FlexGrip User Guide

Contents

1. Introduction

FlexGrip (FLEXible GRaphIcs Processor for general-purpose computing), is a fully CUDA binary-compatible integer GPGPU which has been optimized for FPGA implementation. The amount of parallelism is customizable at multiple levels including the number of parallel operations per instruction (processors) per multiprocessor. The hardware can be used for numerous CUDA programs without hardware resynthesis. FlexGrip has been designed based on the NVIDIA G80 architecture with compute capability, version 1.0. The architecture has been implemented in VHDL for a variety of parameters and evaluated in hardware using an ML605 Virtex-6 FPGA platform which includes DRAM. The RTL code supporting FlexGrip has been written to allow the design to be quickly customized for a variety of FPGA devices. A total of five CUDA benchmarks have been directly compiled to the architecture using standard NVIDIA compiler products [1]. This user guide will help you in setting up FlexGrip and running simulations. The user guide also talks about how to add a new CUDA application to FlexGrip.

More information on FlexGrip can be found in the paper http://www.ecs.umass.edu/ece/tessier/andryc-fpt13.pdf.

2. System Requirements

1. Microsoft Windows 7
2. Xilinx ISE 14.7
3. Mentor Graphics ModelSim 10.0 SE
4. nVidia cuda Toolkit 4.0 64-bit (https://developer.nvidia.com/cuda-toolkit-40)
5. Microsoft Visual C++ 2010 Express (http://www.visualstudio.com/en-us/downloads#d-2010-express)
6. Java 7 (https://www.java.com/en/download/)
7. Msys (http://www.mingw.org/wiki/MSYS)

NOTE: Xilinx ISE 14.7 and Mentor Graphics Modelsim 10.0 SE require a license key and is available for download from their respective websites.

3. Getting Started with FlexGrip

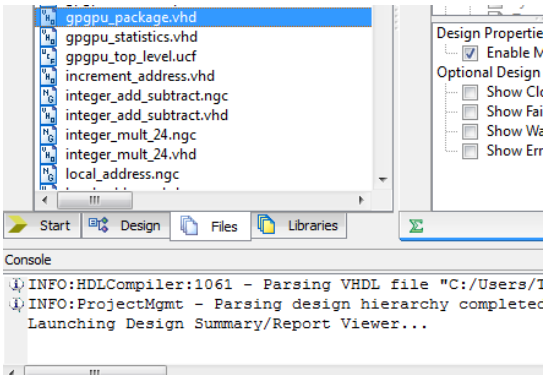
In this section, we will take you through the basic steps required to setup FlexGrip to run simulations.

1. Download FlexGrip.rar from FlexGrip website & untar the data.
2. Start Xilinx ISE.
3. In ISE, select "Open Project" and browse to \FlexGrip\flexgrip-gpgpu\virtex6.
4. Select gpgpu_virtex6.xise and click "Open".

3.1 Defining the Number of Cores

Firstly we need to define the number of cores. FlexGrip allows you to simulate 8, 16 or 32 cores.

1. In ISE, click the "Files" tab located on the left side of the pane above the "Console" as shown below.



2. In the "Name" pane, double click "gpgpu_package.vhd". It should open in the source pane.

3. There are two parameters that need to be adjusted when changing the number of cores, CORES and WARP_LANES. CORES is simply 8, 16, or 32. WARP_LANES is determined by following formula:

   WARP_LANES = WARP_SIZE / CORES where WARP_SIZE is set to 32

4. The number of cores is changed with the following definition:

```
--
-- Constants
--
constant CORES                          : integer := 8;
```

5. The number of lanes is changed with the following definition:

```
--
-- Constants
--
...
constant WARP_LANES                     : integer := 4;
```

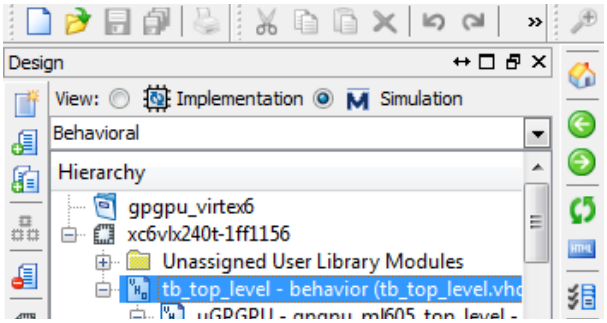
NOTE: You must change both the CORES AND WARP_LANE to their correct values as per the formula.

6. Save the file.

3.2 Setting up the Application

Here we define the application for which we need to run the simulations.

1. In ISE, select "Simulation" in the "Design" panel as shown below.



2. Double click "tb_top_level" to open up the test bench file.
3. In the source pane, scroll down to the instantiation of "uTestBenchConfig", you should see something like the following,

```
uTestBenchConfig : tb_configuration
   generic map (
      APPLICATION             => "REDUCTION"
   )
```

Change the APPLICATION to one of the following
   a. "AUTOCORR"
   b. "BITONIC_SORT"
   c. "MATRIX_MULT"
   d. "REDUCTION"
   e. "TRANSPOSE"

4. Next, scroll down to "uWriteInstructions". You should see something like the following,

```
uWriteInstructions : write_instructions
   generic map (
      APPLICATION          => "REDUCTION",
      MEM_ADDR_SIZE        => SYSMEM_ADDR_SIZE
    )
```

Change the APPLICATION to one of the following (NOTE: it should match the one you selected in 3).
   a. "AUTOCORR"
   b. "BITONIC_SORT"
   c. "MATRIX_MULT_SASS"
   d. "TRANSPOSE"
   e. "REDUCTION"

3.3 Generating Sample Input Data

Once you have selected the application to simulate, you next need to generate some sample data.

1. Open “gpgputoolbox.jar” from FlexGrip folder. The jar file generates sample data for various applications.
2. Select the "Generate Data" tab.
3. From the dropdown menu, select the application. Again, it should be the same one selected in section 3.2 steps (3) and (4) above.
4. Select the size of the dataset to generate (i.e. 8, 16, 32, 64, 128, 256 for vector applications and 8x8, 16x16, 32x32, 64x64, 128x128, and 256x256 for matrix applications).
5. Click "Generate".
6. In the folder you ran the gpgputoolbox.jar application from, copy the file named "init_global.coe" to the \FlexGrip\flexgrip-gpgpu\cores\virtex6\ram_8x262144 folder. If the file exists, overwrite it.
7. We now need to place the data found in Coe file into memory (essentially we are initializing the block memory with the data we generated).

    a. In the ISE project, inside the "Hierarchy" panel on the left, click the "+" sign next to "tb_top_level" to expand it down.
    b. Click the "+" next to “uGPGPU” if it has not been expanded.
    c. Double click on “uGlobalMemory” (there should be a light bulb next to it).
    d. The "Block Memory Generator" window should appear. Perform the following steps:
        i. Click Next
        ii. Click Next
        iii. Click Next
        iv. You should now be on Page 4 of 6. In "Memory Initialization" pane, ensure the "Load Init File" checkbox is CHECKED, and in the Coe File text box, ensure it says (without the quotes): ".\init_global.coe"
        v. Click "Generate" (NOTE: If a warning box opens asking about overwriting, click "Yes")
        vi. When complete, you should see a message in the ISE "Console":

```
Wrote CGP file for project 'ram_8x262144'.
Core Generator edit command completed successfully.
```

NOTE: If there is an error, try to re-run from Step 7c.

3.4 Configuring the Application

In order to tell FlexGrip the size of the application to execute (block and grid size), the size of the data and the address to input and output data, we must modify the configuration files.

a. In ISE inside the "Hierarchy" panel on the left, click the "+" sign next to "tb_top_level" to expand it down.
b. Click the "+" next to “uTestBenchConfig” if it has not been expanded.
c. Perform the following for the application chosen to simulate:
    i. If you selected the "AUTOCORR" application, double click “uAutocorConfiguration” so that the source shows in the source pane.
        1. First, we need to change the address of where the data and results are stored and the size of the array. Scroll down until you see the kernel register memory map, as shown below:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
            (--  15(0F)
             x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
             x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
             x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
             x"00000000",x"00000100",x"00000400",x"00000000"); -- 0

             -- addr 0: start address of data
             -- addr 1: start address of results
             -- addr 2: size of array
```

The start address of data doesn't need to change. The start address of the results need to change depending on the size of the data chosen in Section 3.3. Since the data we are using are integers, we

must multiply the data size by 4. The size is entered in hexadecimal; hence you should change it to one of the following:

8 * 4 = 32 => 0x00000020
16 * 4 = 64 => 0x00000040
32 * 4 = 128 => 0x00000080
64 * 4 = 256 => 0x00000100
128 * 4 = 512 => 0x00000200
256 * 4 = 1,024 => 0x00000400

Change it in the location highlighted below:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
           (--  15(0F)
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
            x"00000000",x"00000100",x"00000400",x"00000000"); -- 0

            -- addr 0: start address of data
            -- addr 1: start address of results
            -- addr 2: size of array
```

Next, we define the size of the array and change it in addr 2. Using the size again chosen in Section 3.3, change it to one of the following:

8 => 0x00000008
16 => 0x00000010
32 => 0x00000020
64 => 0x00000040
128 => 0x00000080
256 => 0x00000100

Change it in the location highlighted below:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
           (--  15(0F)
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
            x"00000000",x"00000100",x"00000400",x"00000000"); -- 0

            -- addr 0: start address of data
            -- addr 1: start address of results
            -- addr 2: size of array
```

2. Next, we need to define the kernel block and grid sizes. Based on our selection of data sizes, we will only change the Block X and Grid X sizes. Scroll down until you see the following,

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010"; -- 16
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0001"; -- 1
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001"; -- 1
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0010"; -- 1
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0001"; -- 1
```

Change the sizes as follows:

a. For a data size of 8:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0008";
…
…
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0001";
```

b. For a data size of 16:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
…
…
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0001";
```

c. For a data size of 32:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
…
…
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0002";
```

d. For a data size of 64:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
…
…
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0004";
```

e. For a data size of 128:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
…
…
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0008";
```

f. For a data size of 256:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
…
…
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0010"
```

ii. If you selected the "BITONIC_SORT" application, double click “uBitonicSortConfiguration” so that the source shows in the source pane.

