OpenCL FPGA Lab 1 – Matrix Matrix Multiplication

In the previous lab, we go over the process of performing Intel FPGA OpenCL emulation and simulation. This time, we will use the process introduced before explore some basic optimization techniques that can be applied to your OpenCL kernel.

Matrix Matrix Multiplication (MMM):

MMM is a typical application used for acceleration on GPUs and FPGAs. As there are no data dependency in the calculation, it can be easily paralyzed. Generally, it contains 3 nest loops, that performs calculation on each row and column. The inner most loop usually used to calculate the dot product of the row vectors from A and column vector from B. There are various algorithm level optimizations targeting MMM, for example loop interchange, blocking, etc. But that is beyond the scope of this lab. In this lab, we will start with the raw MMM code, and try some simple optimization on the OpenCL toolflow level.

```
#define SIZE 16
__kernel  void mmm(__global float* restrict a, __global float* restrict b, __global float* restrict c){
    for (int i = 0; i < SIZE; i++){
        for (int j =0; j < SIZE; j++){
            float temp = 0;
            for (int k =0; k < SIZE; k++){
            temp += (a[i*SIZE+k] * b[k*SIZE+j]);
            }
            c[i*SIZE+j] = temp;
        }
    }
}
```

OpenCL Kernel Type:

- NDRange Kernel:

This kernel is structured as general CUDA style code, which divide the entire job into multiple small tasks that shares the same functionality, but each one only working on a subset of the problem. Taking the MMM for example, we can divide the entire workload into several work-items, each work-items handles the calculation of each row of the result matrix, or even divided into element level. Each work-item is assigned a local id. During execution, the work-item will first fetch its id from dispatcher, then fetch the related memory element from the global memory and perform the execution, as shown in the code fragment below:

```
#define SIZE 16
__kernel  void mmm(__global float* restrict a, __global float* restrict b, __global float* restrict c){

    int i = get_local_id(0);
    int row_start_index = i*SIZE;

    for(int j = 0; j < SIZE; j++){
        float temp = 0;
        for (int k =0; k < SIZE; k++){
            temp += (a[i*SIZE+k] * b[k*SIZE+j]);
        }
        c[row_start_index+j] = temp;
    }
}
```

- Single Work-item Kernel:

Unlike the NDRange Kernel, this get rid of all the function calls of get_local_id(), get_global_id(). The compiler treats the task as a single work-item. This will give the compiler more freedom to analyze the kernel code and let the kernel working in a pipeline manner.

Task 1: Multiple Work-item Kernel

Enter the multi_workitem directory.

"cd multi_workitem"

0, Read the kernel code locate in "device/kernel.cl", understand how does each work-item works.

1, Perform emulation

"source run_emulation.sh"

2, Check the report:

0) "firefox reports/report.html"
1) Navigate to the "Summary" page, record the Resource usage.
2) Navigate to the "Loop analysis" page, record the pipeline status.
3) Navigate to the "System Viewer" page, record the memory access pattern.

