HeteroSim User Guide

FENG, Liang

Department of Electrical and Computer Engineering
Hong Kong University of Science and Technology

April, 13th, 2016

User Guide Version Number: 1.0
HeteroSim Release Version: 1.0
Contact Email ID: lfengad@ust.hk
Project Page: http://www.ece.ust.hk/ eeweiz/projects.html

There are some guidelines to use HeteroSim simulator. *heterosim* is the home folder of HeteroSim simulator, which has three sub-folders: $legup - 3.0$,$m2s$ and $verilator - 3.880$. $legup - 3.0$ is modified LegUp environment handling the compilation flow. $m2s$ is the modified Multi2Sim environment handling the main part of simulation. $verilator - 3.880$ includes the files needed to use Verilator. LegUp, Multi2Sim and Verilator should be installed separately in their respective folders. Command *./install.sh* can help user to complete the installation for the whole HeteroSim automatically.

1 Main Executable Generation

1.1 Specification for Accelerated Functions

In the source code, the arguments of an accelerated function should have the following format $int(int*, int, int*, int)$. First argument is the pointer to the base address of input data. The second argument is the size of input data in bytes. The third argument represents the pointer to the base address of output data and the last argument means the size of the output data in bytes. The function would generate the output based on the input.
For example, $result = float64_add(x1, x2)$; is used to calculate the addition for two 8 bytes floating point numbers $x1$ and $x2$ using function $float64\_add$. When $float64\_add$ is accelerated on FPGA, all the input data should be on a continuous memory region, such as using $x[0]$ and $x[1]$ to replace the inputs $x1$ and $x2$ here. Then the base address for input data is at $\&x[0]$ and the size of input data is totally 16 bytes. Also, the base address for output data is at $\&result$ and the size of output is 8 bytes. Following the format specified in last paragraph, $a = float64_add(\&x[0], size1, \&result, size2)$ should be used here to do the floating point addition on FPGA with $size1$ as 16 and $size2$ as 8. More details about the format of the accelerated functions can be found in the example C codes in each sub-folder of $heterosim/legup - 3.0/examples/chstone/$.

1.2 Compilation for Main Executable

In the folder for an application such as $heterosim/legup-3.0/examples/chstone/str/$, there are C files for the application, the declaration file of accelerated functions $config.tcl$ and $Makefile$. The C file with main function should be specified as $NAME$ in $Makefile$. All the accelerated functions need be declared in $config.tcl$ with the form of $set\_accelerator\_function$. With commands *make hybrid* in the folder, the main executable will be generated in elf form which will be simulated.

2 Verilator Executable Generation

A Verilator executable is needed to simulate the kernel. To generate the Verilator executable, a C++ wrapper needs to be provided with the kernel description

```
while (fgets(line, sizeof(line), input))
{if (i < ARRAY_SIZE) {
sscanf(line, "%llx", &buff[i]);}
else {
printf("Invalid input for %s\n",KERNEL_NAME);
exit(1);    }
i++;}
```

Figure 1: code block 1: input data read part

```
if (!top->ap_done) {
VL_PRINTF ("A Test failed\n");
abort();
} else {
fprintf(output, "%d\n", (main_time - 10)/2);
for (i = 0; i < array_size; i++) {
fprintf(output, "%x\n", buff[i]);}}
```

Figure 2: code block 2: output data write part

in Verilog.

2.1 C++ Wrapper Specification

The example of the C++ wrapper is as *sim_main.cpp* in each sub-folder of $heterosim/verilator-3.880/VeriBench$. Take $heterosim/verilator-3.880/$ $VeriBench/verilog\_dfadd$ as example. The top module of the design is $float64$ $\_add.v$, thus in the wrapper *sim_main.cpp*, there should be two sentences $\#include$ "$Vfloat64\_add.h$" and $Vfloat64\_add * top$; to instantiate the C++ class for the kernel $Vfloat64$ $\_add$ as *top*. *top* here just represents the kernel.

The code block in figure 1 is needed at the beginning of the wrapper to read the input data from an input file generated by modified Multi2Sim part to the data buffer variables in the wrapper such as $buff$. The data buffer variables should correspond to each kind of input data kernel needed. The data pattern in the input file corresponds with the data pattern in the input memory region starting at the first argument of the accelerated function. Therefore, when we read the input data from the input file to the data buffer variables, we should follow this pattern to guarantee the consistency. The code block in figure 2 is needed at the end of the wrapper to write the execution time number and the generated output data to an output file to be used by modified Multi2Sim part. When writing the output file, first the kernel execution time number should be written, then the generated output data should be written from the data buffer variables such as $buff$ here to the output file following the data pattern in the output memory region starting at the third argument of the accelerated function.

