Program Assignment 2: GPGPU-Sim

Brian Park

North Carolina State University, Computer Engineering 786

March 2023

1 Task 1: Instruction Semantic

For this task, I chased down where possible the add instruction would be implemented `instructions.cc`. There were two possible options: `add_impl` and `addp_impl`. It seems like the p version was predicate version, so I tried modifying `add_impl` and it worked. It was pretty simple to modify based on the information I knew, so it's only a one line change. I only modified the FP32 instructions to do power.

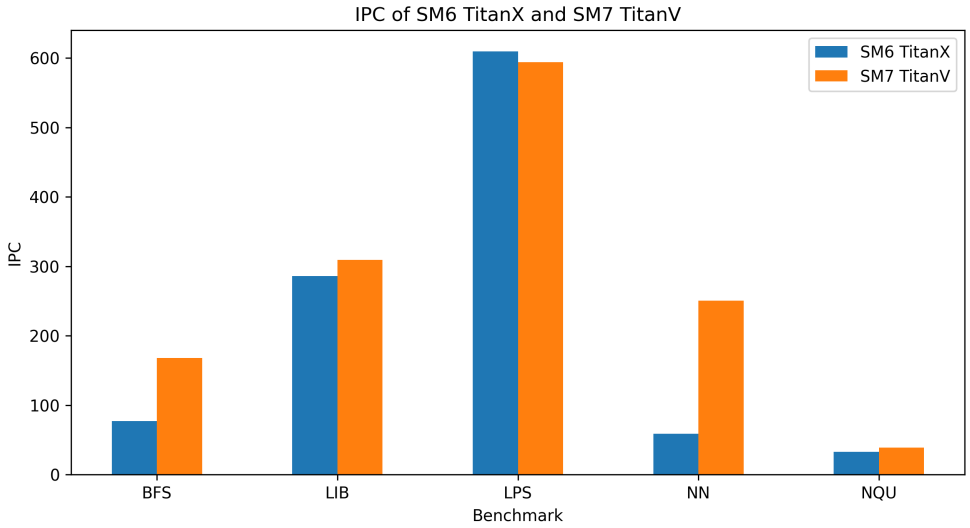
On line 1061:

```
void add_impl(const ptx_instruction *pI, ptx_thread_info *thread) {
...
    // data.f32 = src1_data.f32 + src2_data.f32;
    data.f32 = pow(src1_data.f32, src2_data.f32);
...
}
```

```
diff --git a/src/cuda-sim/instructions.cc b/src/cuda-sim/instructions.cc
index 8936fa8..dfa599b 100644
--- a/src/cuda-sim/instructions.cc
+++ b/src/cuda-sim/instructions.cc
@@ -967,7 +967,7 @@ void addp_impl(const ptx_instruction *pI, ptx_thread_info *thread) {
       data.f16 = src1_data.f16 + src2_data.f16;
       break;  // assert(0); break;
     case F32_TYPE:
-      data.f32 = src1_data.f32 + src2_data.f32;
+      data.f32 = src1_data.f32 + src2_data.f32 + src2_data.f32;
       break;
     case F64_TYPE:
     case FF64_TYPE:
@@ -1058,7 +1058,7 @@ void add_impl(const ptx_instruction *pI, ptx_thread_info *thread) {
       data.f16 = src1_data.f16 + src2_data.f16;
       break;  // assert(0); break;
     case F32_TYPE:
-      data.f32 = src1_data.f32 + src2_data.f32;
+      data.f32 = pow(src1_data.f32, src2_data.f32);
       break;
     case F64_TYPE:
     case FF64_TYPE:
```

2 Task 2: Benchmark Performance Study

Note that the argument `-gpgpu_max_insn 100000000` was added to `gpgpusim.config` to limit the number of instructions.



Here, we measure IPC. This indicates a higher number is better. The IPCs are abnormally high. Trusting the output of GPGPU-Sim, the only conclusion for this high IPC must be that the instructions must be decoupled per SM to give such a high IPC (instructions are aggregated across multiple SMs and divided by total number of cycles for one SM).

Assuming that SM7 TitanV is a succeeding generation of SM6 TitanX microacrchitecture (based on the naming), we see that it outperforms SM6 TitanX on BFS, LIB, NN, and NQU. Only LPS is where SM6 is better, but only by a little bit.

3 Task 3: Branch Instructions and Divergence

The total number of instructions that were conditional branches turned out to be 102600 instructions, while the number of those which diverged is 58616. That means 57.13% of the branches incurs divergence, showing that maybe this code is not optimal for GPU or needs to be carefully performance tuned in software for optimal performance.

For sanity check, I ran this on `vectorAdd.cu`, and I get only 5 conditional branches with no divergence. That sounds about correct, as the conditional branch will probably come from checking whether `tid` is less than number of elements. If that branch is taken, then there is no divergence as there is no reconvergence point and the threads remain inactive.

The code changed in simulator is pretty straightforward, by modifying `scheduler_unit::cycle()`, I can insert conditional statements that checks the active mask whether it is set to `0xffffffff` or not. Because we know warp size is 32, the active mask `0xffffffff` indicates that all the threads are active and there is no divergence. Thus, if the condition is not met, then there is a diverence. The code is shown below:

```
void scheduler_unit::cycle() {
...
  const active_mask_t &active_mask =
        m_shader->get_active_mask(warp_id, pI);
  m_stats->conditional_branch_instr++;
  if (active_mask != 0xffffffff) {
    m_stats->conditional_branch_instr_diverged++;
    ...
  }
...
}
```

4 Task 4: Memory Access Space

There are 448400 global memory accesses and 448400 local memory accesses. This aligns with what is already outputted in `gpgpu_n_mem_read_global + gpgpu_n_mem_write_global` and `gpgpu_n_mem_read_local + gpgpu_n_mem_write_loc`

Global and local memory accesses being the same shows that there is no cache reuse of the L1 or local memory.

The code modified is in `ldst_unit::memory_cycle`

```
bool ldst_unit::memory_cycle(warp_inst_t &inst,
                             mem_stage_stall_type &stall_reason,
                             mem_stage_access_type &access_type) {

  ...
    } else if (inst.space.is_global()) {  // global memory access
    // skip L1 cache if the option is enabled
    m_stats->global_memory_accesses++;
    if (m_core->get_config()->gmem_skip_L1D && (CACHE_L1 != inst.cache_op))
      bypassL1D = true;
  }

  if (bypassL1D) {
    m_stats->local_memory_accesses++;
    ...
  }
}
```