1. In this application, the application uses the input address as the destination address of the data as well. Therefore, in this case, the input data and output data address are 0x00000000. The kernel registers should all be zero as shown below:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
           (--  15(0F)
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
            x"00000000",x"00000000",x"00000000",x"00000000"); -- 0

            -- addr 0: start address of data
```

2. We next define the size of the array and change it in the constant memory registers. Scroll to find the following code:

```
signal cmem_regs                : cmem_regs_type;
    constant cmem_regs_default      : cmem_regs_type :=
           (--  63(3F)
            x"00",x"00",x"00",x"00",  -- 60
            x"00",x"00",x"00",x"00",  -- 56
            x"00",x"00",x"00",x"00",  -- 52
            x"00",x"00",x"00",x"00",  -- 48
            x"00",x"00",x"00",x"00",  -- 44
            x"00",x"00",x"00",x"00",  -- 40
            x"00",x"00",x"00",x"00",  -- 36
            x"00",x"00",x"00",x"00",  -- 32
            x"00",x"00",x"00",x"00",  -- 28
            x"00",x"00",x"00",x"00",  -- 24
            x"00",x"00",x"00",x"00",  -- 20
```

```
        x"00",x"00",x"00",x"00",  -- 16
        x"00",x"00",x"00",x"00",  -- 12
        x"00",x"00",x"00",x"00",  -- 8
        x"00",x"00",x"00",x"00",  -- 4
        x"00",x"00",x"01",x"00"); -- 0
```

The size of the array will be stored in memory locations 0x1 to 0x0. The size is stored such that the MSB is stored in the higher address. For example, if the array size is 256, then you would set address 1 to x"01" and address 0 to x"00". Using the size again chosen in Section 3.3, change the array size to one of the following:

a. For a data size of 8:

```
constant cmem_regs_default       : cmem_regs_type :=
     (--  63(3F)
             …
             ...
    x"00",x"00",x"00",x"00",  -- 4
    x"00",x"00",x"00",x"08"); -- 0
```

b. For a data size of 16:

```
constant cmem_regs_default       : cmem_regs_type :=
     (--  63(3F)
             …
             ...
    x"00",x"00",x"00",x"00",  -- 4
    x"00",x"00",x"00",x"10"); -- 0
```

c. For a data size of 32:

```
constant cmem_regs_default       : cmem_regs_type :=
     (--  63(3F)
             …
             ...
   x"00",x"00",x"00",x"00",  -- 4
   x"00",x"00",x"00",x"20"); -- 0
```

d. For a data size of 64:

```
constant cmem_regs_default       : cmem_regs_type :=
     (--  63(3F)
     …
     …
    x"00",x"00",x"00",x"00",  -- 4
    x"00",x"00",x"00",x"40"); -- 0
```

e. For a data size of 128:

```
constant cmem_regs_default       : cmem_regs_type :=
     (--  63(3F)
             …
             ...
    x"00",x"00",x"00",x"00",  -- 4
    x"00",x"00",x"00",x"80"); -- 0
```

f. For a data size of 256:

```
constant cmem_regs_default       : cmem_regs_type :=
     (--  63(3F)
             …
             ...
    x"00",x"00",x"00",x"00",  -- 4
    x"00",x"00",x"01",x"00"); -- 0
```

3. Next, we need to define the kernel block and grid sizes. Based on our selection of data sizes, we will only change the Block X and Grid X sizes. Scroll down until you see the following,

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010"; -- 16
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0001"; -- 1
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001"; -- 1
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0010"; -- 1
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0001"; -- 1
```

Change the sizes as follows:

a. For a data size of 8:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0008";
…
…
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0001";
```

b. For a data size of 16:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
…
…
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0001";
```

c. For a data size of 32:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
…
…
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0002";
```

d. For a data size of 64:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
…
…
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0004";
```

e. For a data size of 128:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
…
…
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0008";
```

f. For a data size of 256:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
…
…
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0010"
```

iii. If you selected the "MATRIX_MULT" application, double click “uMatrixMultConfiguration” so that the source shows in the source pane.

1. First, we need to change the address of the where the data is stored and the size of the array. Scroll down until you see the kernel register memory map, as shown below:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
           (--  15(0F)
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
            x"00000010",x"00000800",x"00000400",x"00000000"); -- 0

            -- addr 0: start address of data M
            -- addr 1: start address of data N
            -- addr 2: start address of results
            -- addr 3: size of array
```

The start address of data M doesn't need to change, it will always be 0. The start address of data N in addition to the results need to change depending on the size of the data chosen in Section 3.3 above. Since our input data is a 2D array of integers, we must multiply the array size together and then multiply it by 4. For example, if your array is 16x16, the, the start address of M would be 0 (0x00000000), the start address of N would be 16*16*4=1,024 (0x00000400), and the start address of the results is (16*16*4)*2=2,048 (0x00000800). The size of the array would be 16 (0x00000010).
Change the data in the highlighted are below:

a. For a data size of 8x8:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
       (--  15(0F)
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
        x"00000008",x"00000200",x"00000100",x"00000000"); -- 0
```

b. For a data size of 16x16:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
       (--  15(0F)
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
        x"00000010",x"00000800",x"00000400",x"00000000"); -- 0
```

c. For a data size of 32x32:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
       (--  15(0F)
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
        x"00000020",x"00002000",x"00001000",x"00000000"); -- 0
```

d. For a data size of 64:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
       (--  15(0F)
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
        x"00000040",x"00008000",x"00004000",x"00000000"); -- 0
```

e. For a data size of 128x128:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
       (--  15(0F)
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
        x"00000080",x"00020000",x"00010000",x"00000000"); -- 0
```

f. For a data size of 256x256:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
       (--  15(0F)
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
        x"00000100",x"00080000",x"00040000",x"00000000"); -- 0
```

2. Next, we need to define the kernel block and grid sizes. Scroll down until you see the following

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010"; -- 16
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0001"; -- 1
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001"; -- 1
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0010"; -- 1
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0001"; -- 1
```

Change the sizes as follows:

a. For a data size of 8x8:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0008";
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0008";
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0001";
```

b. For a data size of 16x16:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0010";
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0001";
```

c. For a data size of 32x32:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0010";
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0002";
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0002";
```

d. For a data size of 64x64:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0010";
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0004";
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0004";
```

e. For a data size of 128x128:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0010";
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0008";
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0008";
```

f. For a data size of 256x256:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0010";
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0010";
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0010";
```

iv. If you selected the "REDUCTION" application, double click “uReductionConfiguration” so that the source shows in the source pane.

1. In this application, the application uses the input address as the destination address of the data as well. Therefore, in this case, the input data and output data address are 0x00000000. The kernel registers should all be zero as shown below:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
           (--  15(0F)
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
            x"00000000",x"00000000",x"00000000",x"00000000"); -- 0
```

```
            -- addr 0: start address of data
            -- addr 1: start address of results
```

The start address of data doesn't need to change. The start address of the results need to change depending on the size of the data chosen in Section 3.3. Since the data we are using are integers, we must multiply the data size by 4. The size is entered in hexadecimal; hence you should change it to one of the following:

8 * 4 = 32	=> 0x00000020
16 * 4 = 64	=> 0x00000040
32 * 4 = 128	=> 0x00000080
64 * 4 = 256	=> 0x00000100
128 * 4 = 512	=> 0x00000200
256 * 4 = 1,024	=> 0x00000400

Change it in the location highlighted below:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
           (--  15(0F)
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
            x"00000000",x"00000100",x"00000400",x"00000000"); -- 0

            -- addr 0: start address of data
            -- addr 1: start address of results
```

2. Next, we need to define the kernel block and grid sizes. Based on our selection of data sizes, we will only change the Block X and Grid X sizes. Scroll down until you see the following,

```
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0001";  -- 1
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001";  -- 1
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0010";  -- 1
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0001";  -- 1
```

Change the sizes as follows:

a. For a data size of 8:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0008";
…
…
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0001";
```

b. For a data size of 16:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
…
…
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0001";
```

c. For a data size of 32:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
…
…
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0002";
```

d. For a data size of 64:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
…
…
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0004";
```

e. For a data size of 128:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
…
…
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0008";
```

f. For a data size of 256:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
…
…
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0010"
```

v. If you selected the "TRANSPOSE" application, double click “uTransposeConfiguration” so that the source shows in the source pane.

