Mondrian Simulator

Representation

1. Since warp is the only unit in GPU which is guaranteed to run together, we have modelled one SIMD unit of Mondrian with one warp.

2. To remove the interleaving of warps due to multithreading, we only allow one warp to be run on one SM, so eventually the terms warp, SM and vault in Mondrian become synonymous for our case.

Realizing the representation

1. We use the config files for GTX480 as base.

2. -gpgpu_shader_core_pipeline <# thread/shader core>:<warp size> parameter is set to 8:8. This allows us to have one warp per shader (SM).

3. -gpgpu_shader_cta is set to 1, which defines maximum number of shader CTAs (concurrent thread arrays or thread_blocks) in one shader. This is the user provided limit. There exists another hard limit in the code, which can also be controlled if need be.

Changes to the architecture

1. GPU Fermi architecture has two warp schedulers per SM which issue two half warps in one cycle, so although we take two cycles to issue one single wrap completely, but end up issuing two of them, which turns out to be 2 instructions per two cycles.
   We also do away with this, and constrain the SM to have single warp scheduler by setting -gpgpu_num_sched_per_core to 1.

2. We rip off the l1 cache and l2 cache by setting gpgpu_cache:dl1 to none as well as gpgpu_cache:dl2 to none.

3. We set the -gpgpu_num_sp_units to 8 to allow 8 physical shader pipeline units.

4. We add our own prefetcher because there exists no already implemented prefetcher in the gpgpusim except for the one used in texture memory, but the prefetcher in the texture memory has logic coupled with the use of caches, so it's not straightforward to use their prefetcher as it is.

CUDA interface

1. Kernel functions should be called with config parameters as <#vaults, 8>. Specifying 8 as the number of threads in a thread block is necessary to allow formation of only one warp which is then guaranteed to occupy one whole SM.

2. Every kernel function would require passing of specific arguments IN ORDER, to allow setting up the prefetch units.
   (num_read_streams, addr_str1, addr_str2, …… addr_strN, size_rd_streams, num_write_streams, addr_str1, addr_str2, …. addr_strN, size_write_streams)

3. Everything should work as in normal CUDA, except you can't use shared memory now, because load and stores to shared memory are now being interpreted as reads and writes to the stream buffers, with the address being interpreted as the stream_number you want to read/write from/to.

4. To issue the read to stream buffer x, we need to do a load to shared memory to address x (which will be interpreted as the stream number)

   Example:

   ```
   unsigned id = threadIdx.x; // %r1 = threadIdx.x
   unsigned a;
   asm ("ld.shared.u32 %0, [%%r1];" : "=r"(a));
   ```

   This loads the head of the stream numbered id to variable a.

   The code read_example.cu can be referred to test the read_interface which simply reads two values from the head of the stream and then prints them out in the host main function.
   The benchmark code for read is written in read_test_inline_ptx.cu.

5. To issue the write to stream buffer x, we need to do a store to shared memory to address x(which will be interpreted as the stream number), and the value passed in to be stored in the stream.

Example:

```
unsigned id = threadIdx.x; // %r1 = threadIdx.x
asm volatile ("st.shared.u32 [%%r1], 4;");
```

This stores the value 4 in the stream buffer numbered id.

The code write_test_inline_ptx.cu provides the benchmark code for writes.

Note(s):

1. We do the hardcoding of the register to get the id, otherwise it would take one more mov instruction to move the value of id to the register and that hampers the rate at which we can issue the instructions.
2. The id should be assigned as the first line in the kernel so that it always gets assigned to the register r1.
3. The user should access the shared memories only by doing loads and stores with inline ptx and not by any other means.

Prefetcher Design

1. The timing model for read and write prefetcher (prefetch_unit_read_timing and prefetch_unit_write_timing) inherit from the base class prefetch_unit_timing.

2. The timing model for both the write and read prefetch units are present as class members in the ldst_unit.

3. The prefetch units are bootstrapped on every kernel function call in shader_core_ctx::issue_block2core here. This reads the arguments supplied to the kernel function in CUDA and uses them to setup the prefetch units. The helper functions to bridge this gap between CUDA program and simulation context are written in class function_info in ptx_ir.h here.

4. The functional model for both the read and write prefetch units is implemented in single class prefetch_unit_functional, since there is no difference in the two use cases. We instantiate two different objects for read and write units however.

