MS108 COMPUTER SYSTEM(1)

Final Report — gpgpu-sim

Kaichun Mo
512030952

June 20, 2014

CONTENTS

1 Introduction

GPGPU-Sim is a simulator aiming at collect a bunch of statistics out when executing an GPGPU program via CUDA or OPENCL. Using this simulator, it’s more convenient for GPU related researchers or programmers to test and improve the efficiency of their projects or programs without the large amount of expense on the purchase of GPU hardware.

In this report, I will show you my experiment on GPGPU-Sim. First, I set up the necessay running environment on my computer and install the simulator successfully, even though I encounters with many troubles and errors during this phase. Next, I scan through the manual and the codes for basic observations both on the architecture for GPU and the basic structure of this simulator. And then, I run the simulator on some benchmarks and obtain a bunch of results. Finally, as required, I focus on the operand collector units and spend much of time to discover how this part works. After that, I modify the latency of operands reading and writing from GPU register files to different values and re-run the simulator on the benchmarks to compare the differences on the results.

2 Installation and Environments Setup

I install GPGPU-Sim on Ubuntu 12.04. Since the compilation processes of CUDA benchmarks requires CUDA libraries and other dependencies, we also need to installCUDA Toolkit and CUDA SDK. Following is the details about what we need.

- **Cuda Toolkit:** cudatoolkit_4.0.17_linux_64_ubuntu10.10.run
  - which can be downloads from: https://developer.nvidia.com/cuda-toolkit-40
- **Cuda SDK:** gpucomputingsdk_4.0.17_linux.run
  - which can be downloads from: https://developer.nvidia.com/cuda-toolkit-40
- **Gpgpu-sim:** git clone git://dev.ece.ubc.ca/gpgpu-sim

– which can be downloads from: https://gpgpu-sim.org

Before compiling them, we need many depencies to install first. Here is all we needs.

```
sudo apt−get install build−essential xutils −dev bison zlib1g−dev flex libglu1−mesa−dev reeglut3−dev
          binutils −gold libboost−system−dev libboost−filesystem−dev libopenmpi−dev openmpi−bin openmpi−
        dev gfortran
```

Followings are my steps to compile them all.

2.1 Install Cuda Toolkit

Download the run file and just run it.

```
sudo ./cudatoolkit_4.0.17_linux_64_ubuntu10.10.run
```

Every thing is just fine, no error occurs.

2.2 Install Cuda SDK

Download the run file and try running it.

```
sudo ./gpucomputingsdk_4.0.17_linux.run
```

And then, we need add some lines to /home/mkch/.bashrc. Here you need use your own name instead of mine.

```
export CUDA_INSTALL_PATH="/usr/local/cuda"
export NVIDIA_COMPUTE_SDK_LOCATION="~/NVIDIA_GPU_Computing_SDK"
export PATH=$CUDA_INSTALL_PATH/bin:$PATH
export GPGPUSIM_ROOT=~/gpgpu−sim/v3 . x/
export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH
```

If you choose to follow the default installing locations, you can find a new archive in/home/mkch, which is named NVIDIA_GPU_Computing_SDK.

Now, let’s begin to compile it.

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

I encounter with an error, which says

```
/usr/local/cuda/include/host_config.h:82:2: error #error −− unsupported GNU version! gcc 4.5 and up
          are not supported!
```

It shows that the installer does not support higher gcc, which means we need to installgcc-4.4 and g++-4.4 to continue the compiling process. Follow the below steps to solve this problem.

```
sudo apt−get install gcc−4.4 g++4.4
sudo ln −s /usr/bin/gcc−4.4 /usr/local/cuda/bin/gcc
sudo ln −s /usr/bin/g++−4.4 /usr/local/cuda/bin/g++
```

The last two commands aim at generating a link to somewhere in the environment path. Note that this will work only if you have done this

```
export PATH=$CUDA_INSTALL_PATH/bin:$PATH
```

to your .bashrc.

