1 The Installation of GPGPUsim

I. System Environment.

I installed GPGPUsim along with CUDA in my virtual machine system. The system environment is given as following

Host OS	Mac OS X 10.9
Host CPU	Intel i7-3740QM @ 2.70GHz × 4 Cores
Virtual Software	Vmware Fusion 6.03
Virtual CPU setting	Intel i7-3740QM @ 2.70GHz × 2 Cores
Vitual Graphics setting	Software default approach
Virtual System Memory	4 GB
Virtual System OS	Ubuntu 12.04 64bit desktop
GCC version	4.6
GPGPUsim version	v3.x
CUDA version	4.2.9

Table 1. System Environment

The reason I didn't install it under my operating system, Mac OS X, is that GPGPUsim itself is designed for linux-kernel OS, and even the system of Mac is unix-like, there remains a lot of functions to be modified in the source code, such as the function of `fopen64()` under linux is not valid for Mac. However it is possible to run the program under Mac and there is one of my classmates managed to do it.

II. Install Steps.

Solve software dependence → Install CUDA → Install GPGPUsim

Table 2. Install Flow Work

III. Software Dependence.

Though each installation step request defferent dependence, the way to solve over-all software dependence is given as following

```
sudo apt-get install git-core build-essential xutils-dev bison zlib1g-
dev flex libglu1-mesa-dev libxi-dev libxmu-dev freeglut3-dev binutils-
gold libcudart4 libboost-system-dev libboost-filesystem-dev libopenmpi-
dev openmpi-bin openmpi-dev gfortran
```

IV. Install CUDA.

1. Download CUDA toolkit and CUDA sdk (version 4.2) from NVIDIA.

cudatoolkit_4.2.9_linux_64_ubuntu11.04.run
gpucomputingsdk_4.2.9_linux.run

Table 3. CUDA Installation File

2. Install CUDA toolkit.

```
sudo bash cudatoolkit_4.29_linux_64_ubuntu11.04.run
```

Using the default install path. The installation successes if 'Installation Complete' is shown.

3. Install CUDA sdk.

```
sudo bash gpucomputingsdk_4.2.9_linux.run
```

Using the default install path. The installation successes if 'Installation Complete' is shown.

4. Compile CUDA sdk demo.

```
cd ~/NVIDIA_GPU_Computing_SDK/C/
sudo make
```

5. Download GPGPUsim.

I'm keeping the files under the user's folder

```
cd ~
git clone git://dev.ece.ubc.ca/gpgpu-sim
```

6. Modify global path.

The installation script for GPGPUsim uses global path to find its dependence. So before installation, paths should be add to the bash profile. Editing `~/.bashrc` and add the following code to the end of the file

```
export CUDA_INSTALL_PATH="/usr/local/cuda"
export NVIDIA_CUDA_SDK_LOCATION="/home/USER_NAME/
NVIDIA_GPU_Computing_SDK"
export PATH=${PATH}:/usr/local/cuda/bin
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/local/cuda/lib
export GPGPUSIM_ROOT=/home/USER_NAME/gpgpu-sim/v3.x
```

When `USER_NAME` is the user's name as to indicate user's folder. Restart Terminal or refresh by

```
source ~/.bashrc
```

The installation can forward now.

7. Installation GPGPUsim.

I'm using the version of v3.x.

```
cd ~/gpgpu-sim/v3.x
source setup_environment
```

It will check your path and environment setting. The settings successes if 'Successed' is shown. Then compile the sources by

```
make
```

8. Compile benchmarks.

   i. navigate into the benchmark folder

```
cd ~/gpgpu-sim/ispass2009-benchmarks
```

ii. Setting path

as the README indicates, adding the following code to the beginning of `Makefile.ispass-2009`

```
CUDA_INSTALL_PATH=/usr/local/cuda
NVIDIA_COMPUTE_SDK_LOCATION=/home/USER_NAME/N
VIDIA_GPU_Computing_SDK
```

the same as in `~/.bashrc`.

iii. Modify makefiles for special pathes

The compile script uses pathes to find its dependence, however as software and system changes, some of these settings change and hence by defult setting the makefile cannot be used. We would modify the makefiles in order to make the compling work.[1]

- `Makefile.ispass-2009`

  change

  ```
  export OPENMPI_BINDIR=/usr/lib64/mpi/gcc/openmpi/bin/;
  ```

  to

  ```
  export OPENMPI_BINDIR=/usr/bin/;
  ```

  This fixes the dependence of `mpi`.

- `/AES/Makefile`

  under `LINKFLAGS`, change `-lboost_filesystem` to `-lboost`. This fixes the dependence of `libboost`.

- `/DG/Makefile`

  under `INCLUDES`, add `-I/usr/lib/openmpi/include` to the end. This fixes the dependence of `mpi`.

- `/DG/3rdParty/ParMetis-3.1/Makefile.in`

  under `INCDIR`, add `-I/usr/lib/openmpi/include` to the end. This fixes the dependence of `mpi`.

And so far we successfully installed GPGPUsim.