3, Perform simulation

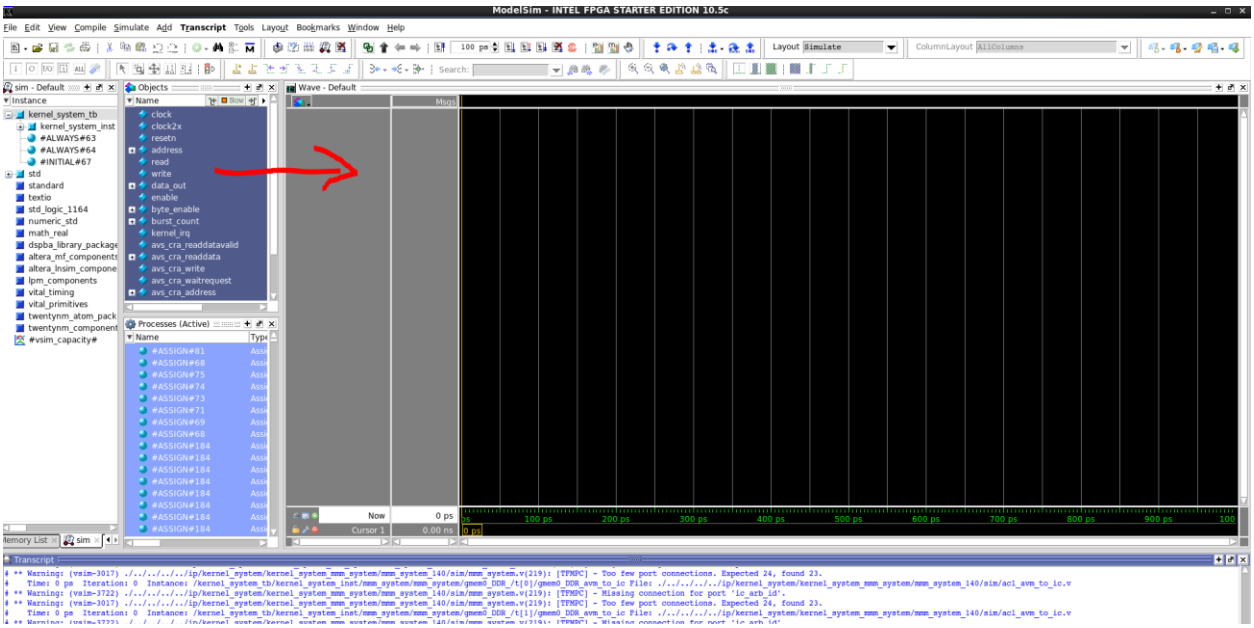
0)"source run_simulation.sh"

