CP3106 Project Report

**Development of a Database Link Between Mainframe and PC**

By

Chua Meng Lee

Department of Computer Science School of Computing

National University of Singapore AY2017/2018, Semester II

CP3106 Project Report

**Development of a Database LINK Between Mainframe and PC**

By

Chua X X

Department of Computer Science School of Computing

National University of Singapore AY2017/2018, Semester II

Advisor: Assoc Prof XXX

Deliverables:

Report: 1 Volume

Manual: 1 Volume

Software: 1 USB

Abstract

Increasing data size and applications has generated huge demand for computation capability, and parallel computing is a potential answer. Plenty of algorithms about parallel computing have been developed, and performance simulators are created accordingly. GPGPU-Sim is one of those simulators which can characterize applications written in NVIDIA’s CUDA programming model by running NVIDIA’s parallel thread execution (PTX) virtual instruction set. But GPGPU-Sim executes all the instructions in CPU-only sequential, which means it will take more time to run the same program compared with the time the GPU version takes. Meanwhile, NVIDIA provides a profiling tool called nvprof which can characterize CUDA programs and it runs programs in GPU. In our work, we tried to overcome the gap between nvprof and GPGPU-Sim. We have modified the GPGPU-Sim to let it generate performance metrics based on statistics that are given manually. We have also designed the transformation from nvprof’s metrics to GPGPU-Sim input. The final output of our modified GPGPU-Sim will be compared with that of original GPGPU-Sim.

Here I have not changed.

Subject Descriptors: https://cran.r-project.org/web/classifications/ACM.html

* + 1. Network Architecture and Design
    2. Network Protocols

C.2.4 Distributed Systems

C.4 Performance of Systems

I.2.9 Keyword

Wireless communication, routing protocols, distributed applications, fault tolerance, sensors

Implementation Software and Hardware:

Ubuntu 16.04.1 Linux garuda 4.13.0

g++ 5.4.0 cuda9.1

TITAN V Graphics Card

## Acknowledgement

Following the abstract page, students may want to acknowledge the contributions or assistance of others to the project. It should be kept in one A4 page.

## Table of Contents

In addition to the heading of each section, sub-heading can also be used but its depth should be kept to a minimum. Details of appendices should also be given here. Students may use more than one A4 page for the content page.

1

* 1. 5.2 Limitations

5.3 Recommendations for Further Work References

Appendix A – Program Listing

Appendix B – How to Use the Program

Introduction

* 1. Motivation
  2. GPGPU-Sim
  3. GPUWattch

Background

* 1. GPUWattch√
  2. GTX480
  3. TITAN V
  4. Benchmarks

Implementation

* 1. Extraction of parameters
  2. Transformation
  3. Changes to GPGPU-Sim calculation process

Results

* 1. Analysis
  2. Comparison with original GPGPU-Sim

Conclusions

5.1 Summary

Title Abstract

Acknowledgement List of Figures

**Table of Contents**

# Introduction

## Motivation

There are plenty of ways to analyze a CUDA program’s performance and GPGPU-Sim is a typical simulator. However, one of its apparent disadvantages is that it takes much more time to finish the whole process than the original CUDA program. We observe that the performance counters used in GPGPU-Sim are possible to achieve from other methods(like NVIDIA profiling tool), so it is possible to generate the equivalent power estimation of GPGPU-Sim by passing these performance counters directly to it and generate the result instead of simulating the whole process in GPGPU-Sim. This means we can get the equivalent estimation of power consumed in the format of GPGPU-Sim’s output at a much faster speed. Faster performance analysis can provide insights in developing better algorithms and architecture, even if it is an estimation instead of real data. As long as the fault is tolerable, the estimation will make sense. It is a trade-off between accuracy and speed. So, our goal is to find a method to estimate the energy metrics of a CUDA program through GPGPU-Sim quickly.

## GPGPU-Sim

GPGPU-Sim was created by Tor Aamodt's research group at the University of British Columbia.[[1]]

It is a cycle-level simulator modeling contemporary graphics processing units (GPUs) running GPU computing workloads written in CUDA or OpenCL. Also included in GPGPU-Sim is a performance visualization tool called AerialVision and a configurable and extensible energy model called GPUWattch.

GPGPU-Sim and GPUWattch have been rigorously validated with performance and power measurements of real hardware GPUs. More details will be introduced at the part of Background.

## GPUWattch

GPUWattch was collaboratively developed by researchers at UTAustin, UWisconsin, and UBC.[[2]] It is an energy model based upon McPAT that is integrated with GPGPU-Sim. McPAT is an architectural modeling tool for chip multiprocessors (CMP) The main focus of McPAT is accurate power and area modeling[[3]], and a target clock rate is used as a design constraint. McPAT performs automatic extensive search to find optimal designs that satisfy the target clock frequency

# Background

## GPGPU-Sim

Because GPGPU-Sim is so important to our project, so more details need to be included. GPGPU-Sim is a cycle-level GPU performance simulator that focuses on “GPU computing” (general purpose computation on GPUs). Its microarchitecture model only reports the cycles where the GPU is busy, which means it does not model either CPU timing or PCI Express timing(i.e memory transfer between GPU and CPU). So its performance measurements are for the GPU code only. GPGPU-Sim models GPU microarchitectures similar to those in the NVIDIA GeForce 8x, 9x, and Fermi series. Its accuracy has passes careful validation. As the author says, when they use the native hardware instruction set(PTXPlus), GPGPU-Sim 3.1.0 obtains IPC correlation of 98.3% and 97.3% respectively on a benchmark suite

### Top-Level Organization

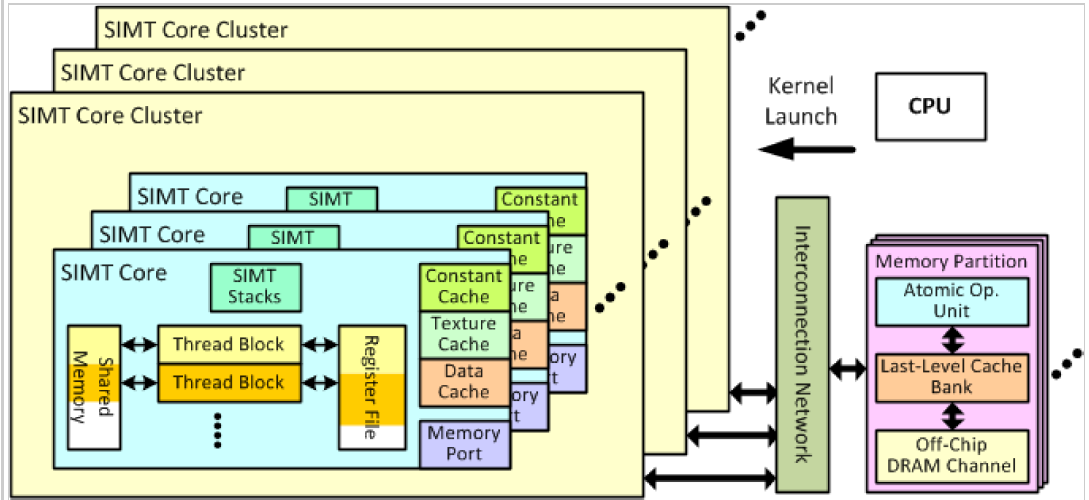
The GPU modelled by GPGPU-Sim is composed of Single Instruction Multiple Thread(SIMT) cores. Here a SIMT core is roughly equivalent to what NVIDIA calls an Streaming Multiprocessor(SM), and it models a highly multithreaded pipelined SIMD processor. A Stream Processor(SP) would correspond to a lane within an ALU pipeline in the SIMT core.

#### SIMT Core Clusters

Multiple SIMT cores make up one SIMT Core Cluster. The SIMT cores in one cluster share a common port to the interconnection network.