5. max_in_flight_requests parameter in the timing model is set up in the constructor using the max_in_flight_requests_percentage of the total stream_buffer size passed as argument controlling how many pending requests are allowed to be in

flight used to decide when to issue a prefetch request.
Currently max_in_flight_requests_percentage is 50 and can be changed here.

6. New class members namely stream_number and the flag is_prefetch have been added to the class mem_access_t [code] and mem_fetch [code].

Prefetcher Functional Model

1. Keeps track of the head of each stream in the vector prefetch_address.

2. Uses data_size in bytes to increment the prefetch_address, which can be configured here (currently set to 4 for both read and write units).

Read Prefetcher Timing Model

1. bool read(stream_number, addr) is used to read from the stream, and returns true on success, otherwise false when the address has not yet been prefetched to the buffer.

2. void receive_response(stream_number, addr) adds a response to the prefetch_unit after it has arrived at the ldst_unit. code

3. bool can_issue(stream_number) checks whether a stream can issue a prefetch request based on the current size of the buffer and the number of requests in flight.

4. The prefetch reads are issued every cycle in the ldst_unit::memory_cycle by following a round robin policy to select the stream which issues a read. code

Write Prefetcher Timing Model

1. bool accept_write_request(stream_number, warp_id) returns true if the write prefetch buffer is not full, and increments the store_requests counter for the corresponding warp.

2. The requests for write are only sent in batches of batch_size (can be configured as argument to constructor here, currently 4).

3. In every cycle of ldst_unit::memory_cycle, a round robin policy is used to select the stream which can possibly issue a write, which further checks if the write_buffer has requests greater than equal to the batch_size available to be sent, which then puts one batch in the ejection buffer. If the ejection buffer is not empty, one request from the ejection buffer is sent to the interconnect.
Note that ejection buffer is not refilled until the older batch has been served completely. code

Control Flow

1. In the outermost layer gpgpu_sim::cycle is called, which cycles the DRAM, CORE as well as ICNT.

2. Inside the shader_core_ctx::cycle, the issue stage calls the scheduler_unit's issue, which calls the issue_warp, which calls func_exec_inst, which is responsible for functionally simulating the instruction as well as setting up certain parameters to be used during timing simulation.

3. Inside func_exec_inst, execute_warp_inst is called which calls ptx_exec_inst further, which executes the ptx_code. Inside this function, it is detected if the instruction involves prefetcher access and a flag is hence set. It then calls the appropriate (ld or st) implementation for prefetch access in the instructions.cc, and upon their return sets the stream number as well as prefetch address in the warp_inst_t's m_per_scalar_thread's respective thread index.

4. Inside the instructions.cc, the functions stream_prefetch_read_impl and stream_prefetch_write_impl implement the read and write to prefetch buffers. The stream_number is read from the address [read_code & write_code]. This currently assumes we are writing inline ptx to read and write from the stream_buffer, but if we do not, we need to divide this address by data_size in bytes to decode the stream_number correctly.

5. Inside stream_prefetch_read_impl (and stream_prefetch_write_impl), the flag is_prefetch is set in warp_inst_t object which is important for two reasons :
    a. To convey the information to the timing model

    b. To convey the information to the next call to ptx_exec_inst since by then we would already set the space to global, and our earlier check won't otherwise work.

6. Upon return from execute_warp_inst_t inside func_exec_inst, a call to generate_mem_accesses is made which utilizes the is_prefetch flag set in the functional simulation, and if set, calls generate_prefetch_accesses [code]. Note that the name of the function may be misleading to the fact that we set is_prefetch = false in the mem_fetch object [code], but it is due to the fact that we classify those mem_fetch requests which are issued by the prefetch units to be is_prefetch, as opposed to the ones which are requested by the user. This function solely does the job of distinguishing the process of generating memory accesses for prefetch units from the normal global accesses.

7. The next function to focus on is the ldst_unit::cycle which is called from the execute stage [here]. In this function, the receipt of read requests back from the memory is pushed to the read prefetch unit here, and the receipt of ack for write request is handled here.

8. ldst_unit::cycle calls ldst_unit::memory_cycle, where the reads and writes to the prefetch buffer are made, stalling the pipeline if the requests can't be served updating the relevant counters for stream buffer stalls (see here and here for new stall types added), as well as the new requests from the prefetch units are sent every cycle. [code]

Simulating DRAM

1. According to Bruce Jacob's book : Memory Systems - cache, DRAM, disk (2007), Chapter 11

   "Specialized or high-performance DRAM memory systems such as Direct RDRAM, GDDRx, and FCRAM have slightly varying sets of DRAM commands and different command timings and interactions",

   Hence we do not use GDDR memory of gpgpu-sim to model dram.

