IS2202 – Laboration 2

Simulating Graphics Processing Units (GPUs) in GPGPU-Sim

Artur Podobas

Introduction

The laboration is designed as a walkthrough with questions that the student (read: you) needs to answer. Each exercise focuses on a particular subject, and once finished, you are expected to write a laboration rapport that you hand in on Canvas. The goal is for you to learn how simulators can be used to perform exploration on Graphics Processing Units (GPUs). For this lab, we will use the GPGPU-Sim[1] simulator.

- A tip throughout the lab is to always save outputs from different simulations in between runs, in order to analyze the results further,
- It is a good idea to Copy instructions from the manual and paste them in the terminal window (Right-click Paste),
- A Virtual Machine is run as a simulation on your physical machine. The GPGPU-sim is running a simulation emulating GPU PTX instructions, so in practice, we have a simulation running a simulation, which is very slow, so running Linux on your machine is recommended. If you run the GPGPU-Sim natively instead of on a VM, things will speed up significantly.

Grading

The lab consists of a number of directed parts and a number of open-ended parts. The directed part is compulsory for a passing grade **E**. Finishing the directed part and one open-ended part completely can give you a grade **C**. Finishing the directed part and both open-ended parts completely can give you a grade **A**.

1 "Analyzing CUDA workloads using a detailed GPU simulator", A Bakhoda et al., ISPASS'09

Preparation: Setting up the GPGPU-Sim System

Note: *We recommend you setup GPGPU-Sim on your own machine running Linux (we used Ubuntu 22.04 LTS). If you have no other option but to use the pre-supplied Virtual Image, then you can skip this part. Simulation estimates in this lab manual are natively (not using VM).*

Step 1: Install dependencies

Start by installing dependencies required by GPGPU-Sim. Execute the following commands:

```
sudo apt-get install build-essential xutils-dev bison zlib1g-dev flex libglu1-mesa-dev
sudo apt-get install doxygen graphviz
sudo apt-get install python-pmw python-ply python-numpy libpng12-dev python-matplotlib
sudo apt-get install libxi-dev libxmu-dev freeglut3-dev
```

Step 2: Install GCC-7

GPGPU-Sim relies on being compiled with a rather old version of GCC, and while there are reports of successfully using newer GCCs, we had the best success with GCC-7.

To install GCC-7, do the following (use your favorite editor, in our case, emacs):

```
sudo emacs /etc/apt/sources.list
```

Add the line:

```
deb [arch=amd64] http://archive.ubuntu.com/ubuntu focal main universe
```

and install GCC-7:

```
sudo apt-get install g++-7
```

Next, we need to set GCC-7 as the default gcc/g++ compiler in the system. Execute the following:

```
sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-7 200
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-7 200
```

Step 3: Download and Install CUDA Toolkit 10.0

Go to ”https://developer.nvidia.com/cuda-10.0-download-archive” and download the CUDA Toolkit 10.0 for Linux x86_64 Ubuntu 18.04. Choose runfile (local).

Once downloaded, install it by executing the following command in the terminal:

```
sudo sh cuda_10.0.130_410.48_linux.run
```

After the above has been executed, open .bashrc in your favorite editor (e.g., emacs ~/.bashrc) and add the following lines:

```
export CUDA_INSTALL_PATH=/usr/local/cuda-10.0
export PATH=$CUDA_INSTALL_PATH/bin:$PATH
```

Restart your terminal now.

Step 4: Installing GPGPU-Sim

Start by cloning the GPGPU-Sim repository by writing:

```
git clone https://github.com/gpgpu-sim/gpgpu-sim_distribution.git
cd gpgpu-sim_distribution
```

Next, set up the GPGPU-sim environment:

```
source setup_environment
```

Next, there is one C file that can cause trouble due to ill-formatted strings. The source file is called ptx_parser_decode.def, and it needs to be modified by inserting'\' before the two internal strings on lines 2 and 4. Do this.

Finally, start building GPGPU-sim:

```
make
```

Step 5: Download applications

Finally, download the applications (called ”CUDA Apps” on the Canvas Laboration page) and un-tar them into your system.

You are now ready for the directed part of the lab.

Exercise 1: Compiling CUDA applications (Directed part)

Our first exercise will be to learn how to compile CUDA applications in a way that allows us to simulate them using GPGPU-Sim. We will focus on compiling the GEMM kernel (Matrix Multiplication), which we will study and simulate in forthcoming exercises.

Start by navigating to where your GEMM source code is. In the virtual machine, this is in **/home/IS2202/LAB2/apps/gemm/**:

```
cd /home/IS2202/LAB2/apps/gemm/
ls
```

In this directory, you will find a single file called ”gemm.cu”, which contains both the host and GPU-kernel code. Analyze this code.

Question 1: What is the CUDA kernel called, and what is its function signature? What is the datatype that we used in the matrix multiplication?

Question 2: Analyze the GEMM implementation and derive a formula for how many Floating-Point Operations (FLOP) are needed to compute a N x N matrix multiplication (**hint**: look at the simpler naive CPU-version **called matrix_multiply_cpu**).