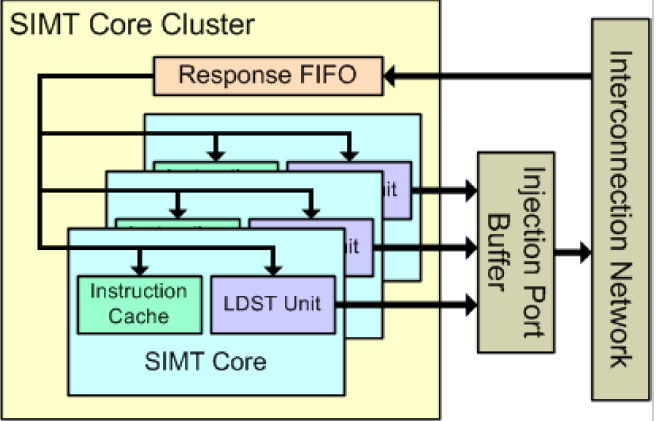
The figure below describes the overall situation, all the SIMT cores are grouped into clusters, and the clusters are connected to the interconnection network to memroy partitions that interface to graphics GDDR DRAM.

Figure :Overall GPU Architecture



The below figure describes the process that how a cluster deals with the packets ejected from the interconnection network. If the packet is a memory response servicing an instruction fetch miss, it will be directed to a SIMT Core’s instruction cache. Otherwise, it will be directed to the LDST Unit(the memory pipeline). Also, it describes the details of one SIMT Core Cluster. One SIMT Core Cluster contains multiple SIMT Core and can maintain the overall management of the cluster.

Figure :SIMT Core Clusters



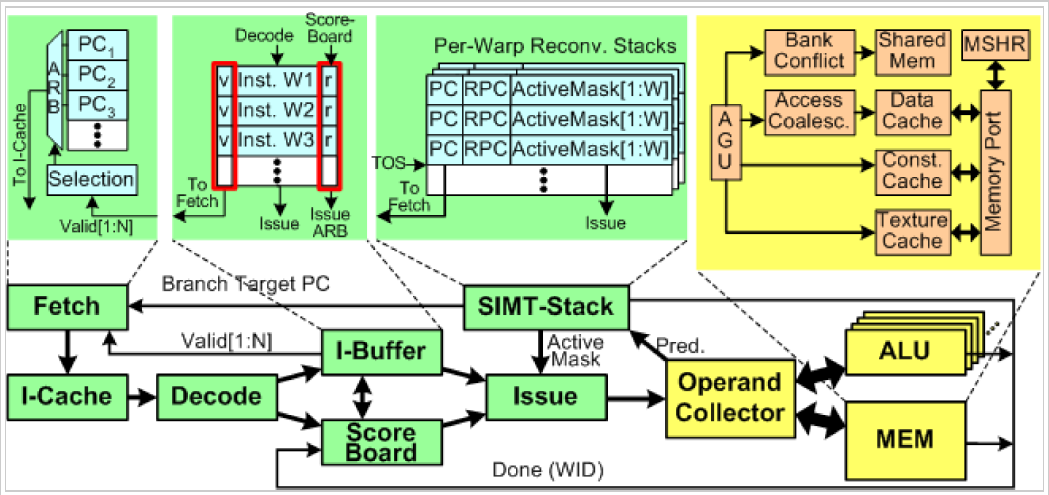
#### SIMT Cores

A SIMT core models a highly multithreaded pipelined SIMD processor, which is roughly equivalent to what NVIDIA calls a Streaming Multiprocessor(SM).

In one SM the warps are executed using pipeline technique and interleaving technique as well. The lifecycle of one instruction consists of six stages: fetch, decode, issue, read operands, execute and writeback.

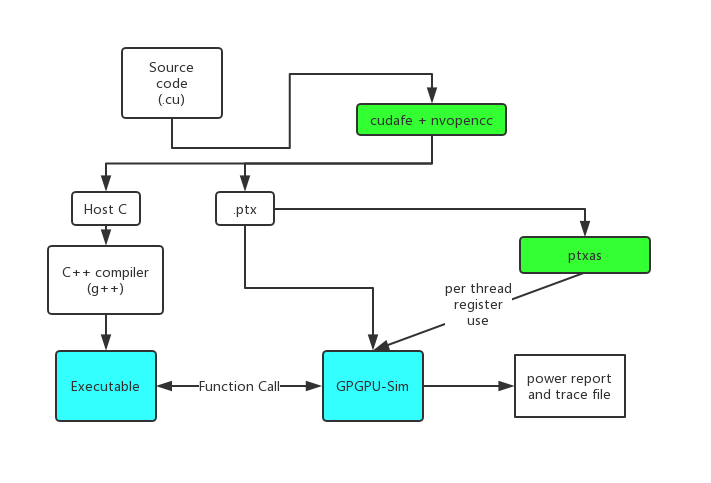
The figure below shows the detailed microarchitecture Model of SIMT Core. For more details, can refer to GPGPU-Sim manual.

Figure :Detailed Microarchitecture Model of SIMT Core



To support CUDA programs, GPGPU-Sim contains its own CUDA Parallel Thread Execution (PTX) instruction set. The picture below shows the compilation flow for GPGPU-Sim from a CUDA application.

It still uses *cudafe* to transform the source code of a CUDA application into host C code and device C code. Then, the device C code (running on the GPU) will be compiled into PTX assembly by *nvopencc* and GPGPU-Sim will parse these PTX assembly code to get code for the compute kernels. Also, GPGPU-Sim optimizes register usage to avoid spilling. Before simulation, some other work is done by GPGPU-Sim to increase the reality of the power model, such as determining the register usage per thread. During each simulation, the PTX functional simulator executes all the instructions in the order of time. After the simulation, the host C code is allowed to resume and continue to execute the remaining part of code. It does not model the CPU timing and energy because the host code runs on a normal CPU.



### GPUWattch

GPUWattch is an energy model based upon McPAT. Compared with previous models, it offers **Flexibility**(the use of a bottom-up methodology and parameters abstracted from the microarchitectural components), the **Adaptability**(both program and microarchitectural level interactions are captured and can adapt to new power-management techniques specifically targeted at GPUs) and the **Stability**(it passes carefully designed experiments on commercial GPUs).

Below the table describes the components modeled in the power model

|  |  |  |
| --- | --- | --- |
| Components modeled in GPUWattch | | |
| Group | Description | Microarchitectural |
| SM | SM pipeline | Pipeline |
| Caches | Instruction Cache |
| L1 Data Cache |
| Texture Cache |
| Constant Cache |
| Memory Coalesce Logic |
| Shared Memory | Shared Memory Banks |
| Shared Memory Crossbar Network |
| Register Files | Register File Banks |
| Operands Collectors |
| Operand Collection Crossbar Network |
| Execution Units | Integer ALU |
| Floating Point Unit(FPU) |
| Special Function Unit(SFU) |
| Memory | Memory Controller | On-chip Memory Controller |
| Interconnection Network | Interconnect | Network On Chip(NOC) |
| DRAM | Main Memory | GDDR5/GDDR3 |

