# ECE786: PROGRAMMING ASSIGNMENT TWO

## TASK 1: Instruction Semantics

***CODE ANALYSIS FLOW DIAGRAM***

***CODE ANALYSIS DISCUSSION***

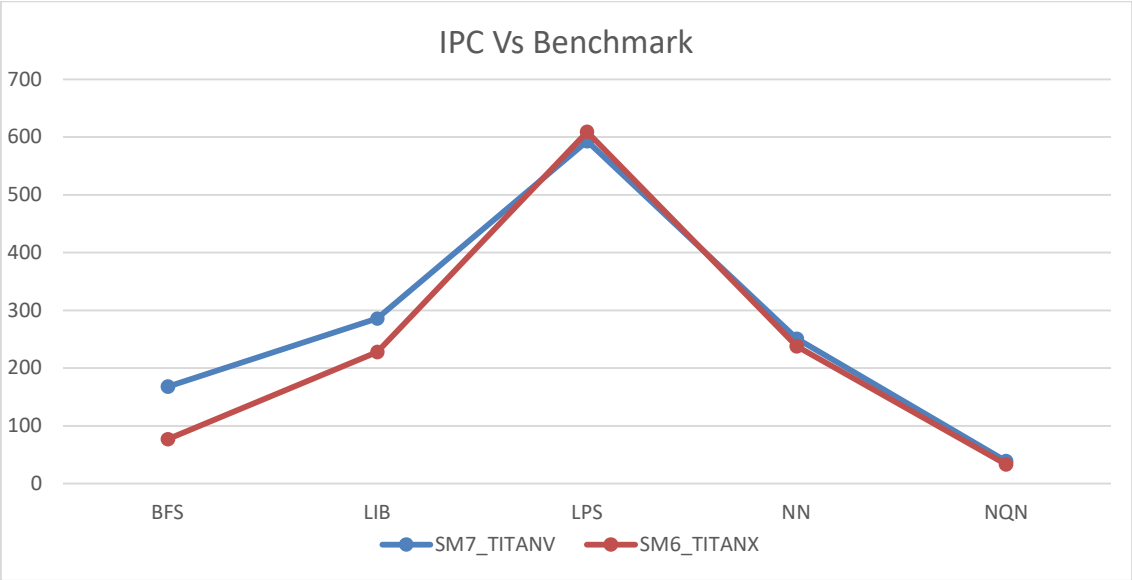
- The task required modifications to the functional simulator code of gpgpu-sim so as to modify the semantics of the floating point add instructions.
- Code inspection revealed instructions.cc as the source code file for various instruction semantics for the simulator, the opcodes for which were defined in opcodes.def. These opcodes were mapped to separate functions within the instructions.cc file.
- As described in the above diagram, changes were made to the add_impl function to implement the task. The function utilized switch case to implement add functionality for different data types. The required type for floating point instructions was modified to convert the add instruction to a power instruction as per the provided semantics.

***RESULTS***

```
gpgpu_simulation_time = 0 days, 0 hrs, 0 min, 1 sec (1 sec)
gpgpu_simulation_rate = 286 (inst/sec)
source vector A: 1.000 2.000 3.000 4.000 5.000 6.000 7.000 8.000 9.000 1.000
source vector B: 2.000 3.000 2.000 3.000 2.000 3.000 2.000 3.000 2.000 3.000
result vector C: 1.000 8.000 9.000 64.000 25.000 216.000 49.000 512.000 81.000 1.000
GPGPU-Sim: *** exit detected ***
```

