CS 758
Fall 2019
Homework 1
Due: Friday, October 4th, 2019 11:59 PM

You are also encouraged to talk with classmates in person or on Piazza about any issues you may have encountered. I will also post solutions to this assignment on Canvas. You are allowed two "free" late days for the homework assignments across the semester. Beyond that, the standard late assignment policy applies: you may submit up to 48 hours late with a 15% penalty per 24 hours. Additional extensions may be possible in exceptional circumstances, but you must **notify me (via email) at least one week beforehand**.

What to Hand In

To submit this assignment, zip or tar the following files together and submit them as a **single file named <netID>-hw1.tgz or <netID>-hw1.zip** on Canvas:

- $HANDIN_ROOT_DIR/report.pdf – a short report describing your optimizations and the results you see; see below for more details on the report
- $HANDIN_ROOT_DIR/gpgpu_sim – your checkout of the GPGPU-Sim repo, including any changes you made for the assignment.
- $HANDIN_ROOT_DIR/results – folder for the results output for each of the 5 benchmarks you are to run (see **Deliverables**). Note that GPGPU-Sim pipes this output to stdout by default – you will need to redirect it to a file. For each benchmark, your results file should be named *<benchmarkName>-results.txt* where benchmarkName is the name of the benchmark you are running (bfs, backprop, hotspot, nn, nw, optionally jacobi).

This means that your top-level directory should have three things: your report, your output files, and your version of the simulator. All files for GPGPU-Sim should be inside of that corresponding sub-folder. If you don't have experience with tar, I recommend consulting tutorials such as this one.

Total Points: 40

The breakdown for these points is:

- 2 points: turn-in contains appropriate files in appropriate places
- 8 points: your copy of GPGPU-Sim compiles (e.g., we can source the setup_environment file, then type make, and everything compiles successfully)
- 20 points:
  - 17 points: functional design works correctly
  - 3 points: your average performance for a set of benchmarks compared to others
- 10 points: report

Overview

For this assignment you will add a TLB to a GPU simulator, GPGPU-Sim. GPGPU-Sim is a detailed GPU functional and performance simulator that models NVIDIA's Fermi, Pascal, and Volta architectures with more than 90% correlation in most cases [Bakhoda '09, Khairy '18, Lew '19]. It executes Parallel Thread Execution code (PTX), a pseudo assembly ISA, compiled from both CUDA and OpenCL applications. As discussed in class, many of the architectural structures in a GPU are not fully disclosed by NVIDIA, the authors of GPGPU-Sim had to make assumptions about certain hardware structures by piecing together information from patents.

As CPUs and GPUs increasingly share data, it is important for programmability to have the memory is uniformly virtualized. Doing this efficiently for a GPU is challenging though, as GPUs have tens of thousands of threads, and could potentially require hundreds or thousands of translations per cycle. Moreover, the current implementation of GPGPU-Sim does not model TLBs. Prior work has shown that this can significantly impact the results for given applications [Pichai '14, Power '14]. In this assignment, your task is to add a simple TLB model to GPGPU-Sim and measure the impact this has on the performance of a small set of benchmarks.

By carrying out this assignment in GPGPU-Sim, you will be able to (a) describe/explain in greater detail what a timing simulator is and how to modify a timing simulator to carry out experiments, (b) explain how TLBs need to be adapted for GPUs, (c) explain/describe many other details of the microarchitecture and memory system of a modern GPGPU processor especially those related to address translation in a GPU pipeline.

The basic design of the TLB you will add to GPGPU-Sim is described in [Power '14]. You should use the provided gpgpusim.config configuration file as the baseline, as well as the provided config_volta_islip.icnt interconnect configuration file (Note: this corresponds to the baseline, correlated, tested files for the Titan V GPU: https://github.com/gpgpu-sim/gpgpu-sim_distribution/tree/dev/configs/tested-cfgs/SM7_TITANV). Depending on how you implement your TLB, you will likely need to add new configuration options to gpgpusim.config (e.g., for TLB associativity, size, etc.). Any updates you make to the gpgpusim.config will need to be copied to the folder for EACH application you are running.

Deliverables

Your goal is to implement a TLB mechanism and explore any interactions with the surrounding architecture and report your results quantitatively (using data tables and bar charts) and qualitatively (by explaining the data). In your experimentation, study any changes you can make to improve the effectiveness of the translation mechanism (you might want to understand any bottlenecks that limit its effectiveness).