Energy consumed by the above components will be estimated by these counters

|  |  |  |
| --- | --- | --- |
| List of performance counters | | |
| Index | Performance Counters | Description |
| 1 | IC\_H | Counts the number of instruction cache hits |
| 2 | IC\_M | Counts the number of instruction cache misses |
| 3 | DC\_RH | Counts the number of data cache read hits |
| 4 | DC\_RM | Counts the number of data cache read misses |
| 5 | DC\_WH | Counts the number of data cache write hits |
| 6 | DC\_WM | Counts the number of data cache write misses |
| 7 | TC\_H | Counts the number of texture cache hits |
| 8 | TC\_M | Counts the number of texture cache misses |
| 9 | CC\_H | Counts the number of constant cache hits |
| 10 | CC\_M | Counts the number of constant cache misses |
| 11 | SHRD\_ACC | Counts the number of shared memory accesses |
| 12 | REG\_R | Counts the number of register file reads in all instructions |
| 13 | REG\_W | Counts the number of register file writes in all instructions |
| 14 | NON\_REG\_Ops | Counts the number of non register file operands |
| 15 | SFU\_ACC | Counts the all instructions that exercise SFU pipeline (it also includes multiplications/division) |
| 16 | SP\_ACC | Counts the all instructions that exercise SP pipeline with integer operands |
| 17 | FPU\_ACC | Counts the all instructions that exercise SFU pipeline with floating-point operands |
| 18 | TOT\_INST | Counts the all decoded instructions |
| 19 | FP\_INT? | Counts the all instructions without operands |
| 20 | DRAM\_RD | Counts the dram read accesses |
| 21 | DRAM\_WR | Counts the dram writes accesses |
| 22 | DRAM\_PRE | Counts the dram precharge accesses |
| 23 | L2\_RH | Counts the number of L2 data cache read hits |
| 24 | L2\_RM | Counts the number of L2 data cache read misses |
| 25 | L2\_WH | Counts the number of L2 data cache write hits |
| 26 | L2\_WM | Counts the number of L2 data cache write misses |
| 27 | PIPE | Ratio of committed number of instructions to the maximum peak of committed instructions |
| 28 | NOC\_A(SIMT to Mem) | Counts the number of flits traveling from SIMT cluster to memory partition |
| 29 | NOC\_A(Mem to SIMT) | Counts the number of flits traveling from memory partition to SIMT cluster |
| 30 | IDLE\_CORE\_N | Counts the average number of idle cores over cycles of each sample |
| 31 | TOT\_FP\_INST | Counts the total number of floating points instructions |
| 32 | TOT\_INT\_INST | Counts the total number of integer instructions |
| 33 | L1D\_R\_ACC | Counts the all L1 data cache read instructions |
| 34 | L1D\_W\_ACC | Counts the all L1 data cache write instructions |
| 35 | L1\_RH | Counts the number of L1 data cache read hits |
| 36 | L1\_RM | Counts the number of L1 data cache read misses |
| 37 | L1\_WH | Counts the number of L1 data cache write hits |
| 38 | L1\_WM | Counts the number of L1 data cache write misses |

Its output’s format is like this:

*Kernel Average Power Data:*

*gpu\_avg\_power = 41.6007*

*gpu\_avg\_IBP, = 0.169145*

*gpu\_avg\_ICP, = 0.114284*

*gpu\_avg\_DCP, = 0.0175825*

*gpu\_avg\_TCP, = 0*

*...*

*gpu\_avg\_IC\_H, = 171.167*

*gpu\_avg\_IC\_M, = 32*

*gpu\_avg\_DC\_RH, = 3.25*

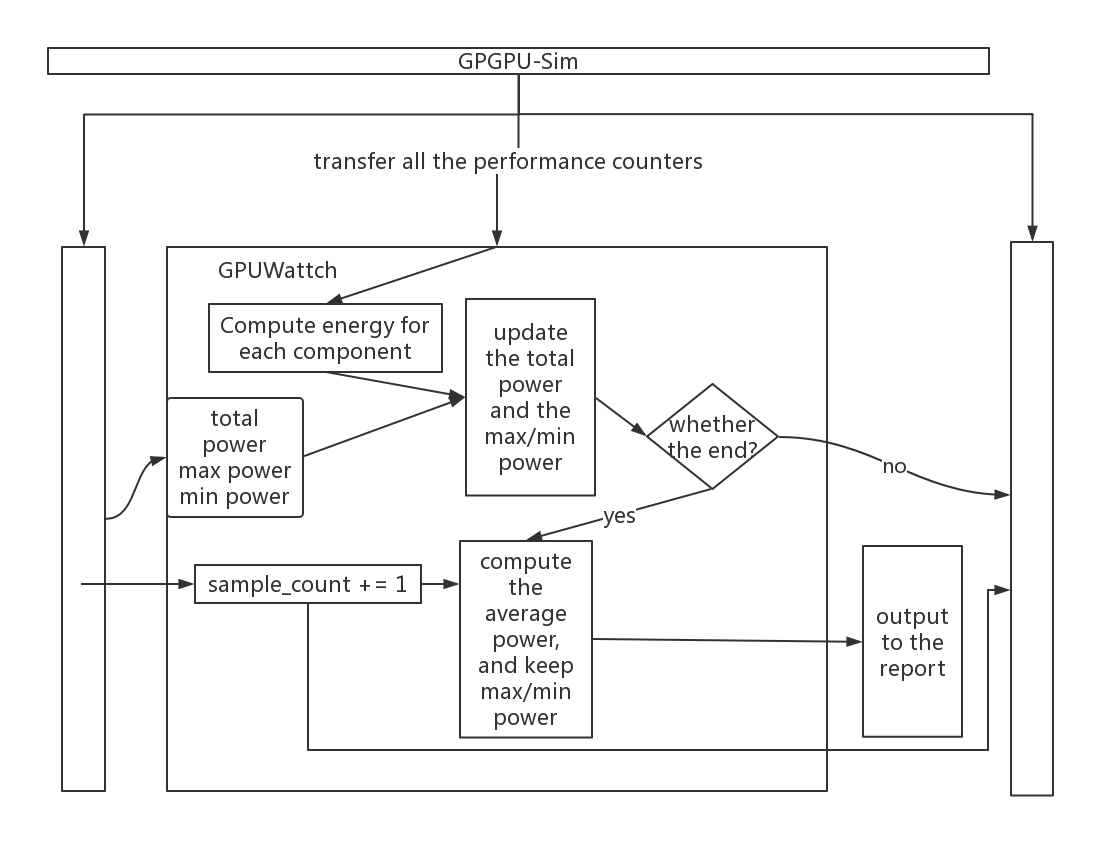
*gpu\_avg\_DC\_RM, = 11.75*

*...*

Every time the GPGPU-Sim finishes executing one sample period (500 cycles by default), it will transfer all the performance counters to GPUWattch and let GPUWattch generate the power report. GPUWattch receives all the performance counters and computes the power of all the components based on the performance counters and parameters set in configuration. In the step of Configuration, GPUWattch sets a constant as the energy consumed for each single action. For example, in the configuration file, if a variable called “IDOE\_CORE\_N” is set as 1.59, it means that during each sample period, the estimation of energy consumed by an idle core is 1.59 Watts. It defines all the details to calculate the energy estimation by performance counters. After computing all the components’ energy, GPUWattch will try to generate the power report if it reaches the end of the program modeled, otherwise it will continue to next cycle.

Its basic process to generate the power report (take one cycle as an example):

Figure 4:Process to generate the report



## Nvprof

Nvprof is a powerful profiling tool added from CUDA5. It is included in the CUDA Toolkit. It is easy to use nvprof in the terminal, you just need to type “nvprof a.out”( assuming you have an executable called a.out), and you will see the output. But nvprof is much more than just a GUI-less version of the profiling features available in the NSight or Visual Profiler. It is light-weight and can reach where other tools can not.

Nvprof supports several modes, the default mode is summary mode. Its sample output is like this:

*==9261== Profiling application: ./tHogbomCleanHemi*

*==9261== Profiling result:*

*Time(%) Time Calls Avg Min Max Name*

*58.73% 737.97ms 1000 737.97us 424.77us 1.1405ms subtractPSFLoop\_kernel(float const \*, int, float\*, int, int, int, int, int, int, int, float, float)*

*38.39% 482.31ms 1001 481.83us 475.74us 492.16us findPeakLoop\_kernel(MaxCandidate\*, float const \*, int)*

*1.87% 23.450ms 2 11.725ms 11.721ms 11.728ms [CUDA memcpy HtoD]*

*1.01% 12.715ms 1002 12.689us 2.1760us 10.502ms [CUDA memcpy DtoH]*

In this mode, nvprof presents an overview of the GPU kernels and memory copies in our application. All calls to the same kernel will be grouped together, and print the total time and the percentage of the total application time for each kernel. It will give programmer useful insights to improve their programs’ algorithms and overcome the bottleneck.