## TASK 2: Benchmark performance study

### *IPC PLOTS*

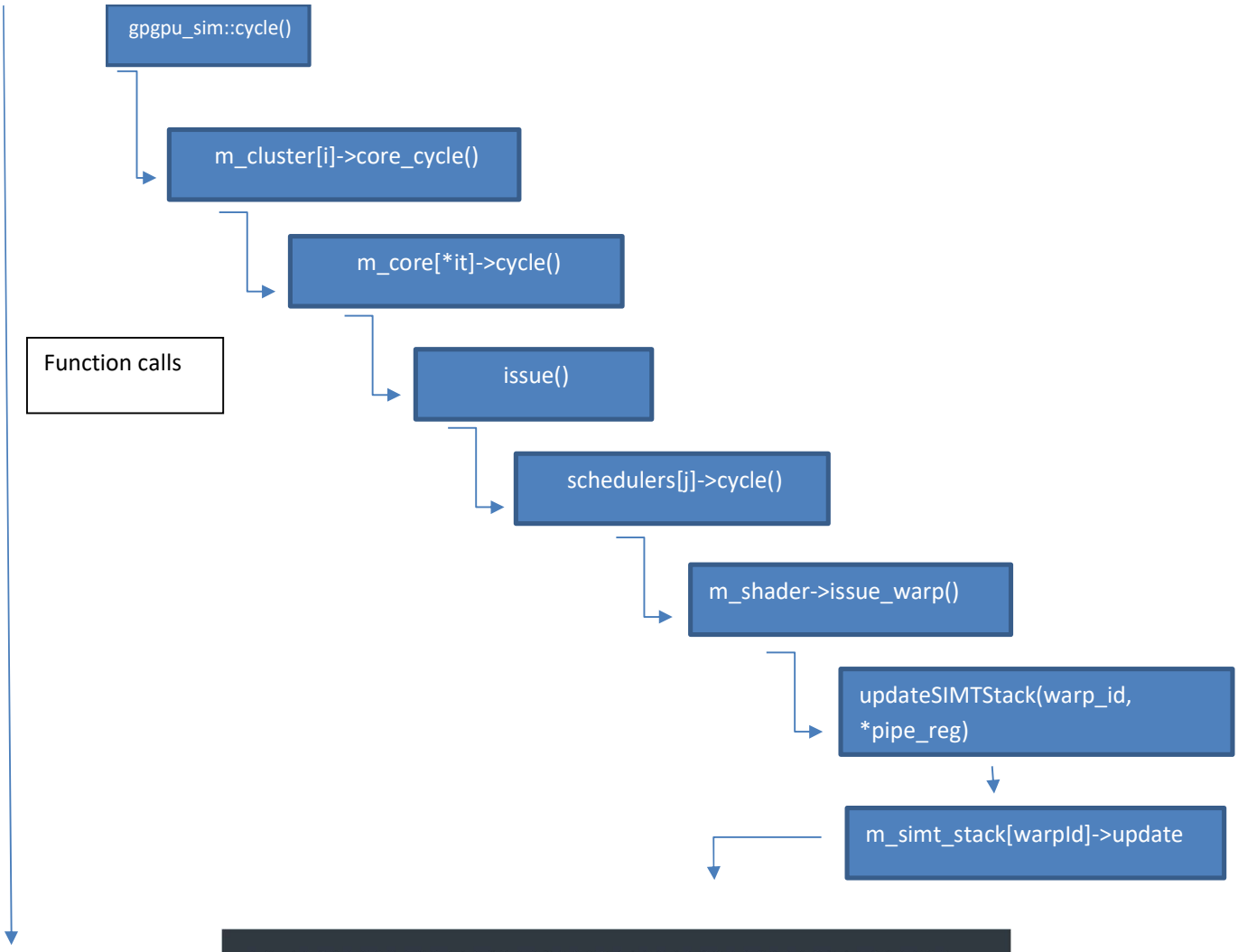



### *PLOT DISCUSSION*

- A python-based automation script was used to run all the benchmarks for the given two gpgpu-sim configurations and extract the IPC from the results.
- A generic observation that can be made for the benchmarks across gpu configurations is that the LPS benchmark extracts the maximum IPC from the simulator. This can elude to the fact that the LPS benchmark actually consists of SIMD style code that ideally matches the use case for which GPUs are made.
- As continuation of previous bullet, the NQN benchmark performs poorly for the gpu configurations. This leads to a conclusion that the benchmark code is less suited to run on a GPU. This also leads to the conclusion that the benchmark models code that has inter data dependencies which reduces the data parallelism that a SIMT style GPU can extract.
- Between the two GPU configurations, SM7_TITANV has more resources in terms of number of GPU clusters and the number of cores per cluster. So, its evidently observed from the graph that most benchmarks perform better from SM7_TITANV configuration. The performance difference is significant for BFS and LIB benchmarks.
- For, other benchmarks, the performance difference is not that significant, this implies that a boost in hardware resources doesn't improve the performance of these benchmarks. This might be the case because the benchmarks don't have enough data parallelism that can be extracted by the enhanced hardware.

## TASK 3: Branch instructions and divergence

***CODE ANALYSIS FLOW DIAGRAM***




```
// ***********************************************
// branch stats logic - Aditya
// ***********************************************
if(!branch_stat.branch)
{
  if(next_inst_op == BRANCH_OP)
  {
    branch_stat.branch = true;
    simt_stack::branch_count++;
  }
}

if(!branch_stat.divergent_branch)
{
  if((next_inst_op == BRANCH_OP) && warp_diverged)
  {
    branch_stat.divergent_branch = true;
    simt_stack::divergent_branch_count++;
  }
}
```