Write a report of 2 pages including figures. IMPORTANT: in your report, interpret the data you collect — i.e., what does the data tell you? Do NOT just cut and paste the output of the simulator into your report. Submit your report through Canvas. To get ideas for how to interpret data, refer to the results section of the paper readings for the course. You should comment on quantitative

results and provide as much insight as you can relevant to getting better performance using stream buffers (what design choices did you need to make? why do you obtain the results you get? what do the results that say about the hardware being modeled and or the software being run? what can be done to improve the benefit of this translation mechanism, etc.). You should assume 4 KB pages and a 128-entry, fully associative L1 TLBs for your results.

Report results for BFS, Backprop, HOTSP, NN, NW, and optionally for your Jacobi implementation from Homework 0. Your results should minimally compare the performance without the TLB you added and with the TLB you added. Make sure you appropriately summarizing the data for all benchmarks across the different hardware configurations you explore (i.e., if you examine multiple different TLB approaches, you should include them in your graphs/tables; similarly if you explored different associativity's, sizes, etc., you should also include that in your graphs/tables). The benchmarks are stored here. Note that I've provided pre-compiled binaries for each non-Jacobi benchmark, so you shouldn't have to worry about compiling them. See the README in each file for the command to run.

TLB

Section 4.1 of Power's paper shows how they implemented their basic TLB approach ("Design 1"). You are responsible for implementing something similar to this. Note that the authors also propose more complicated TLB solutions that add features like multi-level TLBs and page caches. You may consider implementing these, since part of your grade is based on average performance.

Note that "Design 1" in Section 4.1 essentially places a TLB in front of the L1 cache. Since we are not actually converting from virtual to physical addresses in the simulator, your TLB implementation does not need to worry about translating the addresses. Rather, your TLB should be used to indicate whether the translation hits or misses. If it misses, you may assume that the page walker takes 20 (additional) cycles to find the correct translation and bring it into the cache. Your system should model this delay. On a hit, 1 cycle.

Just like a cache, you need to consider allocation, replacement, and eviction. To correctly model eviction, you would need to model TLB shootdowns and traffic between SMs. However, since we're only running 1 application at a time in the simulator, there wouldn't be any shootdowns. Thus, we can safely ignore shootdowns in your model. Thus, evictions will only happen due to space constraints. You may also ignore translations for constant memory and texture memory.

Suggested Approach

The following outlines the suggested procedure to most quickly complete this assignment. This is only a suggestion, there are multiple ways to approach the problem, which you are free to pursue instead if you feel it is easier to implement.

Step 1: First you will have to download GPGPU-Sim and the necessary NVIDIA files. To download GPGPU-Sim, go to www.gpgpu-sim.org to see the command to get the simulator (git clone https://github.com/gpgpu-sim/gpgpu-sim_distribution.git -- **use the dev branch** (**not the default release branch**)) and links to the documentation. You will be using the v4.0 version of

the simulator. Once you have this downloaded onto your machine, go to the GPGPU-Sim directory and read in entirety the README. As described in the README, you will also need to get the NVIDIA CUDA Toolkit v9.1 (available on CSL machines, see below for more details). Make sure to run "source setup_environment <buildType>" in the gpgpu-sim directory (normally buildType is "release", unless you are debugging GPGPU-Sim). Note: as long as you have the CUDA toolkit, you do not need a machine with a GPU to run the simulator.

Step 2: Read the GPGPU-Sim v3.x manual (still mostly relevant for v4.0 simulator), especially the parts about the cache structures, to understand how the simulator is structured and learn how to run the unmodified simulator.

Step 3: Read Power's paper "Supporting x86-64 Address Translation for 100s of GPU Lanes" – especially Section 4.1.