Beside summary mode, nvprof also supports GPU-Trace and API-Trace modes that can let users see a complete list of all kernel launches and memory copies. Moreover, all CUDA API calls are also available.

It also supports remote profiling and some other features.

## GTX480

GPGPU-Sim supports two architectures: NVIDIA G80 and Fermi. In our experiment, we choose Fermi as our basic architecture. GPGPU-Sim provides several configuration files and GTX480 is one of the most popular one, so we choose GTX480 as our basic configuration file but we did some changes to it in order to model our real machine(TITAN V) as precisely as possible. Details about GTX480 will also help model the energy more precisely.

In our experiment, we mainly focus on the following parameters since they are different in TITAN V (There are possibly some other parameters that are different in these two architectures, but here we do not list them because they are hard to find or I do not understand them thoroughly):

https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480

Need more?

|  |  |  |
| --- | --- | --- |
| GTX480 | | |
| Name | Value | Description |
| Number of Cores | 16 | The number of total cores(Streaming Processor) |
| Idle Core Power | 1.59 | The energy consumed by each idle core |
| Clock rate(L1,L2) | 1400MHZ(1400000000) | The clock rate |
| Clock rate(graphics) | 700MHZ(700000000) | The clock rate |
| Memory clock | 200MHZ(200000000) | The clock rate |

## TITAN V

TITAN V is a powerful graphics card created by NVIDIA for the PC in 2018. It is the real GPU we use in real experiments.

We mainly changed the following parameters in the configuration file in order to model the real situation as precisely as possible:

https://www.nvidia.com/en-us/titan/titan-v/

Need more?

|  |  |  |
| --- | --- | --- |
| Name | GTX480(original value) | TITAN V(changed value) |
| Number of Cores | 16 | 5120 |
| Idle Core Power | 1.59 | 0.0318 |
| Clock rate(L1,L2) | 1400MHZ(1400000000) | 1455 MHZ(1455000000) |
| Clock rate(graphics) | 700MHZ(700000000) | 1455 MHZ(1455000000) |
| Memory clock | 200MHZ(200000000) | 850MHZ(850000000) |

## Benchmark

We chose 2 benchmarks. The first one contains only one kernel and it is very simple. The second one contains 4 kernels and it is neural network.

# Implementation

## Extraction of performance counters of GPGPU-Sim

Before we started to transfer nvprof’s metrics, we first got to know the characteristics about GPGPU-Sim’s

Performance counters. So the first thing we did is to extract the performance counters during the whole process. During each sample period (500 cycles by default), GPGPU-Sim will collect all the performance counters and use GPUWattch to generate temporary power metrics and print it in the terminal. When GPGPU-Sim generates temporary power metrics, it will call a function called *mcpat\_cycle* and calculates the power for *instructions, register files, instruction cache, constant/texture cache, shared memory, l1/l2 cache, pipeline, memory controller, execution unit, active lanes for sp and sfu pipelines* and *network on chip* respectively. This function is in the source file */gpgpu-sim/power\_interface.cc*, and I changed this source file. In the original mcpat\_cycle, it will compute all the power metrics and store them in a data structure called power\_stats. This data structure contains all the information about power in GPGPU-Sim, so it is extremely long and not human-readable. So, our first task was to store all these performance counters to a file. The following table describes that which performance counters are need for each component of power.

|  |  |  |  |
| --- | --- | --- | --- |
| Index | Component | Performance Counters | Description |
| 1 | Instructions | TOT\_INST | Counts the all decoded instructions |
| TOT\_FP\_INST | Counts the total number of floating points instructions |
| TOT\_INT\_INST | Counts the total number of integer instructions |
| L1D\_R\_ACC | Counts the all L1 data cache read instructions |
| L1D\_W\_ACC | Counts the all L1 data cache write instructions |
| 2 | register files | REG\_R | Counts the number of register file reads in all instructions |
| REG\_W | Counts the number of register file writes in all instructions |
| NON\_REG\_Ops | Counts the number of non register file operands |
| 3 | instruction cache | IC\_H | Counts the number of instruction cache hits |
| IC\_M | Counts the number of instruction cache misses |
| 4 | Constant cache | CC\_H | Counts the number of constant cache hits |
| CC\_M | Counts the number of constant cache misses |
| 5 | Texture cache | TC\_H | Counts the number of texture cache hits |
| TC\_M | Counts the number of texture cache misses |
| 6 | shared memory | SHRD\_ACC | Counts the number of shared memory accesses |
| 7 | L1 cache | L1\_RH | Counts the number of L1 data cache read hits |
| L1\_RM | Counts the number of L1 data cache read misses |
| L1\_WH | Counts the number of L1 data cache write hits |
| L1\_WM | Counts the number of L1 data cache write misses |
| 8 | L2 cache | L2\_RH | Counts the number of L2 data cache read hits |
| L2\_RM | Counts the number of L2 data cache read misses |
| L2\_WH | Counts the number of L2 data cache write hits |
| L2\_WM | Counts the number of L2 data cache write misses |
| 9 | Idle core | IDLE\_CORE\_N | Counts the average number of idle cores over cycles of each sample |
| 10 | Pipeline | PIPE | Ratio of committed number of instructions to the maximum peak of committed instructions |
| 11 | Memory controller | DRAM\_RD | Counts the dram read accesses |
| DRAM\_WR | Counts the dram writes accesses |
| DRAM\_PRE | Counts the dram precharge accesses |
| 12 | Execution unit | SFU\_ACC | Counts the all instructions that exercise SFU pipeline (it also includes multiplications/division) |
| SP\_ACC | Counts the all instructions that exercise SP pipeline with integer operands |
| FPU\_ACC | Counts the all instructions that exercise SFU pipeline with floating-point operands |
| 13 | Active lanes | SP\_ACTIVE\_LANES | Active lanes for sp pipeline |
| SFU\_ACTIVE\_LANES | Active lanes for sfu pipeline |
| 14 | Network on chip | NOC\_A(SIMT to Mem) | Counts the number of flits traveling from SIMT cluster to memory partition |
|  |  | NOC\_A(Mem to SIMT) | Counts the number of flits traveling from memory partition to SIMT cluster |

For example, to our benchmark 2—neural network, its output is like this:

*0 500 6 4 2 5 0 9 352 192 128 4 0 1 0 0 0 0 5 0 0 0 0 0 0 0 14.976000 0.000733 0 0 0 64 2400 128 0.014000 0.008000 0.000000 0.000000*

*0 500 8 4 4 5 0 3 416 32 64 3 0 1 0 0 0 0 5 0 0 0 0 0 0 0 14.976000 0.000600 0 0 0 0 2900 0 0.014000 0.006000 0.000000 0.000000*

*0 500 6 2 4 5 0 10 512 224 96 4 0 0 0 0 0 0 5 0 0 0 0 0 0 0 14.976000 0.000200 0 0 0 64 3400 128 0.010000 0.004000 0.000000 0.000000*

*0 500 8 7 1 5 0 10 352 192 128 5 0 1 0 0 0 0 5 0 0 0 0 0 0 0 14.976000 0.000667 0 0 0 0 2100 0 0.016000 0.004000 0.000000 0.000000*

*0 500 10 5 5 5 0 8 416 160 160 3 0 1 0 0 0 0 5 0 0 0 0 0 0 0 14.976000 0.000667 0 0 0 0 4500 96 0.016000 0.004000 0.000000 0.000000*

