs should be declared in the arguments as pointers or refs
- Other functions that you call from within the code don’t have to behave this way

```
double func(double A, double B)
{
    return A + B;
}

extern "C"
{
    void kernel_top_level(double *input_A,
                          double *input_B,
                          double *out,
                          int N)
    {
        for(int i = 0; i < N; i++)
        {
            out[i] = func(input_A[i], input_B[i]);
        }
    }
}
```

ACCESSING GLOBAL MEMORY

- FPGA Global Memory is used for transferring memory between FPGA and CPU or between kernels on the FPGA
- Can store more data than inside the FPGA
- Double Data Rate RAM (DDR) or High Bandwidth Memory (HBM)
- Each DDR / HBM bank has separate ports
- Variables in global memory must be declared as arguments in top level function and have interfaces synthesised.
- We'll learn about interfaces later!

Global Memory: can be accessed by CPU or other kernels

```
void kernel_top_level(double *input_A,
                      double *input_B,
                      double *out,
                      int N)
{
    double local_C[1000];
```

Local memory – usually BRAM distributed throughout FPGA, not accessible to anything else

FPGA KERNELS IN C++

- FPGA kernels are hardware designs, and hardware description languages give best results
- In order to convert C/C++ to hardware description, restrictions apply
  - No function pointers, arbitrary pointer casting, or arrays of pointers to pointers
  - No recursion
  - No dynamic memory allocation
  - Many C/C++ library functions (look for alternative in Vitis libraries)
  - No system calls
  - Others!

SOME GOOD PRACTICES

- Use #define & typedef statements to make changing types and experimenting easier
- Give loops fixed bounds wherever possible
- Minimise accesses to global memory
- Try using multiple compute units to run things in parallel
- Think about how you want to pack your data
- Think about patterns: Load-Compute-Store / Singe-Producer-Consumer
- Label your loops!