Now, continue the compiling process and we get another error, which says

```
/usr/bin/ld: error: cannot find −lcuda
obj/x86_64/release/deviceQuery.cpp.o:deviceQuery.cpp: function main: error: undefined reference to '
        cuDeviceGetAttribute '
```

which is probably due to the lack of some dynamic linking libraries. I choose an easy way to solve this problem, which is just to install CUDA driver that should contain what we need. Type the following commands in the terminal to install CUDA driver.

```
sudo apt−get install nvidia −304
```

Then, just type

```
sudo make
```

to continue and I successfully finished the compilation of CUDA SDK.

2.3 Install GPGPU-Sim

This is very easy to compile the simulator itself, since the compiling process of GPGPU-Sim does not have too much dependencies with CUDA libraries. I download GPGPU-Sim and put the archive in /home/mkch. Then, do the following steps to compile v3.x version of GPGPU-Sim.

```
cd ~/gpgpu−sim/v3 . x
source setup_environment
make
```

To ease the code reading phase, I also generate the documents out using a tool named doxygen.

```
make doc
```

This is the most easy installation process until now.

2.4 Compile Ispass-2009 benchmarks

Now, I continue to compile the benchmarks in GPGPU-Sim, which can be find in/home/mkch/gpgpu-sim/ispass2009-benc

First, we need to add the following two lines at the top of Makefile.ispass-2009 to tell some environment configurations to it.

```
CUDA_INSTALL_PATH=/usr/local/cuda
NVIDIA_COMPUTE_SDK_LOCATION=/home/mkch/NVIDIA_GPU_Computing_SDK
```

Then, type this code to activate the compiling process.

```
make −f Makefile . ispass −2009
```

During the compiling process, I have encountered with a bunch of problems.

- **Error One:**

  ```
  /error: undefined reference to 'boost::system::generic_category()'
  ```

  The reason is that my system lacks of some necessary dependencies. First, download all the dependency packages.

  ```
  sudo apt−get install libboost−system−dev libboost−filesystem−dev
  ```

  After installing those dependencies, we still need to modify some files to continue the compilation. I figure out the file /AES/Makefile and change

  ```
  LINKFLAGS                    := −L$(BOOST_LIB) −lboost_filesystem$(BOOST_VER)
  ```

  to

  ```
  LINKFLAGS                    := −L$(BOOST_LIB) −lboost_system −lboost_filesystem$(BOOST_VER)
  ```

  which solves this problem.

- **Error Two:**

  ```
  usr/lib64/mpi/gcc/openmpi/bin/mpicc: no such command
  ```

  Maybe we need to install mpicc in bash.

  ```
  sudo apt−get install libopenmpi−dev openmpi−bin openmpi−dev
  ```

  Then, we need to modify Makefile.ispass-2009 to indicate the location of mpicc. Type

  ```
  whereis mpicc
  ```

  to figure out where mpicc just was installed. I find that my mpicc was installed in /usr/bin. Thus, I do the following modification to Makefile.ispass-2009.

  ```
  OPENMPI_BINDIR=/usr/bin/
  ```

- **Error Three:**

When compiling files in DG, an error occurs.

```
src/Mesh3d.c:1: fatal error: mpi.h: no such file or archive
```

We need to modify DG/Makefile a little bit. More specific, change

```
ifeq($(HOSTNAME),1)
INCLUDES = −Dp_N=$(N) −DNDG3d −DCUDA −I/opt/local/include −I/usr/include/malloc −I$(HDRDIR)
        −I/opt/mpich/include
else
INCLUDES = −Dp_N=$(N) −DNDG3d −DCUDA −I/opt/local/include −I/usr/include/malloc −I$(HDRDIR)
endif
```

to

```
ifeq($(HOSTNAME),1)
INCLUDES = −Dp_N=$(N) −DNDG3d −DCUDA −I/opt/local/include −I/usr/include/malloc −I$(HDRDIR)
        −I/opt/mpich/include −I/usr/lib/openmpi/include
else
INCLUDES = −Dp_N=$(N) −DNDG3d −DCUDA −I/opt/local/include −I/usr/include/malloc −I$(HDRDIR)
          −I/usr/lib/openmpi/include
Endif
```

- **Error Four:**

  When compiling files in WP, an error occurs.

  ```
  gfortran −E −C −P libmassv.F > libmassv.f90
  ```

  This is easy to solve by just installing gfortan as follows.

  ```
  sudo apt−get install gfortan
  ```

After fixing the above errors, I successfully compile all banchmarks.