Next, compile the GEMM CUDA version by invoking the NVIDIA compiler:

```
nvcc -O3 -ccbin g++ --cudart shared gemm.cu -o gemm.gpu
```

By giving the compiler the ”--cudart shared” option, we tell the compiler not to statically link the CUDA runtime library into the binary and instead load it during runtime. This, in turn, allows simulators (such as GPGPU-sim) to provide their own CUDA runtime library that – instead of interfacing with a GPU device – simulates the binary code.

Exercise 2: Simulating Matrix Multiplication on the GTX-480 (Directed part)

Our second exercise will be to simulate a GEMM kernel (Matrix Multiplication) on a GTX-480 GPU. Start by going to the GPGPU-sim folder (/home/IS2202/LAB2/gpgpu-sim_distribution/) and set up the environment.

```
cd /home/IS2202/LAB2/gpgpu-sim_distribution/
source setup_environment
```

This will initialize the GPGPU-sim environment and (amongst others) tell the system where to find the cuda runtime library, which the GPGPU-sim overloads in order to intercept calls to the GPU.

The next step is to create a directory for our simulation experiments, such as for example, **sim-gemm** and enter it:

```
mkdir sim-gemm
cd sim-gemm
```

Next, we need to copy the configuration files for the type of GPU that we would like to simulate. In our particular case, we will start by simulating the NVIDIA GTX-480 GPU and thus proceed to copy the configuration for that GPU:

```
cp ../configs/tested-cfgs/SM2_GTX480/* .
```

Question 3: Look up the NVIDIA GTX480 GPU and describe its characteristics. What year was it introduced? What was its peak performance and external memory bandwidth? What GPU architecture family did it belong to? What is the GPU core frequency?

Next, copy the *gemm.gpu* binary that was created in exercise 1 into the current directory:

```
cp /home/IS2202/LAB2/apps/gemm/gemm.gpu .
```

You are now ready to simulate the matrix multiplication kernel by executing it from the command line. To start out, we will compute a fairly small matrix multiplication of size 32x32 and pipe the output to the file ”stat.txt”:

```
./gemm.gpu 32 > stat.txt
```

Wait until the application finishes simulating. When the application finishes, the GPGPU simulator will give you a short report and will also populate the local directory with files containing statistics from the simulation.

Question 4: How long time did the simulation take? What was the estimate slowdown of the simulator (”…”). Why is the slowdown so large compared to executing on a real GPU?

Next, open the file **stat.txt**. This file contains all the statistics associated with the simulation. Analyze this file thoroughly (to understand the statistics reported, consult **http://gpgpu-sim.org/manual/index.php/Main_Page#Understanding_Simulation_Output**).

Question 5: Analyze **stat.txt** and answer the following questions:

a) How many clock cycles did it take to execute the GEMM kernel?
b) How many instructions were executed?
c) How high IPC (instructions-per-cycle) did the GPU achieve?
d) DRAM Bandwidth obtained and what L2 cache hit rate was achieved?
e) What performance in terms of #FLOPS/s did we achieve
hint: use information from exercise 1 to calculate the FLOP/s.

Next, we are going to use the same GEMM kernel but change the size of the matrices that are being multiplied. Execute the kernel but change N to 64, 128, 256, and 512. For each run, backup **stat.txt** for analysis.

Question 6: Draw and discuss the following graphs:

a) Execution time (in seconds) that the kernel took
(**hint:** calculated as *gpu_sim_cycles/core_frequency[Hz]*)
b) FLOP/s as a function of problem size (N)
c) IPC as a function of problem size (N)
d) Simulation time as a function of problem size (N)
e) External bandwidth as a function of problem size (N)
f) GPU occupancy as a function of problem size (N)

Question 7: Repeat this exercise but with the NVIDIA RTX2060 GPU. Contrast your observation with the RTX2060 GPU against the prior observations with the GTX480 GPU, and discuss and explain these differences and observations.

Exercise 3: Power Simulation (Directed part)

GPGPU-Sim includes the McPAT[2] power simulator, which is coupled to the GPU simulator and allows for a rough estimation of the power consumption of a particular GPU running a particular workload.

To enable McPAT power profiling, open the file gpgpusim.config and add the following options (unless they are already there):

```
-power_simulation_enabled 1
-gpuwattch_xml_file gpuwattch_gtx480.xml
```

Note that if you use a different GPU model (other than GTX480) you will have to change the XML file. Save this file.