2 Run the Program

I. Set the Environment .

Once we get GPGPUsim installed, it's time to get our hands dirty. The GPGPUsim provides a virtual environment which allows CUDA to be executed under non-NVIDIA GPU systems. To do that, first we should set the environment. starting by

```
cd ~/gpgpu-sim/v3.x
source setup_environment
```

which provides paths & links for system variables.

1. The Following steps is only correct under particular system & software enviroments.

II. Choose GPU Type.

GPGPUsim provides some typical GPU types to simulate, such as GTX480 for personal market, or QuadroFX5600 for professional market. When executing CUDA binary programs, GPGPUsim uses GPU profile in the format of `.xml` to determine what kind of GPU to simulate. In common cases, the profile should be under the same path with binarys. The profiles are in the folder of `gpgpu-sim/v3.x/configs`.

III. Executing SDK Demo.

The NVIDIA CUDA SDK provides some example sources, which we have compiled to binaries before. In order to excute those programs, copy the GPU profiles into the binary's folder

```
cp ~/gpgpu-sim/v3.x/configs/GTX480/gpuwattch_gtx480.xml ~/
NVIDIA_GPU_Computing_SDK/C/bin/linux/release/gpuwattch_gtx480.xml
```

and then execute the binary, for example, `vectorAdd`:

```
cd /home/zen/NVIDIA_GPU_Computing_SDK/C/bin/linux/release
./vectorAdd
```

Only some of programs in the NVIDIA CUDA demo can be executed properly. Programs with a visual interface cannot be executed. Some other programs that can be executed takes a long time up to several hours.

IV. Executing Benchmark.

GPGPUsim comes with a benchmark. As we have compiled the benchmarks, it's easy to follow the `README` to execute them. One good thing is GPU type setting is automaticly done by

```
cd ~/gpgpu-sim/ispass2009-benchmarks
bash setup_config.sh GTX480
```

I modified the script `setup_config.sh` hence it automaticly copy the `.xml` file into the directory folder

```
# Change clean step
if [ $1 = "--cleanup" ]; then
    echo "Removing existing configs in the following directories:"
    for BMK in $BENCHMARKS; do
        if [ -f $BMK/gpgpusim.config ]; then
            echo "$BMK"
            OLD_ICNT=‘awk ’/-inter_config_file/ { print $2 }’ $BMK/
gpgpusim.config‘
            rm $BMK/gpgpusim.config $BMK/$OLD_ICNT
        fi
        if [ -f $BMK/gpuwattch*.xml ]; then
            rm $BMK/gpuwattch*.xml
        fi
    done
    exit 0
fi

# Adding the varible of profiles
GPU_WATTCH_FILE=$GPGPUSIM_ROOT/configs/$GPGPUSIM_CONFIG/gpuwattch*.xml
if [ -f $GPU_WATTCH_FILE ]; then
    echo "Found GPGPU-Sim config file: $GPU_WATTCH_FILE"
```

```
else
    echo "Unknown config: $GPGPUSIM_CONFIG"
    exit 0
fi

# Modify link step
for BMK in $BENCHMARKS; do
    if [ -f $BMK/gpgpusim.config ]; then
        echo "Existing symbolic-links to config found in $BMK!
Skipping... "
    else
        echo "Adding symbolic-links to configuration files for $BMK:"
        ln -v -s $GPU_CONFIG_FILE $BMK
        ln -v -s $ICNT_CONFIG $BMK
        ln -v -s $GPU_WATTCH_FILE $BMK
    fi
done
```

and the executing is also automatic, for example, running the benchmark for BFS by

```
cd ~/gpgpu-sim/ispass2009-benchmarks/BFS
bash README.GPGPU-Sim
```

The benchmark of `DG`, `MUM`, `LIB` and `WP` takes relatively very long time to execute.

3 Code Review for GPGPUsim

The SIMT core microarchitecture shown below

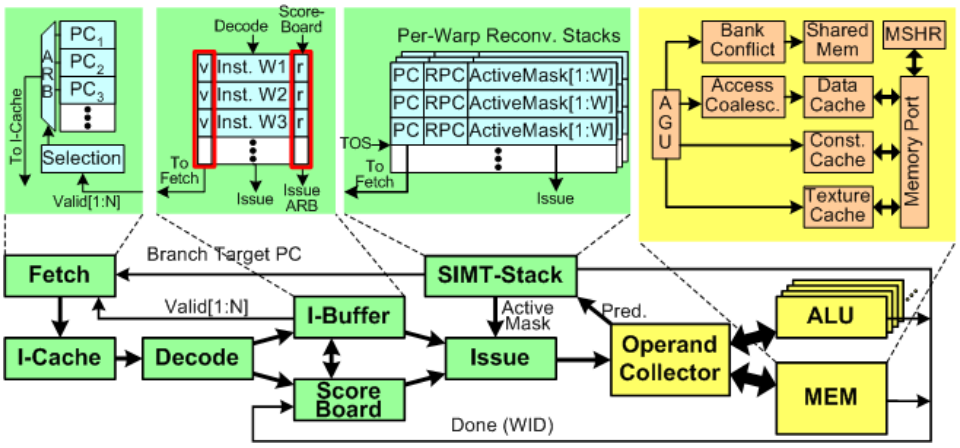

Figure 1. SIMT core microarchitecture

is implemented with the class `shader_core_ctx` in `shader.h/cc` . Derived from class `core_t`(the abstract functional class for a core), this class combines all the different objects that implements various parts of the SIMT core microarchitecture model:

- A collection of `shd_warp_t` objects which models the simulation state of each warp in the core.
- A SIMT stack, `simt_stack` object, for each warp to handle branch divergence.
- A set of `scheduler_unit` objects, each responsible for selecting one or more instructions from its set of warps and issuing those instructions for execution.