Page 6

2.5 Run the benchmarks

To run the benchmarks we just finished compiling, one need to follow the following steps.

First, to set up the environment configurations which aims at re-directing the CUDA programs to GPGPU-Sim, we need process

```
./setup_environment GTX480
```

After doing that, we find many links have been placed in different benchmark folders. Meanwhile, an archive named common have been created in the root folder of GPGPU-Sim.

Next, I try to run benchmark AES as follows.

```
cd AES
sh README.GPGPU−Sim
```

But, unfortunately, I encounter with another error, which says

```
XML Parsing error inside file 'gpuwattch_gtx480.xml'.
 Error: File not found
```

This is just a linking problem. After finding the location of gpuwattch_gtx480.xml, I do the following linking process.

```
sudo ln −s . . / . . / v3 . x/ configs /GTX480/gpuwattch_gtx480 . xml .
```

And, everything is OK. The simulator begins generating a bunch of statistics for me.

3 Basic Observations and Simulations

Following are the brief introductions to GPU microarchitecture and GPGPU-Sim basic structure. During this phase, I read a lot of materials to get more knowledges about how GPU are designed and how GPGPU-Sim simulates the GPU processor.

3.1 Brief Description of GPU Microarchitecture

GPU core is designed to run multiple threads together to dramatically shorten the execution time. It contains many SIMT cores, which is called SM(Streaming Multiprocessor) by NVIDIA. Each SM contains many SP's and SFU's which are respectively execute one thread each cycle using the pipeline technique. Following is a picture to illustrate the microarchitecture of a GPU processor.

Figure 3.1: Detailed Microarchitecture Model of SIMT Core

When we use GPU to run a high-parallelism-featured program, we cut out the part that can be executed using GPU to speedup and call them *kernel*'s. Each *kernel* consist of a great many *thread*s to be executed. The whole threads in one kernel are divided to many *CTA*'s or *Thread Block*'s. Each CTA can be divided into a bunch of *warp*'s, which are the basic executing unit in GPU architecture.

Each warp contains no more than 32 threads. The warps are allocated to one unique SM when executing. While multiple warps can be allocated to one SM to be scheduled during process, there must be exact one warp can be executed in one cycle.

From the manual I know that in one SM the warps are executed using pipeline technique and interleaving technique as well. The stages are: *fetch*, *decode*, *issue*, *read operands*, *execute* and *writeback*. Since we only need to care about the *read operands* stage, I just explore more knowledges about the operand collector unit in SM. Following picture shows the architecture of each operand collector unit.

Figure 3.2: Operand collector microarchitecture

Each operand collector contains one *arbitrator* whose job is to schedule all the reading and writing requests to the register files, many *bank*s which stores all the register operands, one *crossbar*, many *collector unit*s and the other connections.

Here, collector unit is quite unfamiliar for me. I read some materials on manual and figure out that collector unit is used for holding the requests of each instruction. Every instruction is allocated one free collector unit at issue stage. And then, the required registers are needed to actually execute this instruction. Since every reading requests must be sent to arbitrator, when this operand is finished, the arbitrator will modify one specific bit in collector unit. Whenever all source operands are ready, the instruction can be pushed to execution stage. The collector unit is freed after *writeback* stage.

3.2 Brief Description of GPGPU-Sim Simulator

This GPGPU-Sim Simulator can be roughly divided into three parts: *functional simulator*, *performance simulator* and *interconnnection network simulator*. Functional simulator simulate the exact execution procedure of programs, while performance simulator accounts for collect statistical results during the process of one program. The statistical results includes the total simulated cycle and many other detailed statistics. Interconnnection network simulator simulates the network environment when executing one program. For instance, when reading an operand from main memory, the data should be placed in somewhere between functional unit and memory.