*0 500 6 3 3 6 0 9 480 160 96 5 0 1 0 0 0 0 6 0 0 0 0 0 0 0 14.976000 0.000533 0 0 0 32 4000 0 0.018000 0.004000 0.000000 0.000000*

*0 500 10 5 5 4 0 5 416 128 128 2 0 1 0 0 0 0 4 0 0 0 0 0 0 0 14.976000 0.000600 0 0 0 32 800 128 0.020000 0.004000*

*…….*

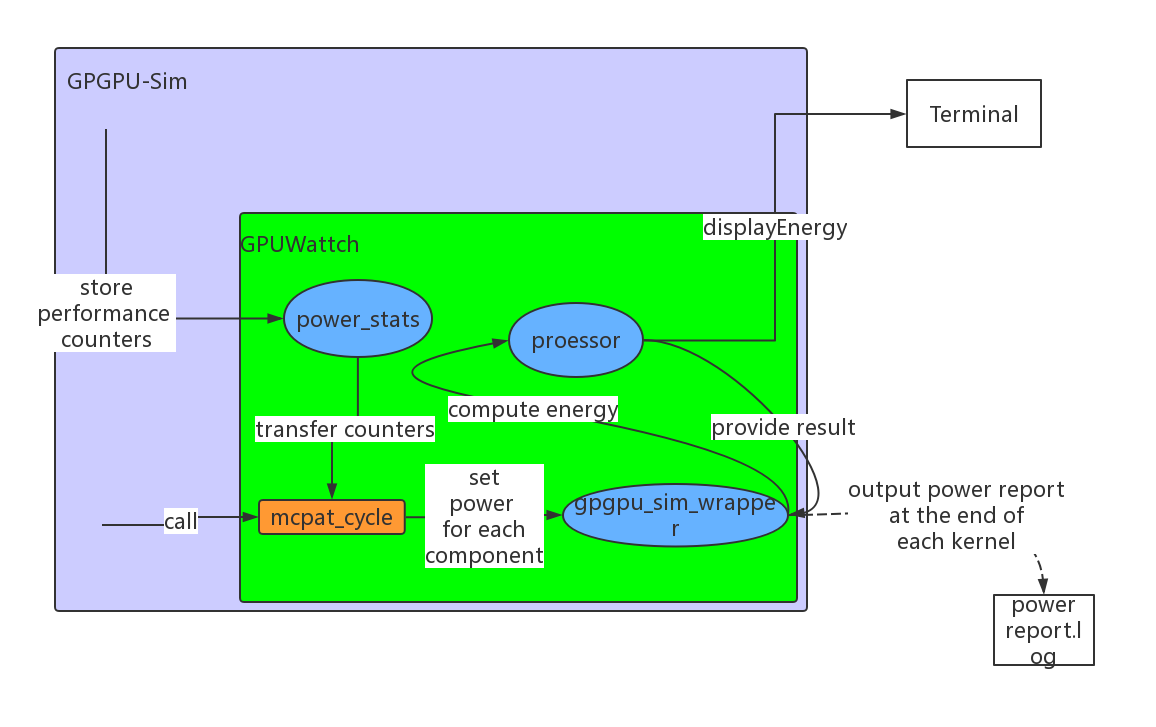
So as we can see, all these performance counters are the input for the final power report. For this benchmark, this output file contains 811345 lines, which means that mcpat\_cycle() is called 811345 times, and there are 81134 periods in total.

So, with these performance counters, we first want to test that whether we can let GPGPU-Sim generate the final power report based on these pre-defined parameters instead of modeling the whole program. Then, our main task is to build GPUWattch or call GPUWattch alone separately from GPGPU-Sim.

## Use GPUWattch separately from GPGPU-Sim

To achieve our final goal, first we want to use GPUWattch based on the parameters in a specific file. In original GPGPU-Sim, GPUWattch is a part of it. Every sample period, GPUWattch will be called, the function mcpat\_cycle will be executed. First, it will get all the performance counters form a variable called power\_stats. Then, it will set power for each component with some preprocess, and store the results of calculation in a global variable gpgpu\_sim\_wrapper. Then, gpgpu\_sim\_wrapper will call the variable processor to do further computation and get the final power result. After calculation, processor will call its function displayEnergy to print the results in terminal. Meanwhile, if it reaches the end of a kernel, gpgpu\_sim\_wrapper will output the final power report based on the results processor provides.

The below picture describes the process.



In the original GPGPU-Sim, GPUWattch cannot be called directly from application. We want to let GPUWattch use the parameters from outside file, so we first wrote a separate file called main.cc and defines the mcpat\_cycle function inside. In the function, we first read in one line from the file, and store all the performance counters. After getting all the performance counters, we want to call the function of wrapper and calculate the power result. So, we have to link the obeject files(.o).

First, we compile and build the original GPGPU-Sim directory, and do not clean all the .o files. There are plenty of .o files, and their dependency relations are complex. Luckily, we found a set of .o files that can include all the functions of GPUWattch without dependency on other GPGPU-Sim .o files.

First, we just enter the build folder to see its structure. The below chart shows the structure of the folder and all its subfolders.

And the set of the .o files are :

abstract\_hardware\_model.o core.o logic.o sharedcache.o arbiter.o crossbar.o main.o subarray.o area.o decoder.o mat.o technology.o array.o gpgpusim\_entrypoint.o memoryctrl.o Ucache.o bank.o gpgpu\_sim\_wrapper.o noc.o uca.o basic\_circuit.o htree2.o nuca.o wire.o asic\_components.o interconnect.o parameter.o XML\_Parse.o cacti\_interface.o iocontrollers.o processor.o xmlParser.o component.o io.o router.o

|  |  |  |
| --- | --- | --- |
| The build folder, **./build/gcc/cuda/debug** | **cuda-sim** |  |
| **intersim2** |  |
| **gpuwattch** | core.o |
| sharedcache.o |
| arbiter.o |
| crossbar.o |
| subarray.o |
| area.o |
| decoder.o |
| mat.o |
| technology.o |
| array.o |
| memoryctrl.o |
| Ucache.o |
| bank.o |
| gpgpu\_sim\_wrapper.o |
| noc.o |
| uca.o |
| basic\_circuit.o |
| htree2.o |
| nuca.o |
| wire.o |
| basic\_components.o |
| interconnect.o |
| parameter.o |
| XML\_Parse.o |
| cacti\_interface.o |
| iocontrollers.o |
| processor.o |
| xmlParser.o |
| component.o |
| io.o |
| router.o |
| **gpgpu-sim** |  |
| **cuobjdump\_to\_ptxplus** |  |
| **libcuda** |  |
| gpgpusim\_entrypoint.o |  |
| abstract\_hardware\_model.o |  |
| logic.o |  |

When we copy all these .o files to a folder where our main.cc stays, more modifications still need to be done in order that our program can use the functions in GPUWattch successfully. First, we initialize a variable whose type is gpgpu\_sim\_wrapper. And if wrapper can be initialized, it means we can generate the results if there are meaningful input. Moreover, initializing gpgpu\_sim\_wrapper will also initialize a lot of other variables which are attributes of the gpgpu\_sim\_wrapper, such as processor. So, actually if we have a variable whose type is gpgpu\_sim\_wrapper, we will be able to use most functions related to GPUWattch.