- A Scoreboard object for detecting data hazard.
- An `opndcoll_rfu_t` object, which models an operand collector.

  A set of `simd_function_unit` objects, which implements the SP unit and the SFU unit (the ALU pipelines).
- A `ldst_unit` object, which implements the memory pipeline.
- A `shader_memory_interface` which connects the SIMT core to the corresponding SIMT core cluster. Each memory request goes through this interface to be serviced by one of the memory partitions.

Every core cycle, `shader_core_ctx::cycle()` is called to simulate one cycle at the SIMT core. This function calls a set of member functions that simulate the core's pipeline stages in reverse order to model the pipelining effect:

- `fetch()`
- `decode()`
- `issue()`
- `read_operand()`
- `execute()`
- `writeback()`

The various pipeline stages are connected via a set of pipeline registers which are pointers to `warp_inst_t` objects (with the exception of Fetch and Decode, which connects via a `ifetch_buffer_t` object).

Each `shader_core_ctx` object refers to a common `shader_core_config` object when accessing configuration options specific to the SIMT core. All `shader_core_ctx` objects also link to a common instance of a `shader_core_stats` object which keeps track of a set of performance measurements for all the SIMT cores.

4 Change RF's Latency [2]

I. Modify for register read

   The `opndcoll_rfu_t::allocate_reads(...)` processes read requests that do not have conflicts, in other words, the read requests that are in different register banks and do not go to the same operand collector are popped from the arbitrator queues. This accounts for write request priority over read requests.

   In order to increase read cycles, we add a counter for each bank that counts its current executing cycle. The units start to count when it is assigned to one read request. When allocating register units, the function set all the banks that remain in self-counting negetive, and only adding those counts to 2-cycles to read committing queue.

II. Modify for register write

   The function `opndcoll_rfu_t::writeback( const warp_inst_t &inst )` is called at the write back stage of the memory pipeline. It is responsible to the allocation of writes.

2. See into `abstract_hardware_model.h`, `shader.h`, and `shader.cc`.

It allocate writing banks and calls `void shader_core_ctx::writeback()` to commit writing. The original codes assumes the pipelines are un-stallable, and hence it commits the writing request no matter whether the operand collector has return a positive answer. Our approach to increase latency by increasing writing cycles, is indeed to add stalls, so the original's assumption fails and the structure should be changed.

The first attemption is to instead of letting the writing requst go, catch the return value of `m_operand_collector.writeback(*pipe_reg)` and decide if the writing progress should be commit.

The second attemption is to add a counter to count for pipline stalls.

The third attemption is to add a counter for each register units. Any register under the writing progress should be busy at the first cycle, and get ready after 2 cycles.

III. Modify for unit register file[3]

We make half of the bank using 1-cycle read&write while the other half suffer 2-cycle. This is done in `shader.h` when initing arrary of the banks' cycle.

The codes will be annexed to this document. All modified or new variables & functions are under the name of `mod_name`.

5 Runing Results

We select 6 benchmarks executed in reasonable time to compare the performance of different latency set.[4]

Program	1 cycle latency	half-half latency	2 cycle latency
AES	39769	44731	49248
BFS	386621	790582	786140
CP	241538	249909	250941
NQU	35116	38810	43516
RAY	128017	139091	147432
STO	362728	387531	414113

Table 4. Overall gpu total cycles of different latency set

The results go well with our intuition: the performance of 1-cycle latency, half-to-half, and 2-cycle latency decreases progressively. The over-all performance decrease from 1-cycle to 2-cycle latency is around $10\% \sim 20\%$, however the results for BFS shows a great change in cycles, even a jump from 1-cycle to half-half. This may be becauce BFS deals a lot of work in reading and writing files, and the slight increase from half-half to 2-cycle latency may be because non-ordered read&write steps disrrupt the pipeline.

6 Review and Comments

- This work helps me build a strong structure of how pipeline on GPU works, and gives me a brief view of GPU computing. It's happy to have this experience.

- I'm not quite sure about my modifying codes, especially the cases for writing: it seems the only way to solve the problem is to rewrite the pipeline struture, rather than trivially adding the counter.

3. There are two copies of `shader.h` in order to distinguish between 2-cycle case and half-half case.

4. The programs' outputs for 1-cycle latency is under the folder `results0`, 2-cycle `result1`, half-half `result2`.

- Time flys in the final days and remains little have I spent on this project. I'm sorry that I didn't quite understand the software's structure and use much, and sorry for the project being not handled in time.

- And last but not the least, thank you for all your help in the last semester.