COMPILATION AND CONFIGURATION

SOFTWARE EMULATION, HARDWARE EMULATION, AND HARDWARE

COMPILATION FOR FPGA

- Host code is compiled as usual, using e.g. g++
- Host code is compiled down to machine code
- FPGA kernel must be compiled to circuit design to be transferred to FPGA
- FPGA behaviour can be emulated on the CPU using software emulation and hardware emulation
- Compiling for software emulation is comparable to normal software
- Compiling for hardware emulation takes some minutes
- Compiling for hardware takes several hours for a typical kernel

SOFTWARE / HARDWARE EMULATION

- These are designed to approximate the behaviour of the FPGA on the CPU
- Software emulation
  - No hardware compilation or RTL hardware model
  - Compiled as regular C/C++ program
  - Can be used for validation of the C code
  - Simulates transfer on and off the FPGA
- Hardware emulation
  - Generates RTL and runs RTL simulation along with host side code
  - Much slower than software emulation – try to only use for small problem sizes!
  - Provides more accurate information about the possible behaviour of the hardware

COMPILATION PROCESS

- Two stage compile & link process
- Compile step comes first:
- Compile: `v++ -c -t <sw_emu/hw_emu/hw> --config <cfg_file> -k <kernel_name> -I<includes> <source> -o <output .xo>`

COMPILATION PROCESS

- Two stage compile & link process
- Specify the type of compilation: software emulation, hardware emulation, or hardware
- **Compile:** `v++ -c -t <sw_emu/hw_emu/hw> --config <cfg_file>  -k <kernel_name> -I<includes> <source> -o <output .xo>`

COMPILATION PROCESS

- Two stage compile & link process
- Give the compiler your config file – we'll see what goes in a moment!
- **Compile:** `v++ -c -t <sw_emu/hw_emu/hw> --config <cfg_file> -k <kernel_name> -I<includes> <source> -o <output .xo>`

COMPILATION PROCESS

- Two stage compile & link process
- Give the name of your kernel i.e. the top level function name (this is why we need extern “C” to prevent name mangling)
- **Compile:** `v++ -c -t <sw_emu/hw_emu/hw> --config <cfg_file> -k <kernel_name> -I<includes> <source> -o <output .xo>`

COMPILATION PROCESS

- Two stage compile & link process
- Give your include folders and source files as for normal C++ compilation
- **Compile:** `v++ -c -t <sw_emu/hw_emu/hw> --config <cfg_file>  -k <kernel_name> -I<includes> <source> -o <output .xo>`

COMPILATION PROCESS

- Two stage compile & link process
- The output of this process is a .xo file, which you can name as you please
- **Compile:** `v++ -c -t <sw_emu/hw_emu/hw> --config <cfg_file> -k <kernel_name> -I<includes> <source> -o <output .xo>`

COMPILATION PROCESS

- Two stage compile & link process
- An example compilation might be:

```
v++ -c -t sw_emu --config ../u280.cfg -k sum_kernel
-I../include ../kernel_src/sum_kernel.cpp -o sum.xo
```

COMPILATION PROCESS

- Two stage compile & link process
- The next step is linking
- Link: `v++ -l -t <sw_emu/hw_emu/hw> --config <cfg_file> <object .xo file> -o <output binary .xclbin>`

COMPILATION PROCESS

- Two stage compile & link process
- Take the target type and config as before
- Link: `v++ -l -t <sw_emu/hw_emu/hw> --config <cfg_file> <object .xo file> -o <output binary .xclbin>`

COMPILATION PROCESS

- Two stage compile & link process
- Now the .xo file is an input and the output is your kernel binary
- **Link:** `v++ -l -t <sw_emu/hw_emu/hw> --config <cfg_file> <object .xo file> -o <output binary .xclbin>`

COMPILATION PROCESS

- Two stage compile & link process
- An example linking command might be:

```
v++ -l -t sw_emu --config ../u280.cfg ./sum.xo -o
sum.xclbin
```

CONFIGURATION

- Declare target device type (here is it a Xilinx Alveo U280 board)
- Declare what kernels you want to compile for the device, how many copies you want, and how you want to refer to them
- Assign buffers to specific memory banks in DDR/HBM – important for maximising concurrent data accesses
- Profiling options allow you to monitor aspects of performance
- Can also tell add debug options and much more that we will not be getting into!

```
platform=xilinx_u280_xdma_201920_3
save-temps=1

[connectivity]
# Declare 1 instance of v_add_kernel called vak_1
nk=v_add_kernel:1:vak_1
# Assign A & B for vak_1 to different memory banks for parallel access
sp=vak_1.A:DDR[0]
sp=vak_1.B:DDR[1]
sp=vak_1.OUT:DDR[0]

[profile]
data=all:all:all  #monitor all instances of all kernels
~
~
~
```