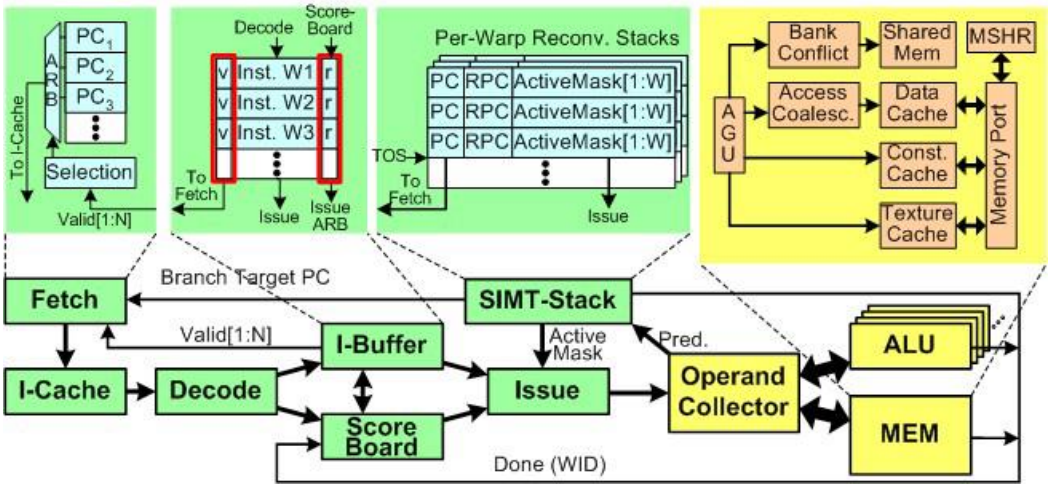
Step 4: Modify gpu-cache.cc/h, shader.cc/h, and gpu-sim.cc. The TLB will be implemented in gpu-cache.cc/h and the interfacing code will be written in shader.cc/h (The code to print your results will also be written in shader.cc). The option parsing code for the gpu-cache configuration will be written in gpu-sim.cc. You will have to study other portions of the simulator code to do this. See the next sections for hints and suggestions.

Step 5: Evaluate your design and tune your TLB design. See how your results change by adding/ removing TLB entries for each SM or by any other tuning methods you see fit.

Guide to (relevant parts of) GPGPU-Sim

Below is a guide to some of the relevant parts of GPGPU-Sim's source code:

- shader.[cc, h]: understanding the GPU pipeline.
    - These files contain many of the key functions used to model the GPU pipeline.
    - The GPU The GPU is composed of many shader cores (**shader_core_ctx** in GPGPU-Sim), each with their own pipeline. The main GPU pipeline for each shader core is implemented in shader.cc: **shader_core_ctx::cycle().** Similar to other simulators, the GPU pipeline is executed in reverse (writeback()→execute()→read_operands()→issue()→decode()→fetch()). A detailed view of the pipeline implemented in GPGPU-Sim is shown in the figure below.



- The part of the pipeline you will be interested is in the memory stage, and specifically the MEM functional units. To see how this piece of the pipeline is implemented, you'll want to look at the execute() stage (called from **shader_core_ctx**::**cycle**()). In this function, the execute stage cycles through all of the various functional units (e.g., Special Function Unit (SFU), load data store unit (LDST), and the Single Precision ALU units (SP)). For the MEM functional units, it calls **ldst_unit**::**cycle**() – in **shader_core_ctx**::**execute**() you will see:

```
m_fu[n]->cycle(); // m_fu may point to FP, SFU,
or LDST
```

which executes for each functional unit every cycle. **ldst_unit::cycle**() processes the memory operations, but before doing this execute() first calls:

```
writeback();
m_operand_collector >step();
```

Here writeback()checks which operations have finished executing on this cycle and need to write back their results to the register file.

Moreover, in between the register file and the functional units, there is some form of operand buffering. In GPGPU-Sim, this operand buffer is called the operand collector (speculated from NVIDIA patents). The purpose of the operand collector is to access the register file over one or more cycles (depending on bank conflicts) and store the operands in an intermediate location called the collector unit (similar to a reservation station). When all of the source operands are available, the instruction is able to execute and the corresponding collector unit is freed for a subsequent instruction. You should not need to change the operand collector for this assignment.