Basically, the whole execution process of GPGPU-Sim is a mixture of functional simulation and performance simulation. For example, when executing an real functional unit we need use functional simulator, while when we want to collect the cycle spent during operands collection we need call performance simulator for help.

To be more practical, each basic unit are implemented using C++ class object. Following is some important ones.

Basic Units	**Class Name**
SIMT core	shader_core_ctx
Operand Collector	opndcoll_rfu_t
Instruction	warp_inst_t
Warp	shd_warp_t
Functional Unit	simd_function_unit
Memory Pipeline	ldst_unit

The class opndcoll_rfu_t contains many interior classes, which are

Page 9

Interior Units	Class Name
Operand Data	op_t
Operand Allocation Status	allocation_t
Arbitrator	arbiter_t

Each cycle, void shader_core_ctx::cycle() will be called to simulate one cycle for all possible events in SM. A sequence of pipeline stages are all executed one cycle, which are writeback(), execute(), read_operands(), issue(), decode() and fetch(). To simulate the effect of pipeline, we inverse the execution sequence.

Now, let's focus on the stage related to register files. The function read_operands() contains nothing since we do not need to really read operands during performance simulation. But, performance simulator need store the information about one reading or writing request and push the instruction to execution stage whenever the operand is ready.

After discovering the interior structure of codes in opndcoll_rfu_t, I find that

- Each instruction is allocate to an unique collector unit during issue stage. All the required source registers are marked *not ready* in collector unit.
- In function shader_core_ctx::execute(), the following codes appear.

```
for ( unsigned c=0; c < multiplier; c++ )
                m_fu[n]−>cycle () ;
```

  which lets each functional unit to simulate one cycle. This simulation may be done during functional simulator. But, when the instruction need to read operands from register files, a function named ldst_unit::cycle() is invoked. At the beginning of this function,

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

  invokes the function opndcoll_rfu_t::step() to process the reading and writing requests to register files.

- In function opndcoll_rfu_t::step(), it invokes many other functions.

```
void step ( )
{
          dispatch_ready_cu () ;
          allocate_reads () ;
          for ( unsigned p = 0 ; p < m_in_ports.size(); p++ )
                    allocate_cu( p );
          process_banks ( ) ;
}
```

  This function executes the following jobs inversely.

  – Tell all the banks the fact that one cycle passed.

Page 10

– Allocate new instructions to free collector units.

– For each waiting queue, which lists all remain requests to the specific bank, process the first request one cycle.

– Dispatch all ready instructions to execution stage.

which concludes that my job to modify the reading and writing latency for each request to register files is to change this four functions a little bit.

Next, I begin to modify the codes to fulfill my goal. Before showing you how I change the codes, I will show you the results generated from original 1-cycle-configuration using different benchmarks.

3.3 Results on Different Benchmarks

I run three different benchmarks from ispass-2009, which are CP, NQU and RAY. Following is the result.

benchmark	gpu_tot_sim_cycle
CP	241538
NQU	35987
RAY	140784

Next, I will begin to modify the register bank latency to more than one cycle.

4 Modifications to GPGPU-Sim

4.1 Modify All Latencies to two

(1) Modify RF Reading Requests For the reading requests, the allocation for operand is done in one cycle originally in the function opndcoll_rfu_t::allocate_reads(). Following is the segment of this function.

```
for ( std::list<op_t>::iterator r=allocated.begin(); r!=allocated.end(); r++ ) {
        ...
        m_arbiter . allocate_for_read (bank, rr ) ;
        read_ops[bank] = rr;
    }
```

where read_ops save all registers which are ready. After computing read_ops out, it do the following.

```
for (r=read_ops.begin();r!=read_ops.end();++r ) {
        ...
        m_cu[cu]−>collect_operand (operand) ;
        ...
   }
```

Page 11

wherecollector_unit_t::collect_operand(unsigned operand)is to set the corresponding operand to ready.

Since we need to set the latency of reading request to 2 cycles, we need to let this operand not ready, which means not to let rr be added to read_ops. I use the following way to do that.