1. First, we need to change the address of the where the data is stored and the size of the array. Scroll down until you see the kernel register memory map, as shown below:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
           (--  15(0F)
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
            x"00000000",x"00000010",x"00000400",x"00000000"); -- 0

            -- addr 0: start address of data
            -- addr 1: start address of results
            -- addr 2: matrix width
```

The start address of data doesn't need to change, it will always be 0. The start address of the results need to change depending on the size of the data chosen in Section 3.3 above. Since our input data is a 2D array of integers, we must multiply the array size together and then multiply it by 4. For example, if your array is 16x16, the, the start address of data would be 0 (0x00000000), the start address of result would be 16*16*4=1,024 (0x00000400) and the matrix width would be 16 (0x00000010).
Change the data in the highlighted are below:

a. For a data size of 16x16:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
       (--  15(0F)
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
        x"00000000",x"00000010",x"00000400",x"00000000"); -- 0
```

b. For a data size of 32x32:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
       (--  15(0F)
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
        x"00000000",x"00000020",x"00001000",x"00000000"); -- 0
```

c. For a data size of 64:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
       (--  15(0F)
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
        x"00000000",x"00000040",x"00004000",x"00000000"); -- 0
```

d. For a data size of 128x128:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
       (--  15(0F)
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
        x"00000000",x"00000080",x"00010000",x"00000000"); -- 0
```

e. For a data size of 256x256:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
       (--  15(0F)
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
        x"00000000",x"00000100",x"00040000",x"00000000"); -- 0
```

2. Next, we need to define the kernel block and grid sizes. In this instance, we are using tiled blocks. Each block is 32x8 thread, with each thread transposing 4 elements, i.e. each block computes the transpose of a 32x32 tile. So for e.g. 64x64 matrix computation, schedule a grid of 2x2 blocks. Scroll down until you see the following

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0020"; -- 32
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0008"; -- 8
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001"; -- 1
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0010"; -- 1
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0001"; -- 1
```

Change the sizes as follows:

a. For a data size of 16x16:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0020";
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0008";
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0001";
```

b. For a data size of 32x32:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0020";
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0008";
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0002";
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0002";
```

c. For a data size of 64x64:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0020";
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0008";
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0004";
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0004";
```

d. For a data size of 128x128:

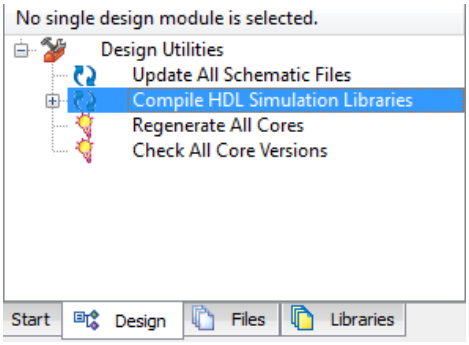
```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0020";
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0008";
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0008";
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0008";
```

e. For a data size of 256x256:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0020";
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0008";
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0010";
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0010";
```

3.5 Running Simulation

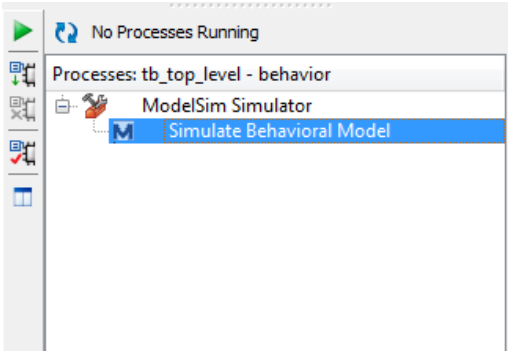
1. For our example, we will use the Matrix Multiply application with 8 cores. Follow the below steps,
   a. Change the number of Cores to 8 and Warp Lanes to 4 by following the steps in Section 3.1.
   b. In the "tb_top_level" file, change the APPLICATION variable in "uTestBenchConfig" to "MATRIX_MULT".
   c. In the "tb_top_level" file, change the APPLICATION variable in "uWriteInstructions" to "MATRIX_MULT_SASS".
   d. Generate a 16x16 Matrix Multiply dataset by following the steps in Section 3.3.
   e. Configure the configuration parameters by following steps in Section 3.4 for a 16x16 Matrix Multiply application
2. In ISE, click the "Simulation" radio button in the design pane.
3. In the "Hierarchy" pane, left-click once on “xc6vlx240t-1ff1156”. It will become highlighted in blue.
4. Now click on “+” button next to “Design Utilities” so that the tree expands.



5. RIGHT-CLICK on "Compile HDL Simulation Libraries" and select "Rerun All”.
6. Now ISE compiles the Simulation Libraries and when you are complete, you will get the following message,

```
Process "Compile HDL Simulation Libraries" completed successfully
```

7. In the "Hierarchy" pane, left-click once on "tb_top_level". It will become highlighted in blue.
8. In the "Processes: tb_top_level - behavior", click the "+" button next to "ModelSim Simulator" so that the tree expands.



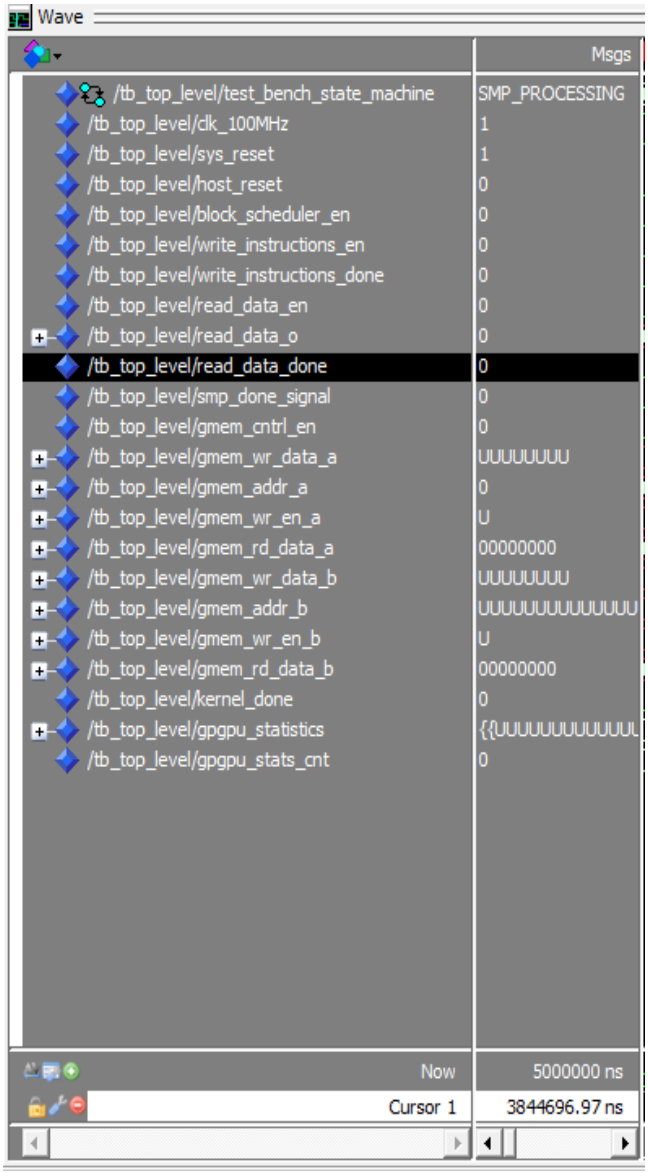
9. RIGHT-CLICK on "Simulate Behavioral Model" and select "Rerun All"
10. ModelSim will start and will show the following error:

```
# vsim -do \{do \{tb_top_level.fdo\}\}
# ** Note: (vsim-3812) Design is being optimized...
# ** Error: (vopt-19) Failed to access library 'work.{tb_top_level' at "work.{tb_top_level".
# No such file or directory. (errno = ENOENT)
# ** Error: Library work.{tb_top_level not found.
# Error loading design
```

11. In the ModelSim command line, type the following:

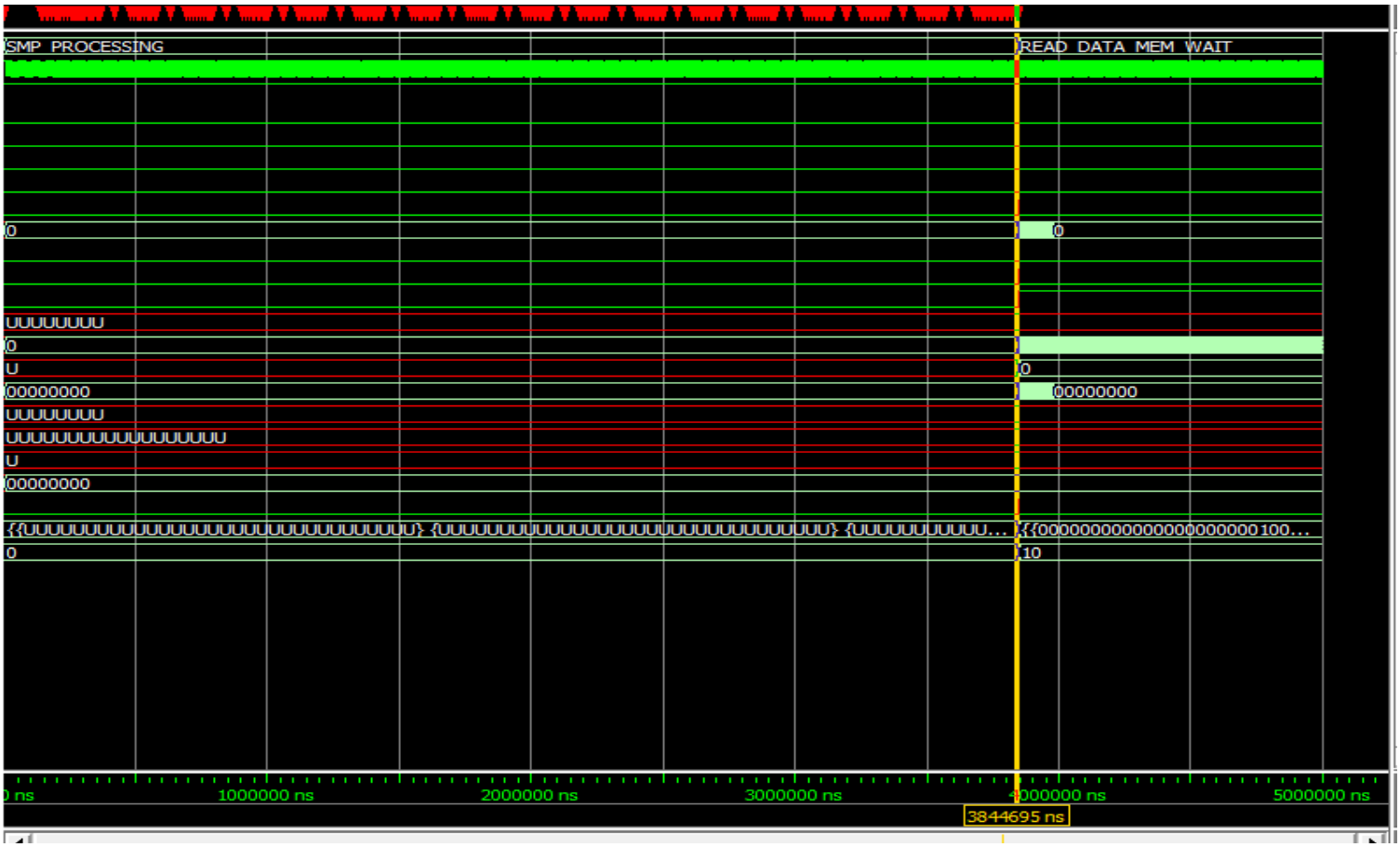
    ModelSim> do tb_top_level.fdo
    Hit <Enter>

12. At this point, ModelSim will compile and optimize the design (NOTE: If an error occurs due to it being not able to move a file, close ModelSim and rerun the simulation starting from Step (9) above.
13. Once complete, ModelSim will show with the waveform.
14. First, we will set up some signals to watch. We will use the existing signals defined and just remove the ones we don't want. You may add or keep as many as you like. Delete the signals from the "Wave" pane such that only the signals shown below are left:

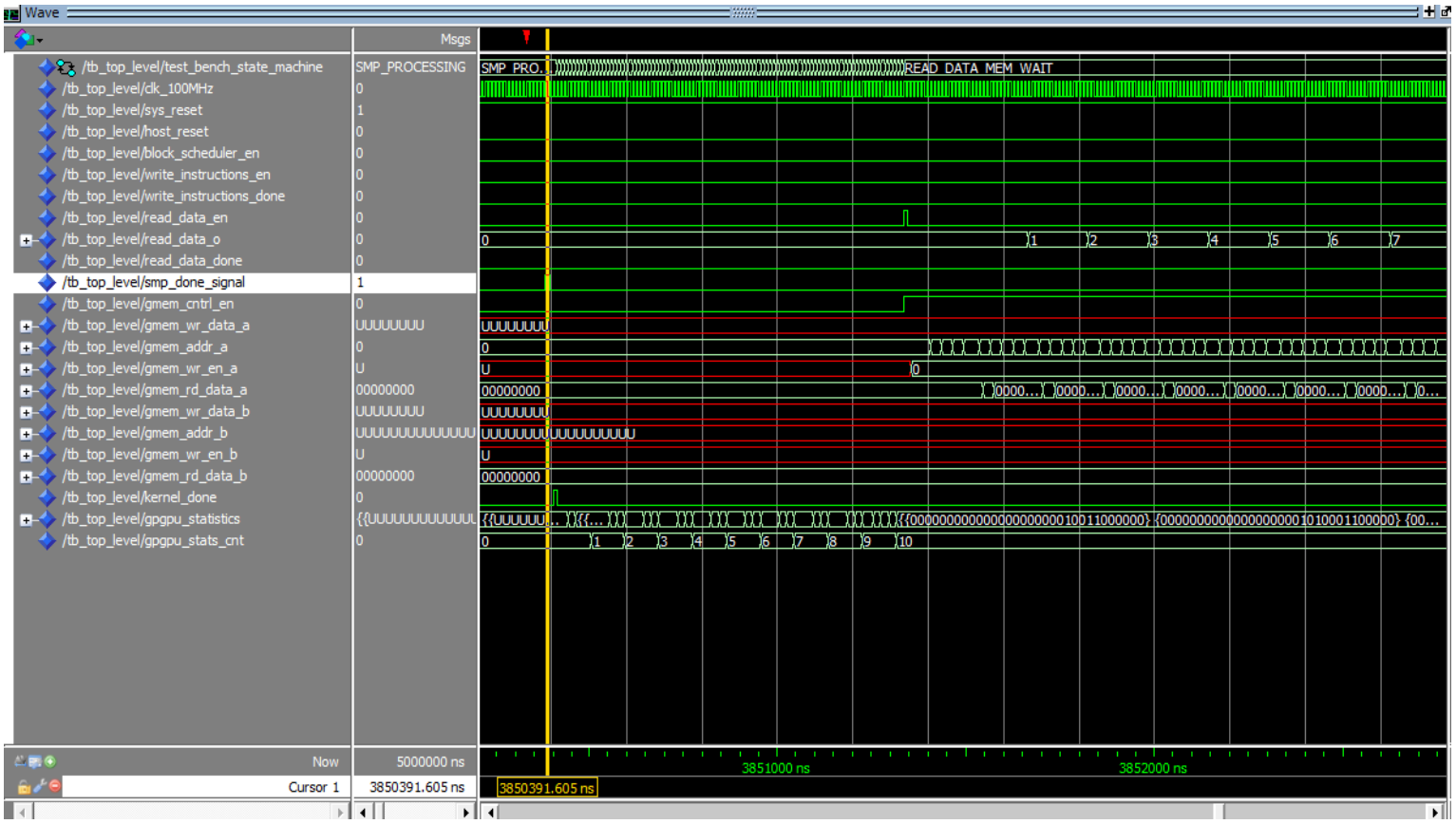


15. Next, we will have to define the length of time. For an 8 core Matrix Multiply, it takes approximately 3.5 ms. To be safe, enter "4 ms" into the "Run Length" text box located in the top toolbar of ModelSim.
16. In the toolbar to the left of the "Run Length", click the "Restart" button.
17. Next, In the toolbar to the right of the "Run Length", click the "Run" button.
    NOTE: If the simulation runs for several hours, you can turn-off the warning message in ModelSim
    (In ModelSim click on *Simulate* → *Runtime Options* → *Message Severity*)
18. Once the simulation completes, inside the Waveform, RIGHT click and select "Zoom Full".

19. Using your cursor in the waveform, move it over the /tb_top_level/test_bench_state_machine and LEFT click where the transition occurs from "SMP_PROCESSING" to "READ_DATA_MEM_WAIT" as shown below:

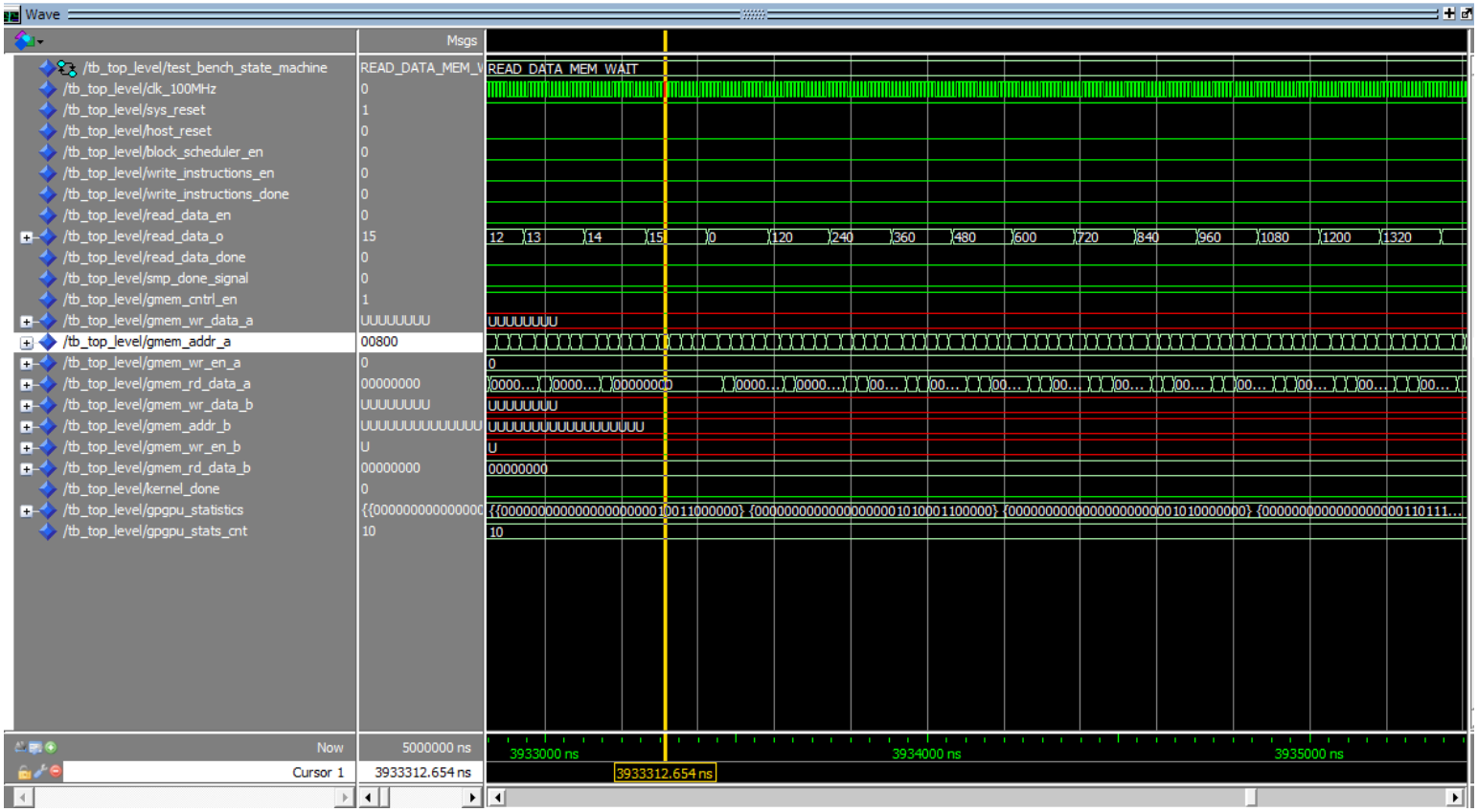


20. Press SHIFT+C to zoom in on the cursor location until it looks like the following (NOTE: You may have to move your cursor to have it centered properly):



21. After the GPGPU has completed processing, the test bench will read all the data from the global memory. We only care about the results. From Section 3.4 above, we know that the results start at address 0x800. To check the results, we first need to convert the signal outputs in the waveform from binary to hex/decimal. Perform the following steps:
    a. RIGHT click on the signal /tb_top_level/read_data_o and select Radix->Decimal
    b. RIGHT click on the signal /tb_top_level/gmem_addr_a and select Radix->Hexadecimal

22. Scroll the cursor to the right while watching the /tb_top_level/gmem_addr_a signal until the address is at 800 as shown below:



23. You will notice that the signal /tb_top_level/read_data_o has a value of 15, this is because the output data takes a couple of cycles before the data is read and displayed. However, if you continue progressing to the right, you will see the correct values for the first row in the Matrix Multiply application as shown in the /tb_top_level/read_data_o signal.

4. Loading a New Application into FlexGrip

In this section, we will demonstrate how to load a new CUDA application “VECTOR_ADD” into the FlexGrip.

4.1 Compiling CUDA code

1. The sample CUDA code “vectorAdd.cu” and “makefile” is available in the FlexGrip\NewApplication folder.
2. Open the makefile and modify the TARGET variable to point to the CUDA file path.
3. Open a Command prompt in Msys (C:\Msys\1.0\msys.bat) and go to the path where the “makefile” is saved.
4. Do “make all”
5. The makefile will generate a vectorAdd.sass file.

NOTE: If you get the following error

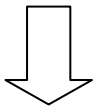
```
nvcc fatal : Cannot find compiler 'cl.exe' in PATH
```

Then you need to add the path of cl.exe (C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin) to the PATH variable in “Environment Variables” in Windows and rerun Step 4.

4.2 Writing the Instructions into FlexGrip

1. Open the “new_application_instructions.vhd” from the FlexGrip\NewApplication folder.
2. In the “new_application_instructions.vhd” file, replace new_application_instructions with vector_add_instructions and NUM_NEW_APPLICATION_INSTRUCTIONS with NUM_VECTOR_ADD_INSTRUCTIONS.
3. Now enter the instruction from the .sass file to “vector_add_instructions.vhd” file as shown in figure below. You can see from the figure that case 0 is simply the 1st 8 bits of the MOV.U16 instruction found in the .sass file and case 1 is the next 8 bits of the same instruction. Like this manner, enter all the binary instructions into the case statements. You may add more case statements if needed or delete the extra case statements if there are any.