Once these calls are done, in **ldst_unit**::**cycle**() you will see a series of calls. The first block of code moves values between the various pipeline registers – much like

is done in a pipelined CPU. The second block of code is handling memory (data) replies coming back from the L2. The third block of code is cycling the L1 texture, constant, and data caches. Remember, cycling the caches is effectively processing the data that was sent to the caches in the previous cycle, since we process in reverse. Finally, the last block of code handles the current memory request by grabbing it from the dispatch register:

```
warp_inst_t &pipe_reg = *m_dispatch_reg;
```

Given the memory requests type, either the shared, constant, texture, or data cache will receive a new request:

```
done &= shared_cycle(pipe_reg, rc_fail, type);
done &= constant_cycle(pipe_reg, rc_fail, type);
done &= texture_cycle(pipe_reg, rc_fail, type);
done &= memory_cycle(pipe_reg, rc_fail, type);
```

For data requests **ldst_unit**::**memory_cycle**() will be used. This calls into the GPU cache files.

- gpu_cache.[cc, h]:
  - You will implement your TLB either in or using the existing gpu_cache. These files contain 4 main structures: *cache_block_t, cache_config, tag_array,* and the *cache_t/read_only_cache* (Note that you will be extending the *cache_t* or *read_only_cache* to handle writes, if you need to model writes to pages in your program). These structures are described below.
  - *cache_block_t:*
    - This structure implements the main cache block within the cache. It contains information about the block such as the tag, allocation time, access time, status, etc.
  - *cache_config:*
    - This structure contains information about the cache, such as line size, number of sets, associativity, replacement policy, etc. You will add a new configuration to the *gpgpusim.config* file to set these parameters appropriately.
  - *tag_array:*
    - This is one of the main structures in the cache. It contains an array of *cache_block_t* and handles all of the reads/writes to the cache. The functions of interest in this class are *tag_array::probe(...),* *tag_array::access(...),* and *tag_array::fill(...).*
  - **tag_array::probe(new_addr_type addr, unsigned &idx):**
    - This function takes in *addr*, extracts the index/tag information from the address, and searches the corresponding set within the cache for the tag. If it finds the cache block, the function returns HIT, otherwise the function will return MISS (or RESERVATION FAIL – not used in this assignment). When MISS is returned, probe sets *idx* to the cache way that is either

INVALID (not yet allocated), LRU (if LRU policy is set), or FIFO (if FIFO policy is set), which specifies the cache line that *addr* should be allocated to. This is an accessor function and does not modify any of the tag_array structures.

- **tag_array::access(new_addr_type addr, unsigned time, unsigned &idx, bool &wb, cache_block_t &evicted):**
  - This function first calls the probe function described above, and modifies the tag_array structures accordingly. On a cache hit, the access time for the cache line is updated. On a miss, depending on the allocation policy (allocate on miss or allocate on fill), the cache line chosen for allocation by the probe function is allocated to the requested cache line and if the existing cache line was modified, it is marked for writeback. For this assignment, you may want to set your cache allocation policy to ON_MISS, such that the line is allocated in this access function, and the line to be evicted is returned.
- *cache_t / read_only_cache:*
  - *cache_t* is the base template for the cache implementations in GPGPU-Sim. One example of how to extend this class is the *read_only_cache* class, in which additional functions are implemented/included, such as *access(...), fill(...), cycle(...),* etc. If you read through these functions, you'll notice that the majority of the code is there to handle writebacks to lower level memory or process incoming fill requests from lower level memory. For this assignment, we are utilizing a separate copy of the cache to act as a TLB, so a lot of this code is unnecessary. You will need to implement a *tlb* class that extends either one of these cache structures. The next section will discuss some hints of necessary code needed to implement this structure.

Suggested Implementation

- gpu-cache.[cc, h]

  For this assignment, you can likely get by without modifying any of the lower level cache structures (such as *block_t, tag_array, cache_config*, etc). Instead, you might want to create a *tlb* class that extends *read_only_cache* (such as is done by *data_cache*), as all of the necessary member variables are included (you could also extend *cache_t* and add in the necessary structures, such as a *tag_array*). Then you will need to modify the *access(...)* function, as well as implement your own *fill(...)* function.

  As was mentioned in the section above, most of the code in the *access/fill* functions for *read_only_cache* handle writebacks and requests coming from different levels of the memory hierarchy, which you may not need for this assignment. Thus, your *access(...)* function could simply determine whether or not the request hit in the cache and update the cache block status/LRU information accordingly. Similarly, the *fill(...)* function could locate the cache line to be evicted as well as fill the cache line with the incoming request. You may also want to utilize the MSHR requests, depending on how you model your TLB.

