There are some guidelines to use HeteroSim simulator. There are four folders in $heterosim$: $legup-3.0$,$m2s$,$verilator-3.880$. $legup-3.0$ 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. The installations should be done in these three folders following the installation steps as in LegUp, Multi2Sim and Verilator respectively.

1 Main Executable Generation

1.1 Specification for Accelerated Functions

In the source code, the arguments of an accelerated function should have the following form $int(int*, int, int*, int)$. 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.

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 need be provided with the kernel description 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 $Vfloat64\_add$ as *top*.

```
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++;}
```

The above statement is needed at the beginning of the wrapper to load the input data from a input file generated by modified Multi2Sim part to the data buffer variables in the wrapper such as $buff$. The data buffers should correspond to each kind of input data kernel needed and the data format in the input file corresponding with the data pattern starting from the first argument of the accelerated function. The following statement is needed at the end of the wrapper to store the execution time number with generated output data from the buffer variables such as $buff$ to an output file to be used by modified Multi2Sim part. The data pattern for the output file should be the execution time number followed by the data format starting at the third argument of the accelerated function.

```
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]);}}
```

Since the kernel specified by $float64\_add.v$ need interface to access the data in the periphal 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 in the wrapper is as following. *main_time* indicates the execution time passing. $while(!top- > ap_{d}one\&\&!Verilated :: gotFinish())$ here checks whether the kernel execution is done or not. The statement inside $if(top- > ap_{c}lk)$ specifies the operations on the interface at the rising edge of clock while the statement inside $if(!top- > ap_{c}lk)$ specifies the operations on the interface at the falling edge of clock. The clock signal generation and the execution time increment should be within each iteration of while as the last two sentences here. This part of statement is similar to writing a testbench for the instantiated kernel design in a C++ class form. Interface operations at each clock edge, finish signal checking, stimulus including clock generation and execution time measure should be specified in this part of statement. For different kernel design, the logic of this part is different which depends on the verilog of the kernel.

```
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...}
```

Following the instructions above, three parts of statement: input data load, output data store and interface time 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 *float*64*_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 *Vfloat*64*_add* in *obj_dir*, which will be used during simulation.

3 Simulation Performing

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/m*2*s/samples/memory/example* − 1.
The x86 configuration file such as *x*86−*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. And to this point, the simulation can be done.