In this exercise, we will use three other benchmarks. Some of these benchmarks should be familiar to you from LAB1 and come from the Rodinia benchmark suite[3] ,and have been pre-compiled by us. You can find the binaries in the following folders (on the VirtualBox): /home/IS2202/LAB2/apps/.
The benchmarks are and how to execute them are:

1) **HotSpot3D:** is a PDE solver for simulating hotspots (temperature gradients) on 3D stackable silicon chips. Its a type of structured grid application and its used in physics.
   Run command (sim-time: ~14 min): *./hotspot3d.gpu 512 8 4 ~/LAB2/apps//data/power_512x8 ~/LAB2/apps/data/temp_512x8 output.out*

2) **BFS:** Is an graphs application implementing breadth-first search:
   Run command (sim-time: ~3 min): *./bfs.gpu ~/LAB2/apps/data/graph65536.txt*

3) **LUD:** LU Decomposition is a dense linear algebra application for factorizing a matrix into two matrixes (lower and upper triangular matrixes).
   Run command (sim-time: ~12 min): *./lud.gpu -s 512* -v

Your task is now to investigate the power consumption of three different applications. Use the GTX480 GPU model and execute each of the three applications. After each simulation, you will now find a file in your folder called **gpgpusim_power_report_DATE.log**. This file contains information about the minimum(**gpu_tot_min_power**), average (**gpu_tot_avg_power**), and maximum (**gpu_tot_max_power**) power consumption experience by the device simulated.

Question 8: Draw and discuss the following graphs. Use the GTX480 GPU model:

a) Min, max, and average power consumption for the three applications,
b) Execution time (in seconds) that the three applications took (ignoring host overhead),
   Hint: Some applications might execute multiple kernels several times. Use the final *gpu_tot_sim_cycle* (instead of gpu_sim_cycle) metric for those.
c) Energy consumption (Joules) of the three applications.
   Hint: Joules is the same as Watt-seconds (**W * s**).

2 "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures", S Li et al., MICRO'09
3 www.cs.virginia.edu/rodinia/doku.php

A well-known technique for controlling performance and power consumption is called Dynamic Frequency Scaling (DFS), where a computing system will change its clock frequency in order to compute faster or to preserve energy. The clock frequency is an important variable that influences power consumption ($P_{dyn}(W) \simeq C * f_{clk}[MHz] * V^2$). Your task is now to perform this frequency scaling manually and investigate its result.

The GPU has four different clock domains: the core clock, the interconnect clock, the L2 cache clock, and the DRAM clock. For the GTX480, the core/interconnect/L2-clock are all set to 700 Mhz. These can be changed by locating the option ”**--gpgpu_clock_domains**” in the **gpgpusim.config** file. If we change the frequency of a domain in the GPU, we have to make a similar change in the power-configuration file (**gpuwattch_gtx480.xml** for the GTX480): target_core_clockrate and clock_rate (for system.core0 and system.NOC0).

Question 9: Use the GTX480 model and change the ***core frequency-, interconnect-,L2 clocks*** to 300 MHz and 1GHz respectively. Keep the DRAM clock unmodified. Redo question 8 but with the new clock frequencies. Discuss the result. Are the applications compute- or memory-bound?

Exercise 4: Examining an Application of your choice! (Open Ended Part)

Objective: The goal of this exercise is for you to apply the experiences learned in this lab to examine a new application and to quantify its performance empirically with four different GPU models. Your task is as follows:

1. Find a non-trivial CUDA application and compile it for the GPGPU-simulator. By non-trivial, we mean an application that performs a meaningful computation and is composed of at least two different GPU kernels.
   Hint: Consider using applications that come with the Rodinia or the CUDA Toolkit.
2. You are now tasked with proposing to buy a new GPU that will exclusively run your application for a large fraction of its lifetime. Use GPGPU-Sim to investigate at least four different GPU models in order to help with the decision-making.
3. Propose which GPU one should buy for the application and use the data obtained from your simulation to argue for your cause. It is not enough to simply answer by proposing the most recent/newest GPU.

Resources:
Rodinia (www.cs.virginia.edu/rodinia/doku.php)
CUDA Toolkit (https://github.com/NVIDIA/cuda-samples).

Exercise 5: Architectural Exploration (Open Ended Part)

Objective: The goal of this exercise is for you to select a new application, and to modify the GPU architecture in order to execute the new application a*s energy-efficient as possible (that is, as few Joules as possible)*. Follow the following steps:

1) Find a non-trivial CUDA application and compile it for the GPGPU-simulator. By non-trivial, we mean an application that performs some meaningful computation and is composed of at least two different GPU kernels. Use a reasonable input size.
   Hint: Consider using applications that comes with the Rodinia or the CUDA Toolkit.
   Note: If you completed Exercise 4, then you can use the same application here.

2) Use the NVIDIA GTX480 model as a baseline and evaluate the performance and energy-consumption.

3) Next, perform design-space exploration using the GPGPU-Sim in order to ”create” an architecture that performs as energy efficient as possible for your application. Use the GTX480 model as base. Include your experiments and the data in the report and motivate why you believe a certain configuration is better than another.
   Consider investigating the following options:
   a) Different core frequencies (MHz)
   b) Different number of cores (SMs)
   c) Number of Warps per SM
   Note: Changing attributes such as ”core_tech_node” or ”temperature” in the XML file to reduce power-consumption is not allowed. You should focus on modifying architectural decisions.

Resources:
GPGPU-Sim Manual: http://gpgpu-sim.org/manual/index.php/Main_Page
GPUWattch Energy Model Manual: http://gpgpu-sim.org/gpuwattch/
Rodinia (www.cs.virginia.edu/rodinia/doku.php)
CUDA Toolkit (https://github.com/NVIDIA/cuda-samples).