***CODE ANALYSIS DISCUSSION***

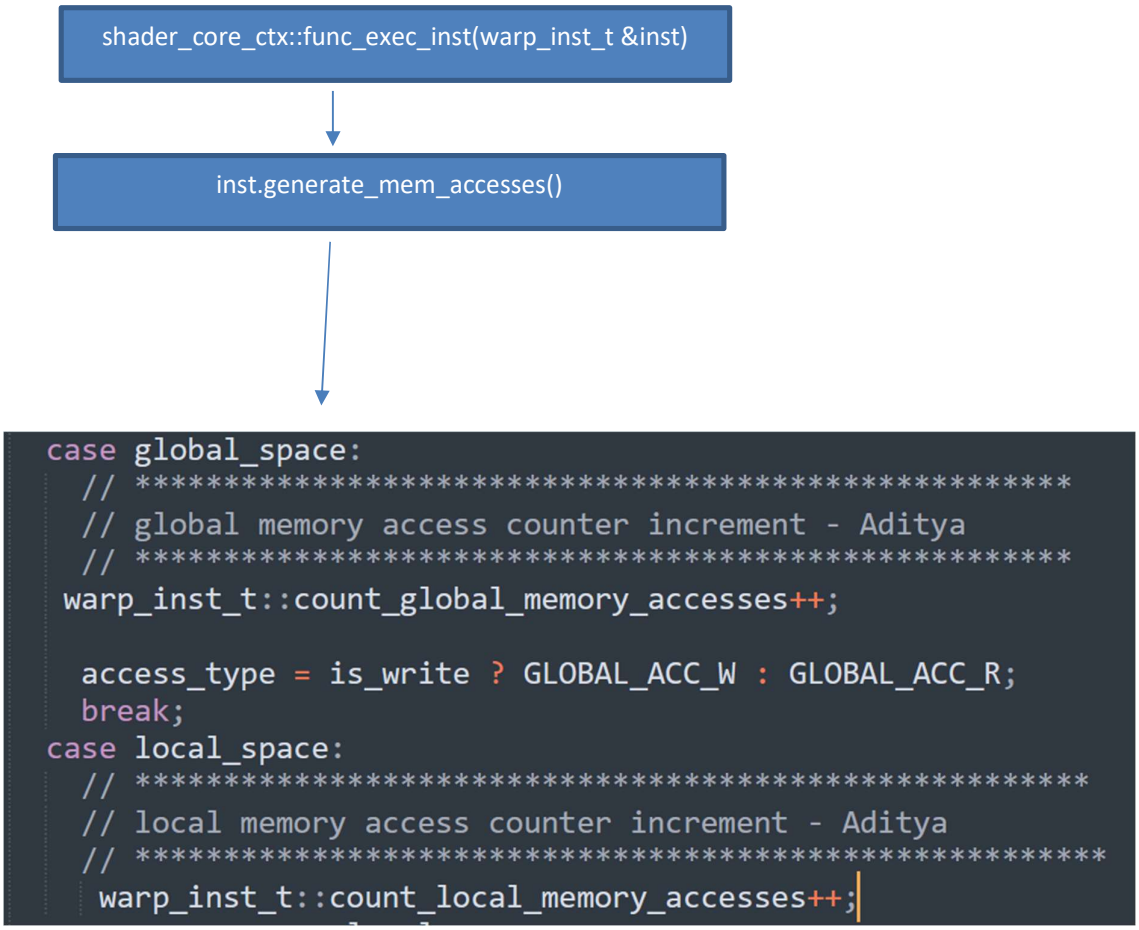
- The task enforced counting of total conditional branches and conditional divergent branches. Based upon the gpu architecture, this required modification to the simulator code related to SIMT stack.
- As described in the above flow diagram, investigating the function calls from the main gpgpu-sim thread down to the SIMT stack, revealed the above functional flow. The update() method within the simt_stack class is responsible for control flow divergence handling.
- As the task required keeping a count of all the branches within a given benchmark, static variables were declared within the simt_stack class to keep a count of the same.
- As the simt_stack is a per warp structure, the static counters kept a global count of all the branches and divergence for all the warps executed by the simulator.
- Looking into the microarchitecture opcodes revealed three opcodes for branches, call and return. The branch opcode was used to distinguish conditional branches from call and return.
- Additional check was required so as to count a warp only once for any number of branches it executes. This was obtained by utilizing a structure that implemented a stick bit to keep track of branch execution within a warp.

***RESULT***

```
############ branch statistics ################
# of warps excuted conditional branch instructions and have divergence: 248
# of warps excuted conditional branch instructions(no matter  they have divergence or not): 400
```

## TASK 4: Memory access space

***CODE ANALYSIS FLOW DIAGRAM***




***CODE ANALYSIS DISCUSSION***

- The task requires count of global and local memory accesses by the benchmark code.
- The analysis from task 3 reveals the fact that the issue_warp method calls func_exec_inst function that handles all memory accesses. These accesses are classified based on access_type and handled in switch case statements as described in the above diagram.
- In order to count all the memory accesses, couple of static counters were declared in the warp_inst_t class and incremented in the given switch case.

***RESULT***

```
################ memory access statistics ##############
# of global memory accesses : 109200
# of local memory accesses : 0
/exit
```