2. We integrate Ramulator (source) with gpgpusim taking inspiration from here, but making a few changes to the source code and the Makefiles because their

integration code does not compile as it is. The details can be seen in this commit.

3. The following parameters have been fine tuned to get the desired read and write bandwidth and may require playing around with them more in future:
    a. In gpgpupsim.config :
        i. -gpgpu_frfcfs_dram_sched_queue_size
        ii. -gpgpu_dram_return_queue_size
        iii. -gpgpu_dram_partition_queues
        iv. -gpgpu_clock_domains
        v. -gpu_dram_timing_opt (didn't change till now, but may be useful to change)
    b. In config_fermi_islip.icnt :
        i. boundary_buffer_size
        ii. ejection_buffer_size
        iii. input_buffer_size
        iv. vc_buf_size
    c. In file src/gpgpu-sim/dram.cc, the parameter mrqq [code]

4. Ramulator can be used to simulate the following DRAM standards :
    a. DDR3 (2007), DDR4 (2012)
    b. LPDDR3 (2012), LPDDR4 (2014)
    c. GDDR5 (2009)
    d. WIO (2011), WIO2 (2014)
    e. HBM (2013)
    f. SALP
    g. TL-DRAM
    h. RowClone
    i. DSARP

    So, this will be useful in future if we want to test the architecture with other types of memory.

Dockerized Development Workflow

1. To ease the process of development, we have setup a docker image pushed in the repository, which is recommended to be used for developing the simulator.

2. If you haven't already built the image you will need to download the cuda runtime library file named cudatoolkit_4.0.17_linux_64_ubuntu10.10.run in the docker

directory and then build the image.

3. Once you have the image, run the image using the following command :

   docker run --privileged -it -v HOST_REPO_CLONE:MOUNTED_REPO_CLONE IMAGE_NAME

4. Once you are inside the image, do the following :
   a. cd /home/gpgpu-sim
   b. source ../.bashrc
   c. source setup_environment
   d. make sure the version of gcc and g++ is gcc-5.5 and g++-5.5 by typing gcc --version and g++ --version.

5. Now you can make changes to the code as well as build it using make.

6. All the cuda programs as well as configuration files are present in Mondrian_test directory. To build a cuda program, go inside this directory and do, source ../setup_environment (you need to do this only once every time you bring up the image). Then to compile, you need to use the script compile.sh provided, which is because cuda requires gcc and g++ versions <= 4.4 and gpgpu-sim + ramulator require gcc and g++ version >=5. So this image comes with both the version installed, and in compile.sh we switch the versions by modifying sym_links and revert them after compilation.

   Usage: ./compile.sh <cuda_program_name>

   Note, we use nvcc with -keep flag, which means the ptx generated will also be available with the same name as your program with .ptx extension in the current directory.

7. Since we launched the image with mount volume, we can simply exit the image without worrying about persisting the work done on the host machine. Note this is valid only for changes made inside the gpgpu-sim directory.

Additional Notes

1. The number of cores in a cluster have been set to 1 and number of clusters also to be 1 in gppgusim.config. If you wish to change these parameters, you should also reflect the appropriate change in config_fermi_islip.icnt file keeping in mind the following rule :

   “Value K in config_fermi_islip.icnt is the number of clusters + gpgpu_n_mem* gpgpu_n_sub_partition_per_mchannel. The "gpgpu_n_mem" is number of memory controllers given in gpgpusim.config file.
   If you change k, you must make appropriate changes to gpgpu_n_clusters parameter in the gpgpusim config file.”

2. The two benchmarks read_test_inline_ptx.cu and write_test_inline_ptx.cu are the latest benchmarks, and show how inline ptx can be used to extract the memory bandwidth. How to write inline ptx can be seen here, and to understand ptx this resource can be used.

3. The following two metrics have been added to report the stream buffer read and write stalls in the stats:
   a. gpgpu_stall_shd_mem[g_mem_ld][stream_buffer_read_stall]
   b. gpgpu_stall_shd_mem[g_mem_st][stream_buffer_write_stall]