- First, I change this function to the following.

```
for ( std::list<op_t>::iterator r=allocated.begin(); r!=allocated.end(); r++ ) {
                ...
                bool ok = m_arbiter.allocate_for_read(bank,rr);
                 if (ok)
                {
                        read_ops[bank] = rr;
                        m_arbiter . read_queue_pop ( bank ) ;
                }
        }
```

  where ok is true if and only if this register is finally ready after 2 cycles latency.

- Second, we need to changearbiter_t::allocate_for_read()to return a bool value. I do the following modifications.

```
bool allocate_for_read( unsigned bank, const op_t &op )
                {
                        assert( bank < m_num_banks );
                         if ( m_allocated_bank[bank].is_free() ) {
                                m_allocated_bank [bank ] . alloc_read (op, 2) ;
                     } else {
                                 if ( m_allocated_bank[bank].op_t_same_to(op) ) {
                                                if ( !m_allocated_bank[bank].is_finished() ) return false ;
                                } else {
                                        return false ;
                                }
                     }
                     return m_allocated_bank [bank ] . is_finished () ;
                }
```

  where allocation_t::is_finished() return true if and only if this is the last cycle left until the operand is ready. While, allocation_t::op_t_same_to() returns true is the current op is exactly the operand that occupies the bank.

- Other changes are required to make the whole things work.

```
void alloc_read( const op_t &op, int k )                                {
                assert ( is_free () ) ;
                m_allocation=READ_ALLOC;
```

```
        m_op=op ;
        step_to_go = k;
}
void reset () {
        step_to_go−−;
        if ( step_to_go==0 ) {
                m_allocation = NO_ALLOC;
        }
}
bool op_t_same_to( const op_t& obj )
{
```

Page 12

```
        int k=0;
        k+= m_op. m_valid==obj . m_valid ;
        k+= m_op.m_cu==obj.m_cu;
        k+= m_op.m_warp==obj.m_warp;
        k+= m_op . m_operand==obj . m_operand ;
        k+= m_op. m_register==obj . m_register ;
        k+= m_op.m_bank==obj.m_bank;
        return k==6;
}
bool is_finished() { return step_to_go==1; }
```

where variable step_to_go works as a counter, which will be decrease by one at each end of the cycle when reset() being invoked.

- Last but not the least, since the original one cycle model assume that every operand can be dispatched after this cycle, in opndcoll_rfu_t::arbiter_t::allocate_reads() , we need to delete the code

```
m_queue[ bank ] . pop_front ( ) ;
```

in the code segment

```
m_last_cu = _pri;
for ( unsigned i=0; i < m_num_banks; i++ ) {
        if ( _inmatch[i] != −1 ) {
                if ( ! m_allocated_bank [ i ] . is_write () ) {
                        unsigned bank = ( unsigned )i;
                        op_t &op = m_queue[bank]. front () ;
                        result . push_back ( op ) ;
                        m_queue[ bank ] . pop_front ( ) ;
                }
        }
}
```

to keep the operand reading request still alive in the next cycle.

(2) Modify RF Writing Requests For the writing part, the major job is to modify a function named opndcoll_rfu_t::writeback(). I change the code segment

```
bool opndcoll_rfu_t::writeback( const warp_inst_t &inst )
{
        ...
        for ( r=regs.begin(); r!=regs.end();r++,n++ ) {
                unsigned reg = *r;
                unsigned bank = register_bank(reg , inst .warp_id() ,m_num_banks,m_bank_warp_shift) ;
                 if ( m_arbiter.bank_idle(bank) ) {
                            m_arbiter . allocate_bank_for_write (bank , op_t(&inst , reg ,m_num_banks, m_bank_warp_shift ) ) ;
                } else {
                            return false ;
                }
        }
        ...
}
```