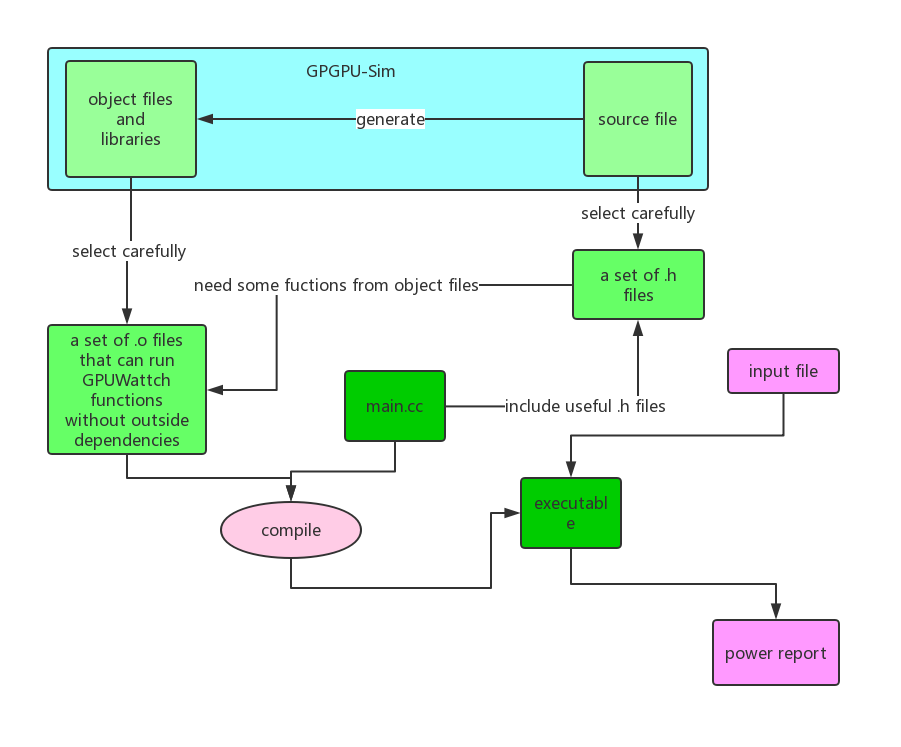
After putting all the .o files and main.cc in the same directory, we have to design the statement to compile the main.cc. If we simply use g++ and include all the .o files and main.cc, it will undoubtedly fail because in the stage of compilation the compiler will report error of type not found. So, we have to include the .h files. In order to be safe, we include all the .h files that are included in power\_interface.cc.

So finally our command to compile is

*/usr/bin/g++-4.8 -std=c++11 -pthread -Wall -g -I/home/zongyi/GPGPU-Sim-with-output/src/gpuwattch/ main.cc -o test arbiter.o area.o array.o bank.o basic\_circuit.o basic\_components.o cacti\_interface.o component.o core.o crossbar.o decoder.o htree2.o interconnect.o iocontrollers.o io.o logic.o mat.o memoryctrl.o noc.o nuca.o parameter.o processor.o router.o sharedcache.o subarray.o technology.o Ucache.o uca.o wire.o XML\_Parse.o xmlParser.o gpgpu\_sim\_wrapper.o libcudart.so –lz*

Note that –lz means we include the library to write/read .gz files, like “gzprint”. And libcudart.so is the dynamic library in /lib/gcc/cuda/debug.

So, our work flow is like this:



After this, we can use the GPUWattch successfully with our input file. First, we use our first benchmark which contains only one kernel. The below is the average power metrics for this program in the power report generated by original GPGPU-Sim..

*kernel\_name = \_Z6vecAddPfS\_S\_i*

*kernel\_launch\_uid = 1*

*Kernel Average Power Data:*

*kernel\_avg\_power = 34.9116*

*gpu\_avg\_IBP, = 2.44618e-05*

*gpu\_avg\_ICP, = 0.000413327*

*gpu\_avg\_DCP, = 0.00167244*

*gpu\_avg\_TCP, = 0*

*gpu\_avg\_CCP, = 0.00157412*

*gpu\_avg\_SHRDP, = 0*

*gpu\_avg\_RFP, = 0.00760162*

*gpu\_avg\_SPP, = 0.00662806*

*gpu\_avg\_SFUP, = 0.00380697*

*gpu\_avg\_FPUP, = 0.00029061*

*gpu\_avg\_SCHEDP, = 0.000337694*

*gpu\_avg\_L2CP, = 0.00468699*

*gpu\_avg\_MCP, = 1.18063*

*gpu\_avg\_NOCP, = 0.0107589*

*gpu\_avg\_DRAMP, = 0.0566805*

*gpu\_avg\_PIPEP, = 0.00820386*

*gpu\_avg\_IDLE\_COREP, = 23.8046*

*gpu\_avg\_CONST\_DYNAMICP = 9.82366*

*gpu\_avg\_TOT\_INST, = 4.8346*

*gpu\_avg\_FP\_INT, = 4.39446*

*gpu\_avg\_IC\_H, = 2.35032*

*gpu\_avg\_IC\_M, = 0.0673684*

*gpu\_avg\_DC\_RH, = 0*

*gpu\_avg\_DC\_RM, = 0.438596*

*gpu\_avg\_DC\_WH, = 0*

*gpu\_avg\_DC\_WM, = 0.219298*

*gpu\_avg\_TC\_H, = 0*

*gpu\_avg\_TC\_M, = 0*

*gpu\_avg\_CC\_H, = 0.846596*

*gpu\_avg\_CC\_M, = 0.0336842*

*gpu\_avg\_SHRD\_ACC, = 0*

*gpu\_avg\_REG\_RD, = 147.492*

*gpu\_avg\_REG\_WR, = 105.079*

*gpu\_avg\_NON\_REG\_OPs, = 70.3484*

*gpu\_avg\_SP\_ACC, = 1756.86*

*gpu\_avg\_SFU\_ACC, = 56.2392*

*gpu\_avg\_FPU\_ACC, = 7.01754*

*gpu\_avg\_MEM\_RD, = 1.31621*

*gpu\_avg\_MEM\_WR, = 0.438596*

*gpu\_avg\_MEM\_PRE, = 0.145193*

*gpu\_avg\_L2\_RH, = 0.00294737*

*gpu\_avg\_L2\_RM, = 0.438807*

*gpu\_avg\_L2\_WH, = 0*

*gpu\_avg\_L2\_WM, = 0.219298*

*gpu\_avg\_NOC\_A, = 3.96421*

*gpu\_avg\_PIPE\_A, = 0.000322204*

*gpu\_avg\_IDLE\_CORE\_N, = 14.9715*

*gpu\_avg\_CONST\_DYNAMICN = 0*

And the power report generate by our main.cc with the input file we set manually. The input file contains all parameters for each cycle generated by our modified version of GPGPU-Sim. ( see “Extraction of performance counters”)

The results of these two method are totally identical, which helps we implement next step.

## Get power metrics of CUDA program running on real GPU

Our first expectation is to find a method to output the details of power metrics every cycle or every sample period (like 500 cycles). But we did not find such a tool, the best power metrics we can get is generated by a tool called *nvprof*. It can give the details of each kernel of the user application.

But it is not easy to get the information we want. There are many options that we have to decide, the first is the *pc-sampling-period*. In our experiment, we found that if we set this option different values, the metrics will be slightly different. The available choices are integers from 5 to 31, and if we set this option as x, the sampling period will become 2^x(E.g. , we set the parameters as 5, the sampling period will be 2^5 = 32 cycles). If we set this parameter too small, the metric will usually smaller. It is possibly because after the sampling period there are some overhead. So, in order to minimum the impact, we set this option to 31.