```
code for sm_10
    Function : _Z9vectorAddPKfS0_Pfi
/*0000*/     /*0x100042050023c780*/     MOV.U16 R0H, g [0x1].U16;
/*0008*/     /*0xa000000504000780*/     I2I.U32.U16 R1, R0L;
/*0010*/     /*0x60014c0100204780*/     IMAD.U16 R0, g [0x6].U16, R0H, R1;
/*0018*/     /*0x3000cffd6c20c7c8*/     ISET.S32.C0 o [0x7f], g [0x7], R0, LE;
/*0020*/     /*0x3000000300000280*/     RET C0.NE;
/*0028*/     /*0x30020009c4100780*/     SHL R2, R0, 0x2;
/*0030*/     /*0x2102e800        */     IADD32 R0, g [0x4], R2;
/*0034*/     /*0x2102ea0c        */     IADD32 R3, g [0x5], R2;
/*0038*/     /*0xd00e000580c00780*/     GLD.U32 R1, global14 [R0];
/*0040*/     /*0xd00e060180c00780*/     GLD.U32 R0, global14 [R3];
/*0048*/     /*0xb0000204        */     FADD32 R1, R1, R0;
/*004c*/     /*0x2102ec00        */     IADD32 R0, g [0x6], R2;
/*0050*/     /*0xd00e0005a0c00781*/     GST.U32 global14 [R0], R1;
    ..................................
```



```
process(instruction_pointer_in)
begin
    case instruction_pointer_in is
            when  0 => instruction_out <= x"10004205";
            when  1 => instruction_out <= x"0023c780";
            when  2 => instruction_out <= x"a0000005";
            when  3 => instruction_out <= x"04000780";
            when  4 => instruction_out <= x"60014c01";
            when  5 => instruction_out <= x"00204780";
            when  6 => instruction_out <= x"3000cffd";
            when  7 => instruction_out <= x"6c20c7c8";
            when  8 => instruction_out <= x"30000003";
            when  9 => instruction_out <= x"00000280";
            when  10 => instruction_out <= x"30020009";
            when  11 => instruction_out <= x"c4100780";
            when  12 => instruction_out <= x"2102e800";
            when  13 => instruction_out <= x"2102ea0c";
            when  14 => instruction_out <= x"d00e0005";
            when  15 => instruction_out <= x"80c00780";
            when  16 => instruction_out <= x"d00e0601";
            when  17 => instruction_out <= x"80c00780";
            when  18 => instruction_out <= x"b0000204";
            when  19 => instruction_out <= x"2102ec00";
            when  20 => instruction_out <= x"d00e0005";
            when  21 => instruction_out <= x"a0c00781";
        when others => null;
    end case;
end process;
```

4. Save the file as “vector_add_instructions.vhd”.
5. In the ISE project, inside the "Hierarchy" panel on the left, left-click “uWriteInstructions” and select “Add Source”
6. Enter the path to “vector_add_instructions.vhd” file in the “File Name” space and click “Open”. Now you can see that this file has been added to your project hierarchy.

7. Now Double-click “uWriteInstructions” to open it in the source pane.
    a. Scroll down until you see something like this,

```
architecture arch of write_instructions is

    component matrix_mult_sass_instructions
        port (
            instruction_pointer_in  : in integer;

            num_instructions_out    : out integer;
            instruction_out         : out std_logic_vector(31 downto 0)
        );
```

    b. We need to add “vector_add_instructions” component to the architecture of this source file. You can add it as highlighted below.