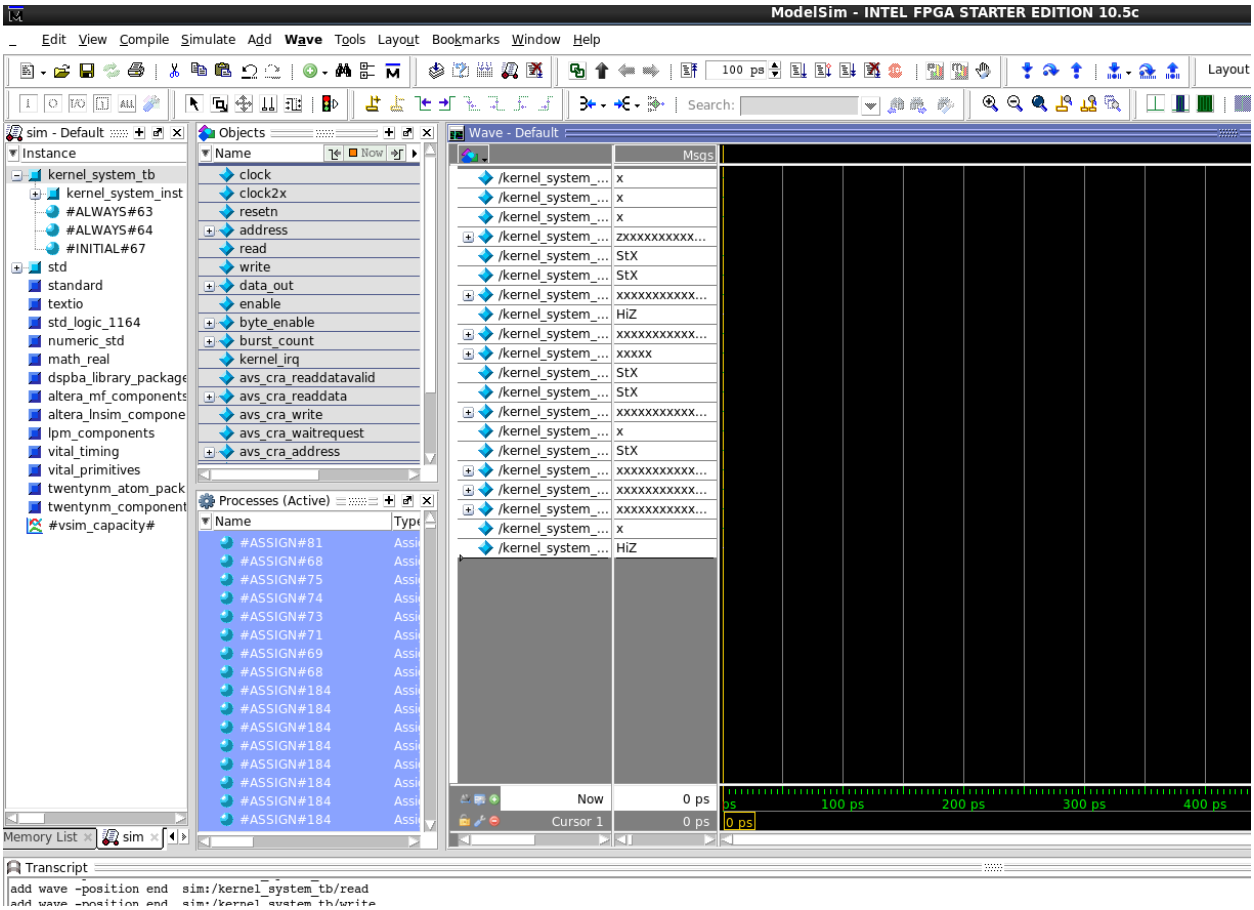
1) In the launched ModelSim GUI, type

"source msim_setup.tcl"

"ld"

2) Drag all the signals from left side to right





3) Set the simulation length as 100ns, run the simulation.

4) Record the # of cycles it takes before the kernel finished executions (kernel_irq signal turn to high).

Task 2: Single Work-item Kernel

Enter the multi_workitem directory.

"cd single_workitem"

0, Read the kernel code locate in "device/kernel.cl", comparing the difference with the multi_workitem kernel code.

1, Perform emulation

"source run_emulation.sh"

2, Check the report:

0) "firefox reports/report.html"
1) Navigate to the "Summary" page, record the Resource usage.
2) Navigate to the "Loop analysis" page, record the pipeline status.
3) Navigate to the "System Viewer" page, record the memory access pattern.

3, Perform simulation

0)"source run_simulation.sh"

1) In the launched ModelSim GUI, type

"source msim_setup.tcl"

"ld"

2) Drag all the signals from left side to right

3) Set the simulation length as 100ns, run the simulation.

4) Record the # of cycles it takes before the kernel finished executions (kernel_irq signal turn to high).

Task 3: Loop Unrolling

Enter the multi_workitem directory.

"cd loop_unroll"

0, Read the kernel code locate in "device/kernel.cl".

The loop unrolling is achieved by adding the following line in front of the for loop:

"#pragma unroll N"

N here represents how many times you want to unroll the loop. The more you unroll, the more resource it will consume. By doing loop unrolling, the compiler will duplicate the kernel hardware multiple times so multiple calculation can be performed in the same time.

1, Uncomment the code fragments in the kernel code following the comments.

1, Perform emulation

"source run_emulation.sh"

2, Check the report:

0) "firefox reports/report.html"
1) Navigate to the "Summary" page, record the Resource usage.
2) Navigate to the "Loop analysis" page, record the pipeline status.
3) Navigate to the "System Viewer" page, record the memory access pattern.

3, Perform simulation

0)"source run_simulation.sh"

1) In the launched ModelSim GUI, type

"source msim_setup.tcl"

"ld"

2) Drag all the signals from left side to right

3) Set the simulation length as 100ns, run the simulation.

4) Record the # of cycles it takes before the kernel finished executions (kernel_irq signal turn to high).

Task 4: Summary

Compare the resource consumption, kernel latency among the 3 implementations, and find the tradeoff between the performance and resource consumption.