to the following

```
bool opndcoll_rfu_t::writeback( const warp_inst_t &inst )
{
        ...
        for ( r=regs.begin(); r!=regs.end();r++,n++ ) {
                unsigned reg = *r;
                unsigned bank = register_bank(reg , inst .warp_id() ,m_num_banks,m_bank_warp_shift) ;
                 if ( m_arbiter.bank_idle(bank) ) {
                            bool ok = m_arbiter.allocate_bank_for_write(bank,op_t(&inst ,reg,m_num_banks,
                                        m_bank_warp_shift ) ) ;
                             if (!ok) return false ;
                } else if ( m_arbiter.is_writing(bank, op_t(&inst,reg,m_num_banks,m_bank_warp_shift)) ) {
                             if ( !m_arbiter.is_finished(bank) )                                   return false ;
                } else {
                            return false ;
                }
        }
        ...
}
```

The reason to do this modification is similar to that of the reading part. When a operand is writing back but not finishes yet, We just make this process stuck during this cycle and let the operand keep occupied the corresponding bank until the writing process is finally finished.

Other related codes to be modified are the following.

```
bool arbiter_t::is_writing( unsigned bank, const op_t& obj)
{
                if ( m_allocated_bank[bank].is_write() ) {
                            if ( m_allocated_bank[bank].op_t_same_to(obj) )                              return true ;
                }
                return false ;
}
bool arbiter_t::is_finished( unsigned bank)
```

```
{
            return m_allocated_bank [bank ] . is_finished () ;
}
bool arbiter_t::allocate_bank_for_write( unsigned bank, const op_t &op )
{
            assert( bank < m_num_banks );
            m_allocated_bank [bank ] . alloc_write (op, 2) ;
            return m_allocated_bank [bank ] . is_finished () ;
}
void opndcoll_rfu_t::allocation_t::alloc_write( const op_t &op, int k ) {
            assert ( is_free () ) ;
            m_allocation=WRITE_ALLOC;
            m_op=op ;
            step_to_go = k;
}
```

4.2 Modify Different Banks with Different Latencies

From the whole bunch of codes, I finally figure out a function namedopndcoll_rfu_t::init() in which the informations about register banks are initialized. Following is the original function.

```
void opndcoll_rfu_t::init( unsigned num_banks, shader_core_ctx *shader )
{
```

Page 14

```
    m_shader=shader ;
    m_arbiter . init (m_cu. size () ,num_banks) ;
    //for( unsigned n=0; n<m_num_ports;n++ )
    //          m_dispatch_units[m_output[n]].init( m_num_collector_units[n] );
    m_num_banks = num_banks;
    m_bank_warp_shift = 0;
    m_warp_size = shader->get_config ()->warp_size ;
    m_bank_warp_shift = ( unsigned )( int ) (log(m_warp_size+0.5) / log(2.0));
    assert( (m_bank_warp_shift == 5) || (m_warp_size != 32) );

    for ( unsigned j=0; j<m_cu.size(); j++) {
          m_cu[ j ]->init( j , num_banks , m_bank_warp_shift , shader->get_config () , this ) ;
    }
    m_initialized=true ;
}
```

where m_arbiter.init(m_cu.size(),num_banks); is called to do the initialization.

Thus, I change this function a little bit to the following.

```
void opndcoll_rfu_t::init( unsigned num_banks, shader_core_ctx *shader )
{
    m_shader=shader ;
    std::map<int , unsigned> latencies;
    FILE *fp = fopen("latency.setup","r");
    int bank=0, tmp;
    while ( fscanf ( fp , "%d" ,&tmp) ==1)
    {
        latencies[bank]=tmp;
        ++bank ;
        printf("Bank %d: latency %d\n", bank, tmp);
    }
    if ( bank==0) tmp=READ_OPERAND_CYCLE;
    for ( unsigned i=bank;i<num_banks;++i) {
        latencies [ i ]=tmp;
        printf("Bank %d: latency %d\n", i , tmp);
      }

    m_arbiter . init (m_cu. size () ,num_banks, latencies ) ;

    //for( unsigned n=0; n<m_num_ports;n++ )
    //          m_dispatch_units[m_output[n]].init( m_num_collector_units[n] );
   m_num_banks = num_banks;
   m_bank_warp_shift = 0;
   m_warp_size = shader−>get_config ()−>warp_size ;
   m_bank_warp_shift = ( unsigned )( int ) (log(m_warp_size+0.5) / log(2.0));
   assert( (m_bank_warp_shift == 5) || (m_warp_size != 32) );

    for ( unsigned j=0; j<m_cu.size(); j++) {
          m_cu[ j ]−> init ( j , num_banks , m_bank_warp_shift , shader−>get_config () , this ) ;
    }
    m_initialized=true ;
}
```