```
architecture arch of write_instructions is

component vector_add_instructions
        port (
            instruction_pointer_in  : in integer;

            num_instructions_out    : out integer;
            instruction_out         : out std_logic_vector(31 downto 0)
        );
    end component;

    component matrix_mult_sass_instructions
        port (
            instruction_pointer_in  : in integer;

            num_instructions_out    : out integer;
            instruction_out         : out std_logic_vector(31 downto 0)
        );
```

    c. Now again scroll down until you see something like this

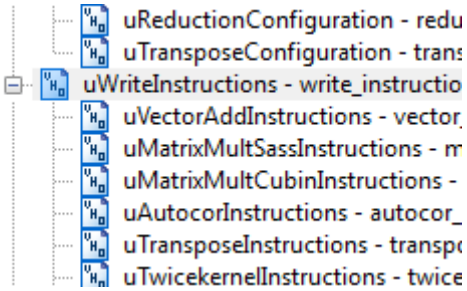
```
Begin

    gMATRIX_MULT_SASS: if APPLICATION = "MATRIX_MULT_SASS" generate
        uMatrixMultSassInstructions : matrix_mult_sass_instructions
            port map (
                instruction_pointer_in  => instruction_pointer,

                num_instructions_out    => max_instruction_length,
                instruction_out         => instruction_i
            );
    end generate;
```

    d. We need to specify the port map for the newly added component. Modify as highlighted below.

```
Begin

    gVECTOR_ADD: if APPLICATION = "VECTOR_ADD" generate
        uVectorAddInstructions : vector_add_instructions
            port map (
                instruction_pointer_in  => instruction_pointer,

                num_instructions_out    => max_instruction_length,
                instruction_out         => instruction_i
            );
    end generate;
```

```
gMATRIX_MULT_SASS: if APPLICATION = "MATRIX_MULT_SASS" generate
    uMatrixMultSassInstructions : matrix_mult_sass_instructions
        port map (
            instruction_pointer_in  => instruction_pointer,

            num_instructions_out    => max_instruction_length,
            instruction_out         => instruction_i
        );
end generate;
```

e. Save the file. You can see that “uVectorAddInstructions” gets added below the “uWriteInstrutions” hierarchy in the “Hierarchy” panel.



4.3 Setting the Configuration for the Application

1. Open the “new_application_configuration.vhd” from the FlexGrip\NewApplication folder.
2. In the “new_application_configuration.vhd” file, replace new_application_configuration with vector_add_configuration.
3. Now we need to modify the settings for the Application configuration.
    i. First, we need to change the address of the where the data is stored and the size of the array. Scroll down until you see the kernel register memory map, as shown below:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
           (--  15(0F)
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
            x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
            x"00000010",x"00000800",x"00000400",x"00000000"); -- 0

            -- addr 0: start address of data M
            -- addr 1: start address of data N
            -- addr 2: start address of results
            -- addr 3: size of array
```

The start address of data M doesn't need to change, it will always be 0. The start address of data N in addition to the results need to change depending on the size of the data chosen. Since our input data is an array of integers, we must multiply the array by 4. For example, if your array is 16, the, the start address of M would be 0 (0x00000000), the start address of N would be 16*4=64 (0x00000040), and the start address of the results is (16*4)*2=128 (0x00000080). The size of the array would be 16 (0x00000010).
Change the data in the highlighted are below:

a. For a data size of 8x8:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
       (--  15(0F)
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
        x"00000008",x"00000040",x"00000020",x"00000000"); -- 0
```

b. For a data size of 16x16:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
       (--  15(0F)
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
        x"00000010",x"00000080",x"00000040",x"00000000"); -- 0
```

c. For a data size of 32x32:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
       (--  15(0F)
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
        x"00000020",x"00000100",x"00000080",x"00000000"); -- 0
```

d. For a data size of 64:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
       (--  15(0F)
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
        x"00000040",x"00000200",x"00000100",x"00000000"); -- 0
```

e. For a data size of 128x128:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
       (--  15(0F)
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
        x"00000080",x"00000400",x"00000200",x"00000000"); -- 0
```

f. For a data size of 256x256:

```
signal kernel_regs              : kernel_regs_type;
    constant kernel_regs_default    : kernel_regs_type :=
       (--  15(0F)
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 12
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 8
        x"00000000",x"00000000",x"00000000",x"00000000",  -- 4
        x"00000100",x"00000800",x"00000400",x"00000000"); -- 0
```

ii. Next, we need to define the kernel block and grid sizes. Scroll down until you see the following

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010"; -- 16
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0001"; -- 1
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001"; -- 1
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0010"; -- 1
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0001"; -- 1
```

Change the sizes as follows:

a. For a data size of 8x8:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0008";
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0001";
```

b. For a data size of 16x16:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0001";
```

c. For a data size of 32x32:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0002";
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0001";
```

d. For a data size of 64x64:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0004";
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0001";
```

e. For a data size of 128x128:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0008";
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0001";
```

f. For a data size of 256x256:

```
constant kernel_block_x     : std_logic_vector(15 downto 0) := x"0010";
constant kernel_block_y     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_block_z     : std_logic_vector(15 downto 0) := x"0001";
constant kernel_grid_x      : std_logic_vector(15 downto 0) := x"0010";
constant kernel_grid_y      : std_logic_vector(15 downto 0) := x"0001";
```

iii. Next, we need to define the number of parameters. Scroll down until you see the following

```
constant kernel_param_size  : std_logic_vector(3 downto 0)  := x"4";  -- 4
```

We can see in the vectorAdd.cu file that there are 4 parameters for the kernel. Modify as highlighted below.

```
constant kernel_param_size  : std_logic_vector(3 downto 0)  := x"4";  -- 4
```

NOTE: If you use nVidia toolkit v2.3 to compile the CUDA code using "nvcc –cubin", you will see a text version of the binary file with all the instructions and definitions for cmem, regs and other information.

4. Save the file as "vector_add_configuration.vhd".
5. In the ISE project, inside the "Hierarchy" panel on the left, left-click "uTestBenchConfig" and select "Add Source".
6. Enter the path to "vector_add_configuration.vhd" file in the "File Name" space and click "Open". Now you can see that this file has been added to your project hierarchy.
7. Now open Double-click "uTestBenchConfig" to open it in the source pane.
   a. Scroll down until you see something like this,

```
architecture Behavioral of tb_configuration is

    component autocor_configuration
        port (
            clk                     : in  std_logic;
            reset                   : in  std_logic;
            reset_registers         : in  std_logic;
```

b. We need to add “vector_add_configuration” component to the architecture of this source file. You can add it as highlighted below.

```
architecture Behavioral of tb_configuration is

    component vector_add_configuration
        port (
            clk                     : in  std_logic;
            reset                   : in  std_logic;
            reset_registers         : in  std_logic;

            cmem_reg_cs             : in  std_logic;
            cmem_reg_rw             : in  std_logic;
            cmem_reg_adr            : in  std_logic_vector(5 downto 0);
            cmem_reg_data_in        : in  std_logic_vector(7 downto 0);
            cmem_reg_data_out       : out std_logic_vector(7 downto 0);

            kernel_reg_cs           : in  std_logic;
            kernel_reg_rw           : in  std_logic;
            kernel_reg_adr          : in  std_logic_vector(3 downto 0);
            kernel_reg_data_in      : in  std_logic_vector(31 downto 0);
            kernel_reg_data_out     : out std_logic_vector(31 downto 0);

            cmem_param_size_out     : out std_logic_vector(5 downto 0);
            kernel_param_size_out   : out std_logic_vector(3 downto 0);
            kernel_block_x_out      : out std_logic_vector(15 downto 0);
            kernel_block_y_out      : out std_logic_vector(15 downto 0);
            kernel_block_z_out      : out std_logic_vector(15 downto 0);
            kernel_grid_x_out       : out std_logic_vector(15 downto 0);
            kernel_grid_y_out       : out std_logic_vector(15 downto 0);
            kernel_gprs_out         : out std_logic_vector(8 downto 0);
            kernel_shmem_size_out   : out std_logic_vector(31 downto 0);
            threads_per_block_out   : out std_logic_vector(11 downto 0);
            warps_per_block_out     : out std_logic_vector(5 downto 0);
            blocks_registers_out    : out std_logic_vector(7 downto 0);
            blocks_shared_mem_out   : out std_logic_vector(7 downto 0);
            blocks_warps_out        : out std_logic_vector(7 downto 0);
            blocks_per_core_out     : out std_logic_vector(3 downto 0)
        );
    end component;

    component autocor_configuration
        port (
            clk                     : in  std_logic;
            reset                   : in  std_logic;
            reset_registers         : in  std_logic;
```

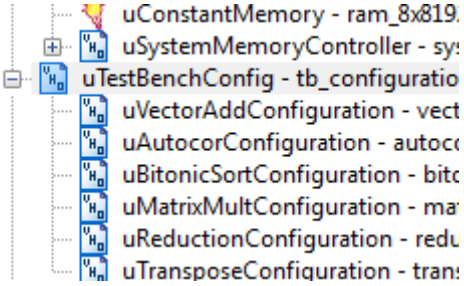
c. Now again scroll down until you see something like this

```
begin

    gAUTOCOR_CONFIGURATION: if APPLICATION = "AUTOCORR" generate
        uAutocorConfiguration : autocor_configuration
            port map (
                clk                     => clk,
                reset                   => reset,
                reset_registers         => reset_registers,
```

d. We need to specify the port map for the newly added component. Modify as highlighted below.