- shader.[cc,h] & gpu-sim.cc

Similar to the L1 instruction cache, each shader core will need to have a TLB. To see how to instantiate your TLB structure per shader (i.e. per *shader_core_ctx),* look at:

```
class shader_core_ctx : public core_t
```

in shader.h and the constructor in shader.cc (specifically, look at how the m_L1I instruction cache is instantiated). If you extended your *tlb* class from *read_only_cache*, your constructor requires a *mem_fetch_interface** and *mem_fetch_status.* You can set the interface to NULL and the status to the same as with the m_L1I.

Also required by your TLB constructor is a *cache_config* object (defined in gpu-cache.h). To do this, look at how the other cache configuration objects are defined / initialized in

`struct shader_core_config` (in shader.h)

You will need to add the option parsing code to gpu-sim.cc. This is located in the **shader_core_config::reg_options**(**class** OptionParser * opp) function (search for m_L1I_config for an example). Studying this code and the cache options in the gpgpusim.config file, it should be straightforward to add the necessary configuration parsing code for your TLB.

You should make your TLB configuration configurable (i.e. associativity, # sets, etc.). You will also need to look at the code in the *cache_config* class to see how the tag/index information is extracted from the requested addresses. Hint: look at the *cache_config::tag(...)* and *cache_config::set_index(...)* code in gpu-cache.h to see how the tag/index is extracted from an address.

Next, you will need to consider how to interface with your TLB. I recommend interfacing it in the same way as the L1 caches, but you could also consider putting it elsewhere.

Finally, you will need to print the results of your TLB. One method (of many) to do this is as follows. Look at the code in **gpgpu_sim::shader_print_l1_miss_stat**(FILE *fout) **const** in shader.cc. This function is called at the end of each kernel. This prints the results for the L1 data, constant, and texture caches. Traverse these function calls and add the necessary functions to interface with your TLB. In particular, we will want to know how many hits and misses occur in your TLB.

GPGPU-Sim Output

In addition to emitting output about the run to the screen (stdout), GPGPU-Sim also generates a number of temp files (e.g., 'bfs.1.sm_30.ptxas and _app_cuda_version_*). You can ignore these temporary files. It will also create a “checkpoint_files” repo – this is a new feature recently added to GPGPU-Sim, but you will not be using this support in this assignment, so you can also ignore that folder. The only GPGPU-Sim output you should need to do this assignment is what gets emitted to the screen.

Where to Run

For this assignment, you will be running your tests on *barolo.cs.wisc.edu*, an AMD CPU cluster in my research group. You should all now have access to the machine, if you do not have a CSL account or cannot access barolo, please let me know ASAP. Barolo has around 100 cores, which should be sufficient for all of you to run your experiments. However, keep in mind that it is a shared resource – please do not launch extra jobs or hog resources unnecessarily. I have sized the inputs for the programs such that memory consumption should not be a problem. You may instead run your experiments on a personal machine, if desired, assuming you install CUDA 9.1 first, but you will need a version of Linux that uses gcc-5.4.0 (I suggest Ubuntu 16.04).

Note that you do not need a "real" NVIDIA GPU installed on this machine to run GPGPU-Sim – all you need to run GPGPU-Sim is to have CUDA installed on the machine. You can access the CUDA compiler and libraries here: */s/cuda-9.1/amd64_ubu16/*. I recommend adding */s/cuda-9.1/amd64_ubu16/bin/* to your PATH, */s/cuda-9.1/amd64_ubu16/lib64/* to your LD_LIBRARY_PATH, and setting CUDA_INSTALL_PATH to */s/cuda-9.1/amd64_ubu16/* (e.g., in your .bashrc or .cshrc). This will give you access to CUDA 9.1 – which you'll use to compile the programs you run in GPGPU-Sim. Since the dev branch of GPGPU-Sim (which we are using) supports this version, you should go ahead and use it.

References

- Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural support for address translation on GPUs: designing memory management units for CPU/GPUs with unified address spaces. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (ASPLOS '14). ACM, New York, NY, USA, 743-758. DOI: https://doi.org/10.1145/2541940.2541942
- J. Power, M. D. Hill and D. A. Wood, "Supporting x86-64 address translation for 100s of GPU lanes," *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Orlando, FL, 2014, pp. 568-578. doi: 10.1109/HPCA.2014.6835965