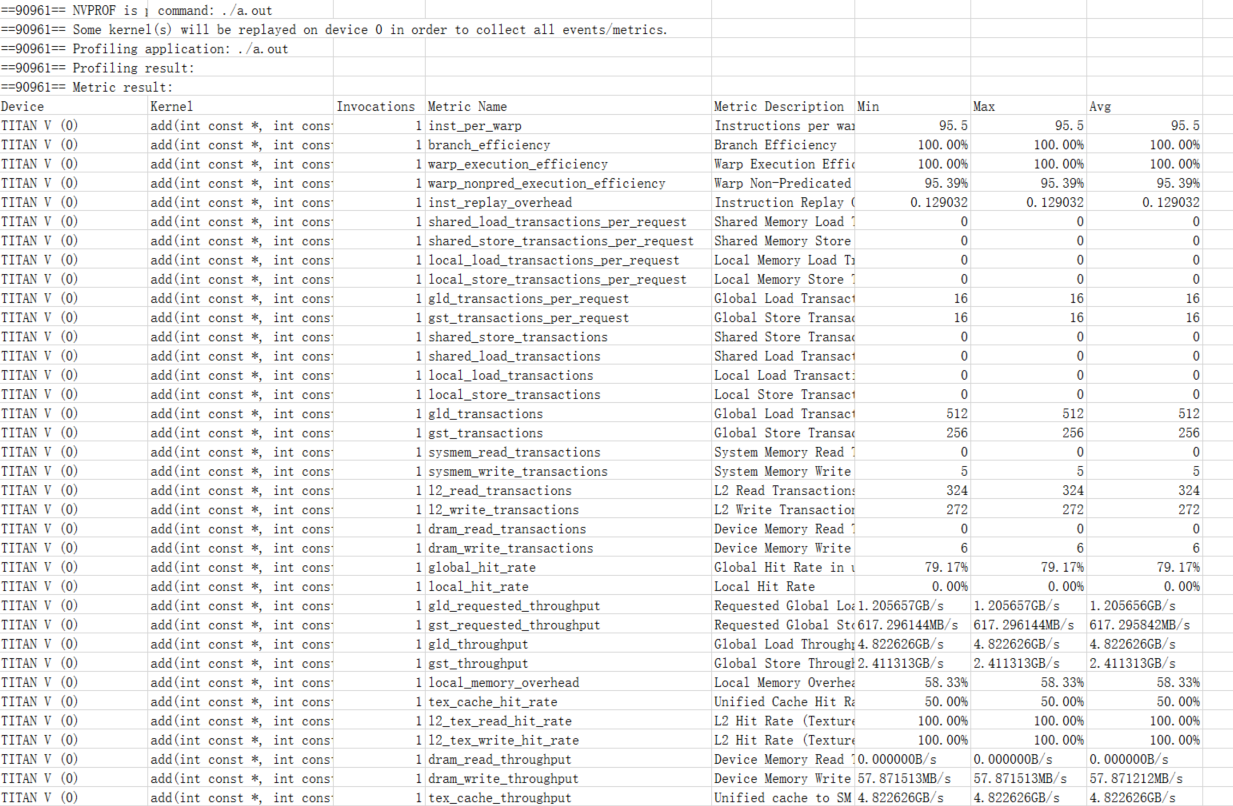
Meanwhile, we set the Concurrent-kernel option to be off. It is because that, GPGPU-Sim models all the application in a serialized order and we want to make the situations in nvprof and GPGPU-Sim as similar as possible. So, we set this parameter to off and turn off the feature, hoping it will make the metrics from nvprof more meaningful.

Our usual command is like this:

*Nvprof –concurrent-kernels off –metrics all –csv –log-file nvprof\_result.csv –pc-sampling-period 31 a.out*

Then, we get all metrics nvprof can provide based on each kernel.

We store the result in a .csv file for better reading, and it is like this:



The below table shows all the metrics available from nvprof..