```
begin

    gVECTOR_ADD_CONFIGURATION: if APPLICATION = "VECTOR_ADD" generate
        uVectorAddConfiguration : vector_add_configuration
            port map (
                clk                     => clk,
                reset                   => reset,
                reset_registers         => reset_registers,

                cmem_reg_cs             => cmem_reg_cs,
                cmem_reg_rw             => cmem_reg_rw,
                cmem_reg_adr            => cmem_reg_adr,
                cmem_reg_data_in        => cmem_reg_data_in,
                cmem_reg_data_out       => cmem_reg_data_out,

                kernel_reg_cs           => kernel_reg_cs,
                kernel_reg_rw           => kernel_reg_rw,
                kernel_reg_adr          => kernel_reg_adr,
                kernel_reg_data_in      => kernel_reg_data_in,
                kernel_reg_data_out     => kernel_reg_data_out,

                cmem_param_size_out     => cmem_param_size_out,
                kernel_param_size_out   => kernel_param_size_out,
                kernel_block_x_out      => kernel_block_x_out,
                kernel_block_y_out      => kernel_block_y_out,
                kernel_block_z_out      => kernel_block_z_out,
                kernel_grid_x_out       => kernel_grid_x_out,
                kernel_grid_y_out       => kernel_grid_y_out,
                kernel_gprs_out         => kernel_gprs_out,
                kernel_shmem_size_out   => kernel_shmem_size_out,
                threads_per_block_out   => threads_per_block_out,
                warps_per_block_out     => warps_per_block_out,
                blocks_registers_out    => blocks_registers_out,
                blocks_shared_mem_out   => blocks_shared_mem_out,
                blocks_warps_out        => blocks_warps_out,
                blocks_per_core_out     => blocks_per_core_out
            );
    end generate;

    gAUTOCOR_CONFIGURATION: if APPLICATION = "AUTOCORR" generate
        uAutocorConfiguration : autocor_configuration
            port map (
                clk                     => clk,
                reset                   => reset,
                reset_registers         => reset_registers,
```

e. Save the file. You can see that “uVectorAddConfiguration” gets added below the “uTestBenchConfig” hierarchy in the “Hierarchy” panel.

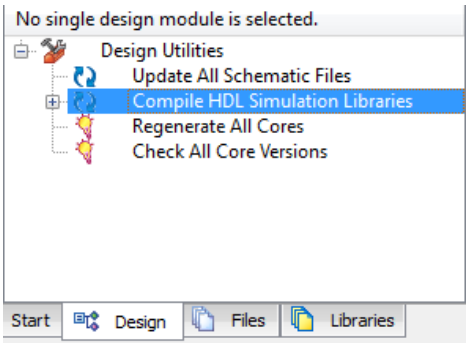


4.4 Running Simulation

1. For our example, we will use the Vector Add application with 16 cores. Follow the below steps,

    a. Change the number of Cores to 16 and Warp Lanes to 2 by following the steps in Section 3.1.
    b. In the "tb_top_level" file, change the APPLICATION variable in "uTestBenchConfig" to "VECTOR_ADD".
    c. In the "tb_top_level" file, change the APPLICATION variable in "uWriteInstructions" to "VECTOR_ADD".
    d. Use the “init_global.coe” file from FlexGrip\NewApplication folder. This contains sample data of size 16 for Vector Add application.
    e. Configure the configuration parameters by following steps in Section 4.3 for a 16 data size Vector Add application

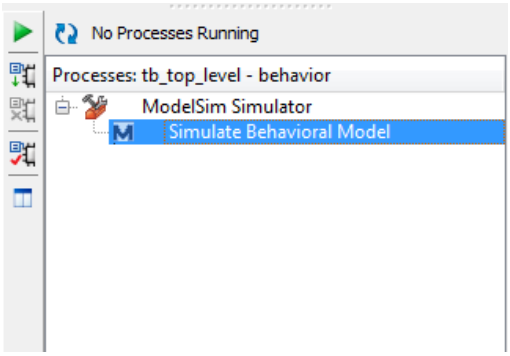
2. In ISE, click the "Simulation" radio button in the design pane.
   NOTE: If you have already compiled your HDL Simulation Libraries in Section 3.5, you can skip steps 3,4,5 & 6.
3. In the "Hierarchy" pane, left-click once on “xc6vlx240t-1ff1156”. It will become highlighted in blue.
4. Now click on “+” button next to “Design Utilities” so that the tree expands.



5. RIGHT-CLICK on "Compile HDL Simulation Libraries" and select "Rerun All”.
6. Now ISE compiles the Simulation Libraries and when you are complete, you will get the following message,

```
Process "Compile HDL Simulation Libraries" completed successfully
```

7. In the "Hierarchy" pane, left-click once on "tb_top_level". It will become highlighted in blue.
8. In the "Processes: tb_top_level - behavior", click the "+" button next to "ModelSim Simulator" so that the tree expands.



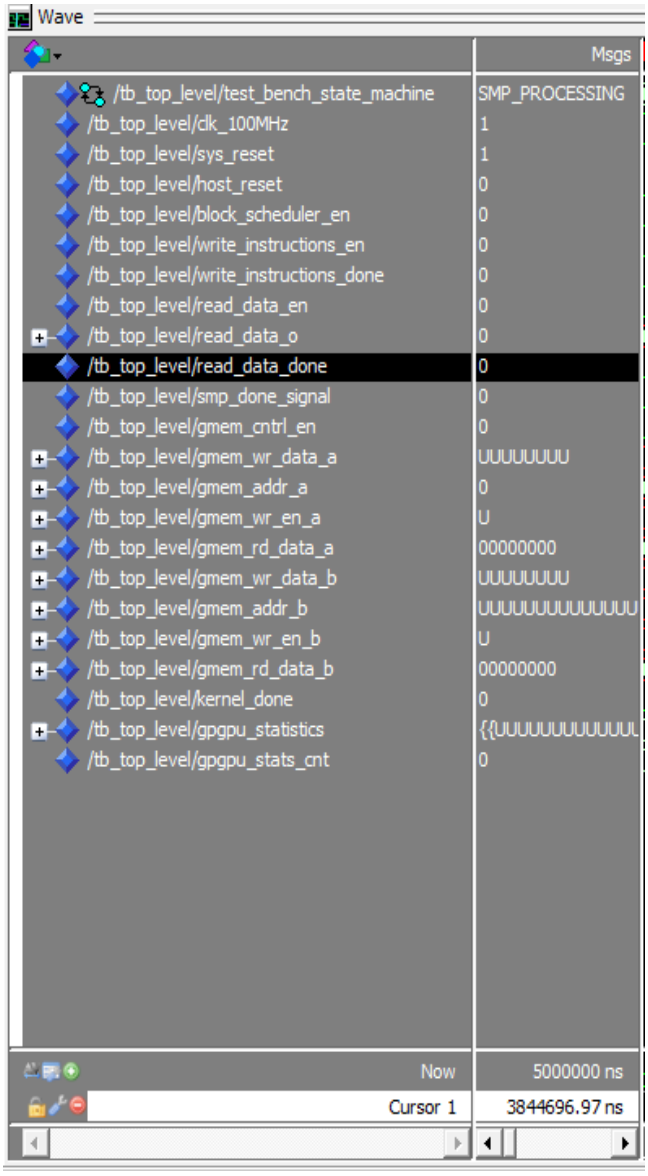
9. RIGHT-CLICK on "Simulate Behavioral Model" and select "Rerun All"
10. ModelSim will start and will show the following error:

```
# vsim -do \{do \{tb_top_level.fdo\}\}
# ** Note: (vsim-3812) Design is being optimized...
# ** Error: (vopt-19) Failed to access library 'work.{tb_top_level' at "work.{tb_top_level".
# No such file or directory. (errno = ENOENT)
# ** Error: Library work.{tb_top_level not found.
# Error loading design
```

11. In the ModelSim command line, type the following:

    ModelSim> do tb_top_level.fdo
    Hit <Enter>

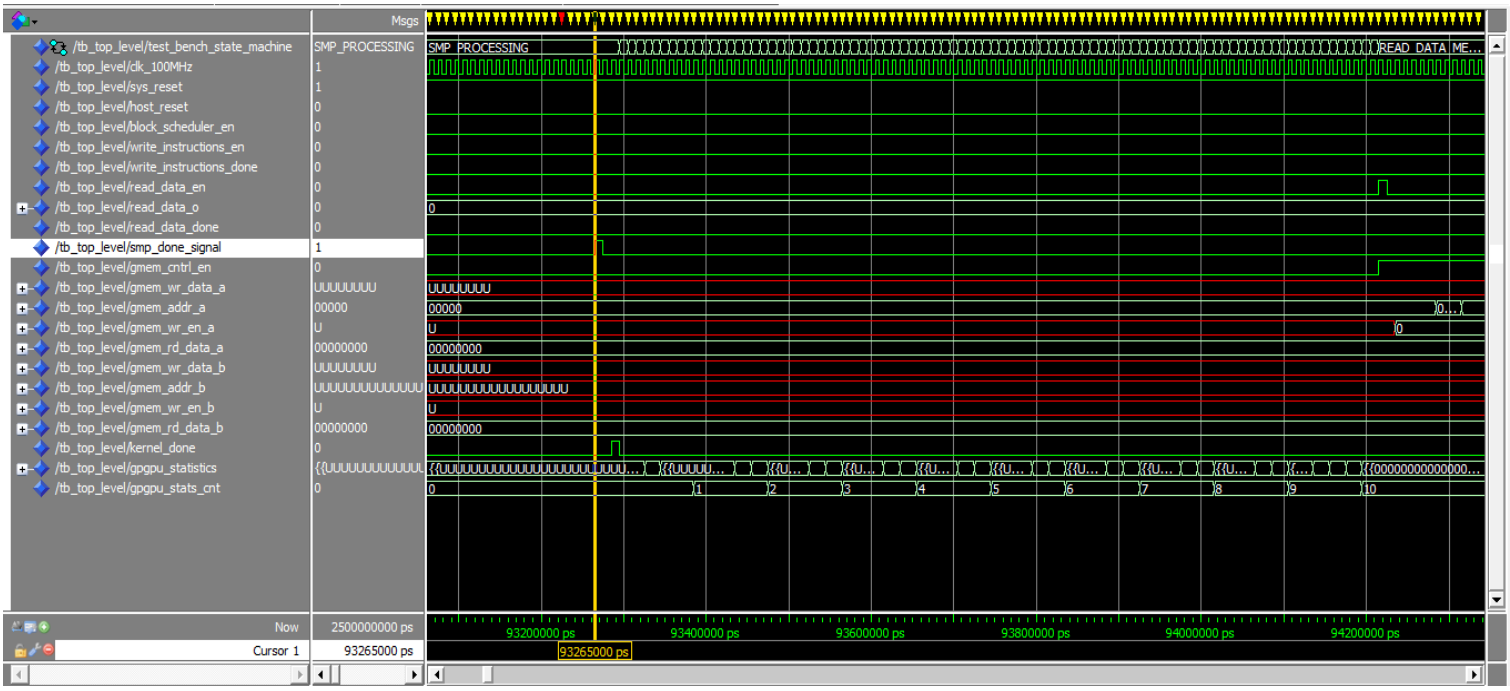
12. At this point, ModelSim will compile and optimize the design (NOTE: If an error occurs due to it being not able to move a file, close ModelSim and rerun the simulation starting from Step (9) above.
13. Once complete, ModelSIm will show with the waveform.
14. First, we will set up some signals to watch. We will use the existing signals defined and just remove the ones we don't want. You may add or keep as many as you like. Delete the signals from the "Wave" pane such that only the signals shown below are left:



15. Next, we will have to define the length of time. For an 8 core Matrix Multiply, it takes approximately 2 ms. To be safe, enter "2.5 ms" into the "Run Length" text box located in the top toolbar of ModelSim.
16. In the toolbar to the left of the "Run Length", click the "Restart" button.
17. Next, In the toolbar to the right of the "Run Length", click the "Run" button.
    NOTE: If the simulation runs for several hours, you can turn-off the warning message in ModelSim
    (In ModelSim click on *Simulate* → *Runtime Options* → *Message Severity*)
18. Once the simulation completes, inside the Waveform, RIGHT click and select "Zoom Full".

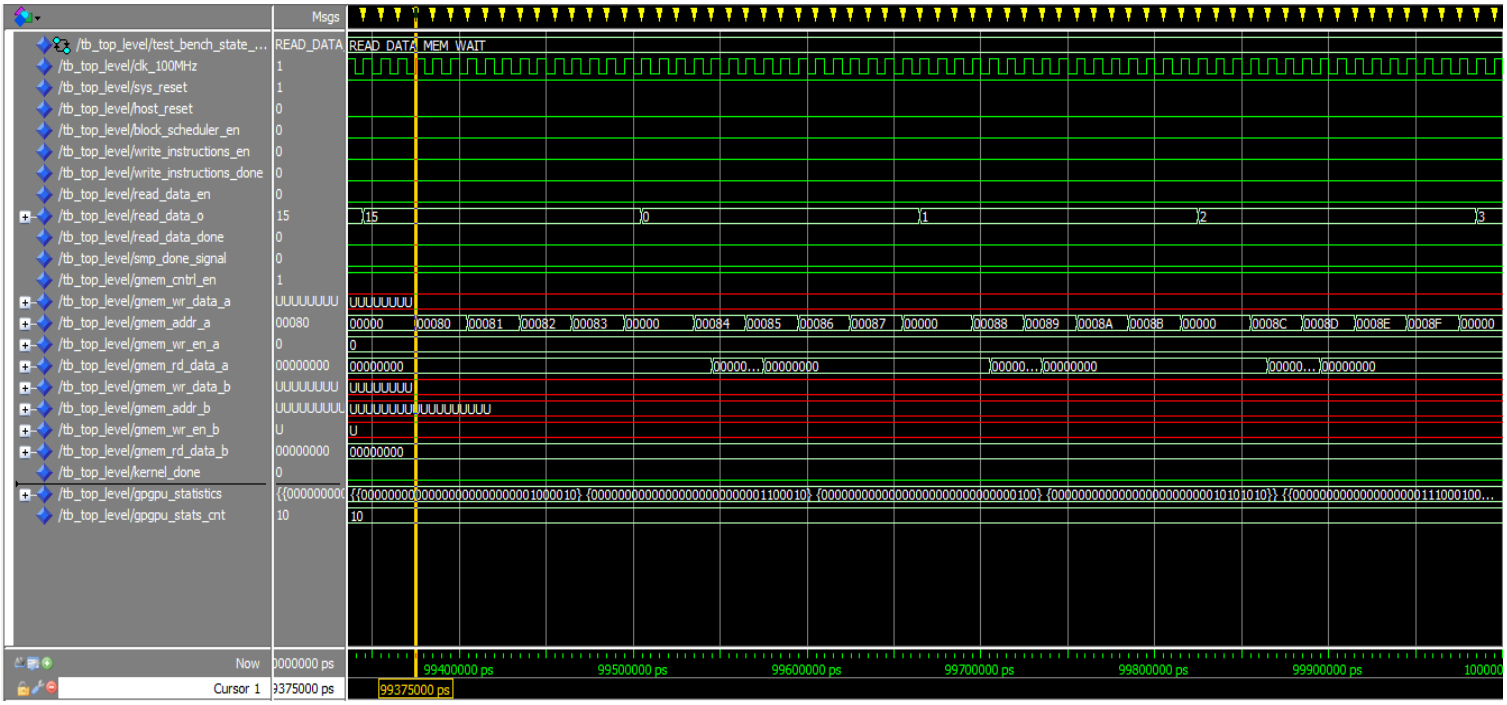
19. Using your cursor in the waveform, move it over the /tb_top_level/test_bench_state_machine and LEFT click where the transition occurs from "SMP_PROCESSING" to "READ_DATA_MEM_WAIT" as shown below:

20. Press SHIFT+C to zoom in on the cursor location until it looks like the following (NOTE: You may have to move your cursor to have it centered properly):



21. After the GPGPU has completed processing, the test bench will read all the data from the global memory. We only care about the results. From Section 3.4 above, we know that the results start at address 0x080. To check the results, we first need to convert the signal outputs in the waveform from binary to hex/decimal. Perform the following :
    a. RIGHT click on the signal /tb_top_level/read_data_o and select Radix->Decimal
    b. RIGHT click on the signal /tb_top_level/gmem_addr_a and select Radix-> Hexadecimal

22. Scroll the cursor to the right while watching the /tb_top_level/gmem_addr_a signal until the address is at 80 as shown below:



23. You will notice that the signal /tb_top_level/read_data_o has a value of 15, this is because the output data takes a couple of cycles before the data is read and displayed. However, if you continue progressing to the right, you will see the correct values for the first row in the Matrix Multiply application as shown in the /tb_top_level/read_data_o signal.

5. Reference

1. Kevin Andryc, Murtaza Merchant, and Russell Tessier, “FlexGrip: A Soft GPGPU for FPGAs
2. CUDA Toolkit Documentation http://docs.nvidia.com/cuda/