Since the kernel specified by $float64\_add.v$ need interface to access the data in the peripheral buffer or accept some input signals such as *reset*. We should mimic the timing of the kernel interface. The example code of interface timing control part in the wrapper is as figure 3. *main_time* indicates the execution time passing. $while(!top- > ap_done\&\&!Verilated :: gotFinish())$ here checks whether the kernel execution is done or not. Only when the kernel execution hasn't been done, this code block will be executed to mimic the operations at the interface. The statement inside $if(top- > ap_clk)$ specifies the operations on the interface at the rising edge of clock while the statement inside $if(!top- > ap_clk)$ specifies the operations on the interface at the falling edge of clock. For example, here inside $if(top- > ap_clk)$, array $buff$ will store data as in $buff[buff_address0] = buff_d0$; or send data out to kernel as in $top- > buff_q0 = buff[buff_address0]$; when related enabling signals are set ,which is just the behavior between kernel and peripheral data buffer at the rising edge. The clock signal generation and the execution time increment should be within each iteration of while as the last two sentences in code block 3. Writing the code of such interface timing control part is similar to writing a testbench for the instantiated kernel design in a C++ class format. Interface operations at each clock edge, finish signal checking, stimulus including clock generation and execution time measure should be specified in such interface timing control part code as we talked above. For different kernel design, the code of this interface timing control part is different due to the different interfaces, which demands users to write the code according to specific kernel interface.

Following the instructions above, three parts of code: input data read, output data write and interface timing control should be included in the C++ wrapper. More details can be found in the example wrapper *sim_main.cpp* in each sub-folder of $heterosim/verilator-3.880/VeriBench$. The wrapper's logic depends on the specific verilog file of the kernel and is similar to a testbench.

2.2 Verilator Compilation

With C++ wrapper *sim_main.cpp* and the verilog files for a kernel, we can compile them to generate a Verilator executable. Take the kernel in $heterosim/verilator - 3.880/VeriBench/verilog\_dfadd$ as example. In the folder with *sim_main.cpp* and $float64\_add.v$, type following commands:

```
verilator -Wno-WIDTH --cc float64_add.v -exe sim_main.cpp
cd obj_dir
make -j -f Vfloat64_add.mk Vfloat64_add
```

Then, we can obtain the Verilator executable $Vfloat64\_add$ in *obj_dir*, which will be used during simulation.

```
while (!top->ap_done && !Verilated::gotFinish()) {
if (main_time > 10) {
top->ap_rst = 0;
top->ap_start = 1;
} else if (main_time > 1) {
top->ap_rst = 1;          }
top->eval();
if(top->ap_clk){
if(buff_ce0==1&&!buff_we0)top->buff_q0=buff[buff_address0];
if(buff_ce0==1&&buff_we0)buff[buff_address0]=buff_d0;
if(buff_ce1==1&&!buff_we1)top->buff_q1=buff[buff_address1];
if(buff_ce1==1&&buff_we1)buff[buff_address1]=buff_d1;}
if(!top->ap_clk){
buff_ce0 = top->buff_ce0;
buff_we0 = top->buff_we0;
buff_q0 = top->buff_q0;
buff_d0 = top->buff_d0;
buff_address0 = top->buff_address0;
buff_ce1 = top->buff_ce1;
buff_we1 = top->buff_we1;
buff_q1 = top->buff_q1;
buff_d1 = top->buff_d1;
buff_address1 = top->buff_address1;}
top->ap_clk = !top->ap_clk;
main_time++;                // Time passes...}
```

Figure 3: code block 3:interface timing control part

3 Performing Simulation

To perform the simulation, there should be the main executables, Verilator executables for each kernel needed and 4 configuration files in the folder such as the folder *heterosim/m2s/samples/memory/example* − 1.
The x86 configuration file such as $x86-config$ in the folder is used to specify the number of cores and number of threads on each core as in original Multi2Sim.The option $--x86-config$ for command $m2s$ is used to specify this configuration file.
The memory configuration file such as $mem-config$ in the folder is used to specify the memory hierarchy as in original Multi2Sim. In addition, it specifies the parameters for the AXI Bus connection, the connection latency between memory modules and the connection mode between FPGA and memory hierarchy. The option $--mem-config$ for command $m2s$ is used to specify this configuration file.
The kernel configuration file such as $dfadd-config$ in the folder is used to specify the register addresses, Verilator executable and the format of the input/output data for each kernel the simulation needs. And each section corresponds to each kernel with a kernel number.The option $--kernel-config$ for command $m2s$ is used to specify this configuration file.
The context configuration file such as $ctx-config$ in the folder specifies the main executables need to be simulated, the kernel number in the kernel configuration file of the kernels needed, etc. The option $--ctx-config$ for command $m2s$ is used to specify this configuration file.
Then with these four configuration files to set the system and the executables needed. Use following commands, the simulation will complete with basic performance metrics shown on the screen:

```
m2s --fpga-sim detailed --x86-sim detailed --x86-config
x86-config --mem-config mem-config --kernel-config dfadd-
config --ctx-config ctx-config
```

Other metrics can be obtained adding some options for command $m2s$, which are the same as the original Multi2Sim.
For further questions, please refer to the detailed files inside the simulator folders or contact us through e-mail. Further releases of the simulator and user guide with more advanced and improved features will be continued.