Assignment 1

Overview

The objective of this assignment is for students to gain an understanding of how to use and modify GPGPU-sim.

Installation/Setup

The simplest way to install GPGPU-sim is to use the developer-provided virtual machine image.

1. Download the virtual machine from http://gpgpu-sim.org/ (http://gpgpu-sim.org/)
2. Install the VM. For a guide, refer to step 2 of http://www.danielwong.org/classes/_media/project_1.pdf (http://www.danielwong.org/classes/_media/project_1.pdf). Although it is for a different virtual machine and assignment, the steps should be similar.
3. Boot the virtual machine. The VM password: gpgpu-sim

Compiling GPGPU-sim

1. In VM, open terminal
2. `cd ~/gpgpu-sim_distribution`
3. `source setup_environment`
4. `make`

Compile Nvidia SDK 4.2

1. `sudo apt-get install libboost-dev libboost-system-dev libboost-filesystem-dev libboost-all-dev mpich2 binutils-gold libcuda1-304`
2. Edit `~/cuda/sdk/4.2/C/common/common.mk` , lines like " `LIB += … ${OPENGLLIB} …. $(RENDERCHECKGLLIB) …` " should have `$(RENDERCHECKGLLIB)` moved before `${OPENGLLIB}` . There should be 3 lines like this.
3. Similarly, edit `~/cuda/sdk/4.2/CUDALibraries/common/common_cudalib.mk`
4. `cd ~/cuda/sdk/4.2`
5. Edit Makefile. Comment all lines with `CUDALibraries` and `OpenCL` as we only want the application binaries. You comment by placing `#` in the front of the line.
6. `make`
7. This compiles helper libraries that other benchmark suites depends on as well as various CUDA binaries in `~/cuda/sdk/4.2/C/bin/linux/release` . These binaries can be run in GPGPU-sim.

ISPASS2009 Benchmarks

1. `git clone https://github.com/gpgpu-sim/ispass2009-benchmarks.git` (https://github.com/gpgpu-sim/ispass2009-benchmarks.git)
2. `export NVIDIA_COMPUTE_SDK_LOCATION=~/cuda/sdk/4.2`
3. `cd ispass2009-benchmarks`
4. Edit `Makefile.ispass-2009`
    a. comment out line 16 and 28. The AES and WP benchmark does not compiler readily.
    b. Change `BOOST_LIB` to the following path: `/usr/lib/x86_64-linux-gnu/`

c. Change `BOOST_ROOT` to the following path: `/usr/include/boost/`
d. Change `OPENMPI_BINDIR` to the following path: `/usr/bin/`

5. Edit `DG/Makefile`
   a. Line 54, change `-I/opt/mpich/include` to `-I/usr/include/mpich`
   b. Line 56, append `-I/usr/include/mpich` to end of the line
   c. Line 59-61, add `.mpich2` to the end of each line. For example, `NEWCC = $(OPENMPI_BINDIR)mpicc.mpich2`
6. `make -f Makefile.ispass-2009` . Note: If you do a `make -f Makefile.ispass-2009 clean` , you may have to recompile the SDK.

How to run a program

1. `source ~/gpgpu-sim_distribution/setup_environment` (You only have to do this once per terminal session)
2. `mkdir test; cd test`
3. Copy configuration files to test directory: `cp -a ~/gpgpu-sim_distribution/configs/GTX480/* .`
4. Now run a binary. For example, to run vectorAdd
   `~/cuda/sdk/4.2/C/bin/linux/release/vectorAdd`
   As another example, we can run the BlackScholes binary in the CUDA SDK.
   `~/cuda/sdk/4.2/C/bin/linux/release/BlackScholes` .
   As yet another example, we can run the CP binary in ISPASS2009. `~/ispass2009-benchmarks/bin/release/RAY 4 4` .

Assignment Details

GPGPU-Sim Warp Schedulers

GPGPU-Sim by default comes with several warp scheduling policies implemented. Specifically, they are the Loose round robin (LRR), Two-level (TL), and Greedy then oldest scheduler (GTO).

The Loose round robin scheduler is a basic naive scheduler which simply picks warps to issue in order (ie. issue 1, issue 2, issue 3,…). If a warp is not ready to issue, it will be skipped until its turn during the next iteration (hence, loose round robin).

The Two-level scheduler separates the warps in an SM into two pools: an active pool and a pending pool. The warps inside the pending pool are waiting on long latency events, such as memory access. The warps in the active pool are either ready, or will be ready shortly (within the next few cycles). The scheduler selects warps from the active pool to issue in a loose round robin manner.

The Greedy then oldest scheduler issues instructions from the same warp until that warp has stalled. At this point, it picks the oldest ready warp to issue from.

Assignment Details

The goal of this homework assignment is to explore the effect that warp scheduling has on performance, specifically the effect on L1 cache and global memory.

Your job is to change the configuration file to run benchmark with each of the three schedulers (-gpgpu_scheduler flag) and answer the following questions. Justify your answers from the data you obtained, along with your knowledge of GPGPUs as discussed in class, to justify your answer.

Some of the existing statistics may not be sufficient to understand why a particular behavior is observed. If existing statistics are not sufficient to answer the question, you need to add custom statistics to the warp scheduler code to gain deeper insight into the interaction between scheduler and cache/memory.

While you can create multiple statistics that you need, the one required new statistic that you need to collect is the memory instruction inter-issue distance. This statistic measures the number of cycles that have passed between two consecutive issued memory instructions. Please plot a histogram of the elapsed cycles by placing all the elapsed cycle counts into 10 buckets. You can pick the range of these ten buckets as you wish.

Some hints: Use the min and max inter-issue cycles you measure from the new statistics to create the 10 bucket ranges. To add the statistics look at the `cycle()` method of class scheduler_unit in `shader.cc/.h` in `src/gpgpu-sim` . The inherited classes (`lrr_scheduler, gto_scheduler, and two_level_active_scheduler`) will be of interest to you as do this assignment.

Benchmarks

For this assignment, we will make use of three ISPASS2009 benchmarks: BFS, CP and RAY. The command to run each is below:

```
./BFS ~/ispass2009-benchmarks/BFS/data/graph4096.txt
```

```
./CP
```

```
./RAY 32 32
```

Note: output statistics are generated at the end of each kernel. Some benchmarks may have multiple kernel, so be sure to use the results from the last kernel only.

Questions

Each question must be answered with supporting data

1. What is the runtime for each configuration?
2. Which scheduler has the best cache hit rate? Why?
3. Which scheduler resulted in the least amount of pipeline stalls due to memory access (gpgpu_n_stall_shd_mem)? Why?
4. In what scenarios would you prefer a Two-level scheduler? Greedy then oldest scheduler?

Submission

Please submit typed document answering the above questions along with justification data on iLearn. In addition to these answers please plot the histogram of the elapsed cycles between two issued memory instructions.