|  |  |
| --- | --- |
| Metric Name | Description |
| inst\_per\_warp | Average number of instructions executed by each warp |
| branch\_efficiency | Ratio of branch instruction to sum of branch and divergent branch instruction |
| warp\_execution\_efficiency | Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor |
| warp\_nonpred\_execution\_efficiency | Ratio of the average active threads per warp executing non-predicated instructions to the maximum number of threads per warp supported on a multiprocessor |
| inst\_replay\_overhead | Average number of replays for each instruction executed |
| shared\_load\_transactions\_per\_request | Average number of shared memory load transactions performed for each shared memory load |
| shared\_store\_transactions\_per\_request | Average number of shared memory store transactions performed for each shared memory store |
| local\_load\_transactions\_per\_request | Average number of local memory load transactions performed for each local memory load |
| local\_store\_transactions\_per\_request | Average number of local memory store transactions performed for each local memory store |
| gld\_transactions\_per\_request | Average number of global memory load transactions performed for each global memory load. |
| gst\_transactions\_per\_request | Average number of global memory store transactions performed for each global memory store |
| shared\_store\_transactions | Number of shared memory store transactions |
| shared\_load\_transactions | Number of shared memory load transactions |
| local\_load\_transactions | Number of local memory load transactions |
| local\_store\_transactions | Number of local memory store transactions |
| gld\_transactions | Number of global memory load transactions |
| gst\_transactions | Number of global memory store transactions |
| sysmem\_read\_transactions | Number of system memory read transactions |
| sysmem\_write\_transactions | Number of system memory write transactions |
| l2\_read\_transactions | Memory read transactions seen at L2 cache for all read requests |
| l2\_write\_transactions | Memory write transactions seen at L2 cache for all write requests |
| dram\_read\_transactions | Device memory read transactions |
| dram\_write\_transactions | Device memory write transactions |
| global\_hit\_rate | Hit rate for global load and store in unified l1/tex cache |
| local\_hit\_rate | Hit rate for local loads and stores |
| gld\_requested\_throughput | Requested global memory load throughput |
| gst\_requested\_throughput | Requested global memory store throughput |
| gld\_throughput | Global memory load throughput |
| gst\_throughput | Global memory store throughput |
| local\_memory\_overhead | Ratio of local memory traffic to total memory traffic between the L1 and L2 caches |
| tex\_cache\_hit\_rate | Unified cache hit rate |
| l2\_tex\_read\_hit\_rate | Hit rate at L2 cache for all read requests from texture cache |
| l2\_tex\_write\_hit\_rate | Hit Rate at L2 cache for all write requests from texture cache |
| dram\_read\_throughput | Device memory read throughput |
| dram\_write\_throughput | Device memory write throughput |
| tex\_cache\_throughput | Unified cache to SM read throughput |
| l2\_tex\_read\_throughput | Memory read throughput seen at L2 cache for read requests from the texture cache |
| l2\_tex\_write\_throughput | Memory write throughput seen at L2 cache for write requests from the texture cache |
| l2\_read\_throughput | Memory read throughput seen at L2 cache for all read requests |
| l2\_write\_throughput | Memory write throughput seen at L2 cache for all write requests |
| sysmem\_read\_throughput | System memory read throughput |
| sysmem\_write\_throughput | System memory write throughput |
| local\_load\_throughput | Local memory load throughput |
| local\_store\_throughput | Local memory store throughput |
| shared\_load\_throughput | Shared memory load throughput |
| shared\_store\_throughput | Shared memory store throughput |
| gld\_efficiency | Ratio of requested global memory load throughput to required global memory load throughput. |
| gst\_efficiency | Ratio of requested global memory store throughput to required global memory store throughput. |
| tex\_cache\_transactions | Unified cache to SM read transactions |
| flop\_count\_dp | Number of double-precision floating-point operations executed by non-predicated threads (add, multiply, and multiply-accumulate). Each multiply-accumulate operation contributes 2 to the count. |
| flop\_count\_dp\_add | Number of double-precision floating-point add operations executed by non-predicated threads. |
| flop\_count\_dp\_fma | Number of double-precision floating-point multiply-accumulate operations executed by non-predicated threads. Each multiply-accumulate operation contributes 1 to the count. |
| flop\_count\_dp\_mul | Number of double-precision floating-point multiply operations executed by non-predicated threads. |
| flop\_count\_sp | Number of single-precision floating-point operations executed by non-predicated threads (add, multiply, and multiply-accumulate). Each multiply-accumulate operation contributes 2 to the count. The count does not include special operations. |
| flop\_count\_sp\_add | Number of single-precision floating-point add operations executed by non-predicated threads. |
| flop\_count\_sp\_fma | Number of single-precision floating-point multiply-accumulate operations executed by non-predicated threads. Each multiply-accumulate operation contributes 1 to the count. |
| flop\_count\_sp\_mul | Number of single-precision floating-point multiply operations executed by non-predicated threads. |
| flop\_count\_sp\_special | Number of single-precision floating-point special operations executed by non-predicated threads. |
| inst\_executed | The number of instructions executed |
| inst\_issued | The number of instructions issued |
| dram\_utilization | The utilization level of the device memory relative to the peak utilization on a scale of 0 to 10 |
| sysmem\_utilization | The utilization level of the system memory relative to the peak utilization |
| stall\_inst\_fetch | Percentage of stalls occurring because the next assembly instruction has not yet been fetched |
| stall\_exec\_dependency | Percentage of stalls occurring because an input required by the instruction is not yet available |
| stall\_memory\_dependency | Percentage of stalls occurring because a memory operation cannot be performed due to the required resources not being available or fully utilized, or because too many requests of a given type are outstanding |
| stall\_texture | Percentage of stalls occurring because the texture sub-system is fully utilized or has too many outstanding requests |
| stall\_sync | Percentage of stalls occurring because the warp is blocked at a \_\_syncthreads() call |
| stall\_other | Percentage of stalls occurring due to miscellaneous reasons |
| stall\_constant\_memory\_dependency | Percentage of stalls occurring because of immediate constant cache miss |
| stall\_pipe\_busy | Percentage of stalls occurring because a compute operation cannot be performed because the compute pipeline is busy |
| shared\_efficiency | Ratio of requested shared memory throughput to required shared memory throughput |
| inst\_fp\_32 | Number of single-precision floating-point instructions executed by non-predicated threads (arithmetic, compare, etc.) |
| inst\_fp\_64 | Number of double-precision floating-point instructions executed by non-predicated threads (arithmetic, compare, etc.) |
| inst\_integer | Number of integer instructions executed by non-predicated threads |
| inst\_bit\_convert | Number of bit-conversion instructions executed by non-predicated threads |
| inst\_control | Number of control-flow instructions executed by non-predicated threads (jump, branch, etc.) |
| inst\_compute\_ld\_st | Number of compute load/store instructions executed by non-predicated threads |
| inst\_misc | Number of miscellaneous instructions executed by non-predicated threads |
| inst\_inter\_thread\_communication | Number of inter-thread communication instructions executed by non-predicated threads |
| issue\_slots | The number of issue slots used |
| cf\_issued | Number of issued control-flow instructions |
| cf\_executed | Number of executed control-flow instructions |
| ldst\_issued | Number of issued local, global, shared and texture memory load and store instructions |
| ldst\_executed | Number of executed local, global, shared and texture memory load and store instructions |
| atomic\_transactions | Global memory atomic and reduction transactions |
| atomic\_transactions\_per\_request | Average number of global memory atomic and reduction transactions performed for each atomic and reduction instruction |
| l2\_atomic\_throughput | Memory read throughput seen at L2 cache for atomic and reduction requests |
| l2\_atomic\_transactions | Memory read transactions seen at L2 cache for atomic and reduction requests |
| l2\_tex\_read\_transactions | Memory read transactions seen at L2 cache for read requests from the texture cache |
| stall\_memory\_throttle | Percentage of stalls occurring because of memory throttle |
| stall\_not\_selected | Percentage of stalls occurring because warp was not selected |
| l2\_tex\_write\_transactions | Memory write transactions seen at L2 cache for write requests from the texture cache |
| flop\_count\_hp | Number of half-precision floating-point operations executed by non-predicated threads (add, multiply, and multiply-accumulate). Each multiply-accumulate contributes 2 or 4 to the count based on the number of inputs. |
| flop\_count\_hp\_add | Number of half-precision floating-point add operations executed by non-predicated threads. |
| flop\_count\_hp\_mul | Number of half-precision floating-point multiply operations executed by non-predicated threads. |
| flop\_count\_hp\_fma | Number of half-precision floating-point multiply-accumulate operations executed by non-predicated threads. Each multiply-accumulate contributes 2 or 4 to the count based on the number of inputs. |
| inst\_fp\_16 | Number of half-precision floating-point instructions executed by non-predicated threads (arithmetic, compare, etc.) |
| ipc | Instructions executed per cycle |
| issued\_ipc | Instructions issued per cycle |
| issue\_slot\_utilization | Percentage of issue slots that issued at least one instruction, averaged across all cycles |
| sm\_efficiency | The percentage of time at least one warp is active on a specific multiprocessor |
| achieved\_occupancy | Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor |
| eligible\_warps\_per\_cycle | Average number of warps that are eligible to issue per active cycle |
| shared\_utilization | The utilization level of the shared memory relative to peak utilization |
| l2\_utilization | The utilization level of the L2 cache relative to the peak utilization on a scale of 0 to 10 |
| tex\_utilization | The utilization level of the unified cache relative to the peak utilization |
| ldst\_fu\_utilization | The utilization level of the multiprocessor function units that execute shared load, shared store and constant load instructions |
| cf\_fu\_utilization | The utilization level of the multiprocessor function units that execute control-flow instructions on a scale of 0 to 10 |
| tex\_fu\_utilization | The utilization level of the multiprocessor function units that execute global, local and texture memory instructions on a scale of 0 to 10 |
| special\_fu\_utilization | The utilization level of the multiprocessor function units that execute sin, cos, ex2, popc, flo, and similar instructions on a scale of 0 to 10 |
| half\_precision\_fu\_utilization | The utilization level of the multiprocessor function units that execute 16 bit floating-point instructions and integer instructions on a scale of 0 to 10 |
| single\_precision\_fu\_utilization | The utilization level of the multiprocessor function units that execute (FFMA, IMAD, FADD, FMUL and FSWZADD) instructions |
| double\_precision\_fu\_utilization | The utilization level of the multiprocessor function units that execute double-precision floating-point instructions |
| flop\_hp\_efficiency | Ratio of achieved to peak half-precision floating-point operations |
| flop\_sp\_efficiency | Ratio of achieved to peak single-precision floating-point operations |
| flop\_dp\_efficiency | Ratio of achieved to peak double-precision floating-point operations |
| sysmem\_read\_utilization | The read utilization level of the system memory relative to the peak utilization on a scale of 0 to 10 |
| sysmem\_write\_utilization | The write utilization level of the system memory relative to the peak utilization on a scale of 0 to 10 |
| stall\_sleeping | Percentage of stalls occurring because warp was sleeping |

## Transform nvprof metrics to GPUWattch

It is clear that there is a great gap between GPGPU-Sim and nvprof. The parameters they use are different, and the way the metrics are organized is also different.

First, we dealt with the gap that nvprof only reports metrics about each kernel, while GPUWattch’s input is for each sample period. To overcome this gap, we have to let GPUWattch consumes parameters only once for the whole kernel. There is an assumption lying behind that, the energy consumed is increasing in the linear scale of the performance counters in each cycle. Actually this may not always right, but in GPGPU-Sim most components’ energy is calculated by performance counter multiples a constant defined in configuration file. So this method is reasonable to some degree, but we have to do special operations to some components.

Here is where now I am writing(20190416, Tuesday)

## Reference

1. Analyzing CUDA Workloads Using a Detailed GPU Simulator
2. Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, Vijay Janapa Reddi, GPUWattch: Enabling Energy Optimizations in GPGPUs, In proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA 2013), Tel-Aviv, Israel, June 23-27, 2013.
3. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures
4. https://www.nvidia.com