In this function, I read the *bank read delay cycle* configurations out from a file named

latency.setup

And I take the following method to read in the configurations. If all the content in the file are k numbers: x_1, x_2, $\cdots$, x_k, then I set the first k banks to the corresponding latencies given by

Page 15

these k numbers. If there are more banks to allocate, I will set them all to x_k.

To fulfill all the job, I add a variable to the class opndcoll_rfu_t::arbiter_t.

```
std::map<int , unsigned> reg_latencies;
```

Each time I allocate a read or write operation from an instruction, I will set the counter step_to_go to the corresponding latency of this bank. For example, in the reading phase, I change the following details.

```
bool allocate_for_read( unsigned bank, const op_t &op )
        {
            ...
            if ( m_allocated_bank[bank].is_free() ) {
                m_allocated_bank [bank ] . alloc_read (op, reg_latencies [bank ] ) ;
           } else {
                ...
           }
        ...
    }
```

which completes the modification.

4.3 Modify Different Registers with Different Latencies

It’s quite easy to change the codes to cater for the need of different register latencies for an operand reading or writing process. One can just change the following code

```
bool allocate_for_read( unsigned bank, const op_t &op )
    {
        ...
        if ( m_allocated_bank[bank].is_free() ) {
            m_allocated_bank [bank ] . alloc_read (op, reg_latencies [bank ] ) ;
        } else {
            ...
        }
        ...
    }
```

to

```
bool allocate_for_read( unsigned bank, const op_t &op )
    {
        ...
        if ( m_allocated_bank[bank].is_free() ) {
            m_allocated_bank [bank ] . alloc_read (op, reg_latencies [op. m_register ] ) ;
        } else {
            ...
        }
        ...
    }
```

since the variable m_register in the class op_t save the index of the register.

The same changes should be done to the function allocate_for_write() too.

Page 16

```
bool arbiter_t::allocate_bank_for_write( unsigned bank, const op_t &op )
{
        assert( bank < m_num_banks );
        m_allocated_bank [bank ] . alloc_write (op, op. m_register ) ;
        return m_allocated_bank [bank ] . is_finished () ;
}
```

Following are the statistical results I get using different configurations.

4.4 Results after Modifications

Here is the results of setting different read and write latencies.

Benchmark	**One Cycle**	**Two Cycles**	**Five Cycles**
CP	241538	243006	457200
NQU	35987	42329	66953
RAY	140784	154662	258997

The detailed results can be obtained from the attached logs as follows.

Benchmark	**One Cycle**	**Two Cycles**	**Five Cycles**
CP	CP/cp1.log	CP/cp2.log	CP/cp5.log
NQU	NQU/nqu1.log	NQU/nqu2.log	NQU/nqu5.log
RAY	RAy/ray1.log	RAY/ray2.log	RAY/ray5.log

Following is the result after changing different bank delays to different latencies, here I set x_i to a random number in [1,5] for all $i = 1,2,\cdots,100$ and get the following result.

Benchmark	**Original Results**	**Modified Results**
CP	241538	284857
NQU	35987	47397
RAY	140784	189287

More detailed log files can be attached at the following location.

Benchmark	**Modified Results**
CP	CP/cp_all.log
NQU	NQU/nqu_all.log
RAY	RAY/ray_all.log

Here comes to the end of my GPGPU-Sim experiment.

5 Acknowledgments

Thanks for the help from my teacher and TA. This project benefits me a lot, since this project

let me know more about the GPU architecture and how to understand other people’s codes as well.

6 REFERENCES

1 The GPGPU-Sim canonical manual.

http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual