GPUWattch Energy Model Manual
Version 1.0

Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, Vijay Janapa Reddi


1. Introduction

1.1. Overview

This manual describes GPUWattch, an energy model based upon McPAT that is integrated with GPGPU-Sim version 3.2.0 and later. GPUWattch was collaboratively developed by researchers at UTAustin, UWisconsin, and UBC.

General-purpose GPU architectures are becoming increasingly prevalent in mainstream computing, and as such they require judicious optimization for energy efficiency. To enable such research, we propose a new GPU power model that offers flexibility, adaptability, and stability. Flexibility is achieved by using a bottom-up methodology and abstracting parameters from the microarchitectural components as model inputs. Adaptability ensures that both program and microarchitectural level interactions are captured during execution, thereby enabling new power-management techniques specifically targeted at GPUs. Stability is examined by validating the power model against measurements of two commercial GPUs comprehensively through the leakage power, average dynamic power, and dynamic power trace. The measured error is within 9.7% and 13.6% across our evaluated benchmark suite for the two target GPUs (GTX 480 and Quadro FX 5600 respectively) and the model accurately tracks the relative power consumption trend over time.

The power model modifies and extends the McPAT CPU power model simulator to model the power of contemporary GPGPUs and drive the modified McPAT version with a cycle-accurate simulator, GPGPU-Sim. This manual first provides an overview of the architecture modeled by GPUWattch, how it is implemented in GPGPU-Sim and McPAT, and the accuracy of GPUWattch with real hardware. Secondly, the GPGPU-Sim performance counters that drive the power model are discussed. Thirdly, the manual describes how to use the power model. Finally, a brief overview of the software design for GPUWattch is presented.

1.2. McPAT

McPAT is an architectural modeling tool for chip multiprocessors (CMP) The main focus of McPAT is accurate power and area modeling, 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.

For complete documentation of the McPAT, please refer McPAT 1.0 technical report and the following paper, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures", that appears in MICRO 2009. Currently we build upon McPAT 0.8 beta release.

1.3. Interface

Figure 1 below illustrates the structure of the power model and how it interfaces with GPGPU-Sim. We modified McPAT to account for the GPU specific architectural components. This included modifying existing structures (such as data caches) to match the GPU configuration and adding new structures (such as the memory coalescing logic). GPGPU-Sim was modified to include performance counters required by McPAT to account for the activity of each modelled component.

GPGPU-Sim passes these performance counters to McPAT through an interface (described in Software Design). McPAT then computes the estimated static and dynamic powers. The runtime power statistics generated by McPAT can be passed back to GPGPU-Sim to perform feedback-driven dynamic optimizations (such as DVFS).

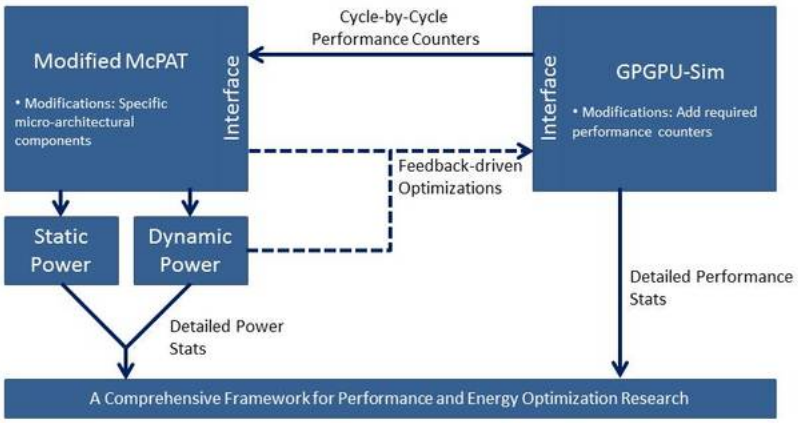

Figure 1: GPUWattch Structure

2. Microarchitecture Power Model

This section presents an overview of the mircroarchitectural components modeled by GPUWattch, how they are implemented in McPAT, and the overall accuracy of GPUWattch against two commercially available GPUs (NVIDIA GTX480 and Quadro FX 5600).

2.1. Overview

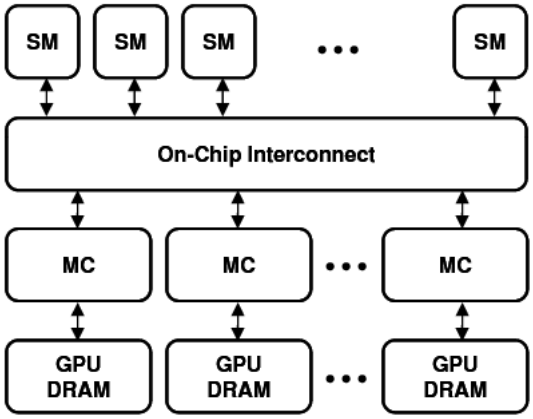

Figure 2: Main Components Modeled in the GPU Architecture.

Currently, we only model GPU architectures similar to NVIDIA's GPUs. As shown in the above figure, the major components in GPU include streaming multiprocessors (SMs), interconnection network, memory controllers (MCs), and GDDR main memory.

For most of these components (SM, interconnection and MC), we used the basic structures (SRAM, logic, wires) to model them. The details of their modeling is explained in the next section. The main memory is only component that we use analytical method to model because we focus on the modeling components in the GPU processor, but the DRAM power model is required for us to compare the modeled power with the real hardware measurement since it includes the DRAM power.

2.2. Main Components

The following table lists the components modeled in our power model.

	Description	Microarchitectural Comoent	Basic Structures
SM	SM pipeline	Pipeline	In oder multithreaded SIMD pipelines
	Caches	Instruction Cache	Architecture dependent size
		L1 Data Cache	
		Texture Cache	
		Constant Cache	
		Memory Coalesce Logic	Synthesis based power model
	Shared Memory	Shared Memory Banks	SRAM
		Shared Memory Crossbar Network	32x32 crossbar network

	Register Files	Register File Banks	1024-bit wide, 64-entry SRAM
		Operands Collectors	1024-bit wide SRAM
		Operand Collection Crossbar Network	1024-bit wide 16x16 crossbar
	Execution Units	Integer ALU (ALU)	Synthesis based power model
		Floating Point Unit (FPU)	
		Special Function Unit (SFU)	
Memory	Memory Controller	On-chip Memory Controller	McPAT's model
Interconnection Network	Interconnect	Network On Chip	
DRAM	Main Memory	GDDR5/GDDR3	Empirical model

2.2.1. Streaming multiprocessor (SM)

Each SM is modeled as an in-order multi-threaded core. The maximum number of hardware threads in each SM is equal to the maximum number of warps per SM in the modeled GPU architecture (e.g. 48 warps per SM for Fermi). All SMs in the GPU are modeled to be identical (homogenous_cores=1 in GPUWattch/McPAT XML file). The tecnology node that a GPU was fabricated at can be set modifying core_tech_node parameter in GPUWattch/McPAT XML file (default=40nm for Fermi). The SM clock frequency is set by target_core_clockrate parameter in the XML file (default=700MHz for Fermi).

2.2.2. Caches

GPGPU has various caches: instruction cache, L1/L2 data cache, texture cache, and constant cache. G80 architecture does not have the L1 data cache. These caches are modeled using the regular cache structure inside McPAT. The cache parameters are configured as the same of the corresponding architecture.

2.2.3. Memory Coalesce Logic (MCL)

The load-store units in GPUs are responsible for coalescing memory accesses to reduce the bandwidth usage. The main components of MCL are shown in the previous figure. Each SM's MCL contains a pending request table (PRT) with multiple entries. Each PRT entry stores the thread IDs, the base and offset addresses for the memory access, and the request sizes for all of the concurrently issued memory requests by a warp. These entries are written to the PRT whenever a memory request is issued. We create a Verilog HDL description of the MCL unit for initial power estimation.

To determine the number of coalesced memory requests that a warp must issue, MCL compares the base address of the first thread's request with the base addresses of all the remaining requests in the PRT entry. The memory request mask of all the SIMT threads with the same base address is set to zero. The process is repeated until masks have been generated for all requests in the PRT entry. These masks are stored in a separate thread masks array. Memory requests are generated for all addresses whose mask bit is set. For each PRT entry, a pending request count (PRC) is maintained, which is incremented when a request is sent to memory and decremented when a response is received. When the PRC becomes zero, requests for the warp are satisfied. We model the PRT as an SRAM array structure using CACTI.

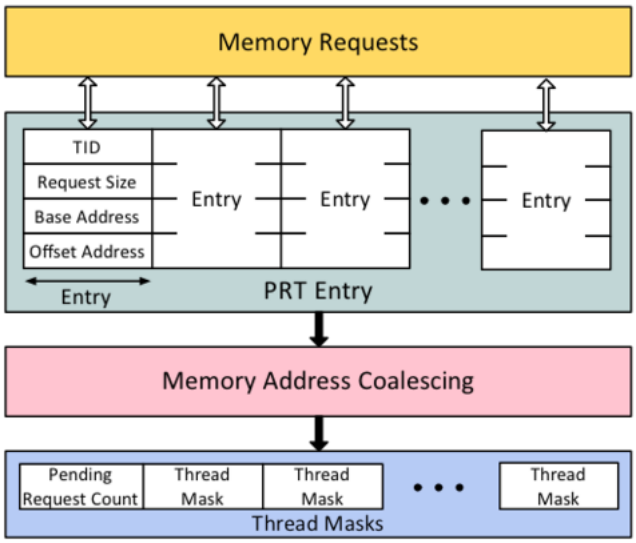

Figure 3: Memory Coalescing Logic (MCL).

2.2.4. Register Files

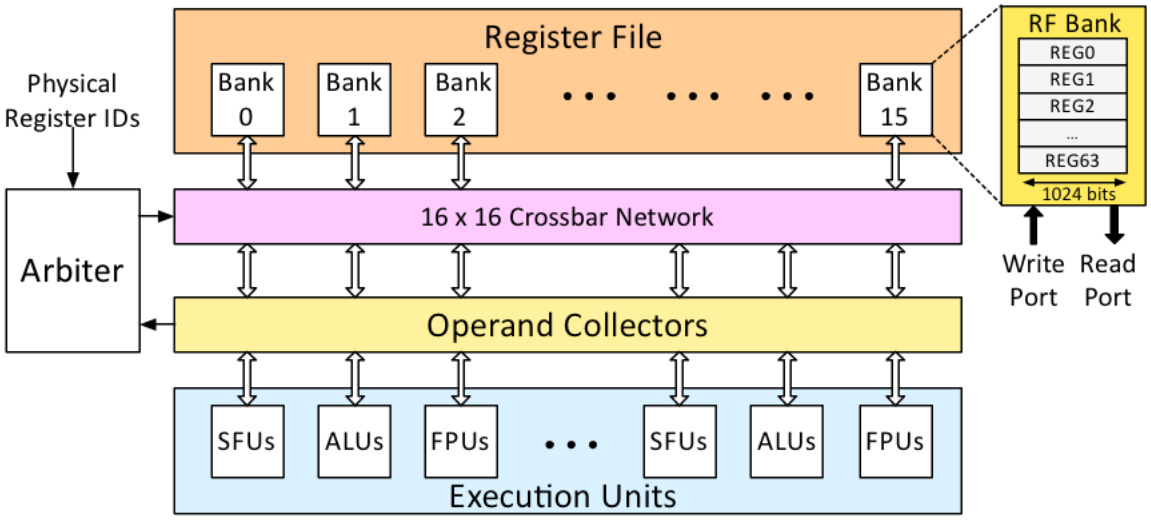

Figure 4: Register File.

Due to the lack of resources for the NVIDIA GPU register file architecture, we adopt the architecture used in GPGPU-Sim version 3.1.1 (in the above figure). Each SM has a large and unified register file that is shared across warps executing in the same SM. GPUs adopt a multi-banked register file architecture in order to avoid the area and power overhead of a multi-ported register file. Crossbar networks and operand collectors are used to transport operands from different banks to the appropriate SIMD execution lane.

The number and size of banks are decided by the register file's bandwidth and size. We model the register files with multiple dual-ported (one read and one write port) banks. the number of register file banks is set by "rf_banks" parameter in the XML file.

A crossbar interconnection network is used to transfer operands from register file banks to proper operand collectors. Furthermore, in the case of bank conflicts, the arbiter shown in the above figure is responsible for serializing the register file access, and the operand collectors are used to buffer the data already read from the register file. Each crossbarr link is as wide as an RF bank. The dimensions of the crossbar network are set by "rf_banks" and "collector_units" parameters in the XML file. Overall, the crossbar is "rf_banks" x "collector_units".

We model each operand collector as an SRAM array bank with 8-entries per collector unit.

2.2.5. Shared Memory

NVIDIA GPUs contain shared memory per SM for each thread block. It can be used for inter-thread communication for threads in a thread block. Because SIMD lanes can access any address concurrently, shared memory is multi-banked and contains a crossbar interconnect to improve performance. That is, shared memory has a very similar structure to the register files. Shared memory is configured similar to caches (using the capacity, block_width, associativity, bank, throughput, latency, output_width, cache_policy parameters in XML file for 'sharedmemory'). We modeled shared memory as 'pure_ram' in McPAT and parameters such as associativity and cache_policy are not used by McPAT. In the case of the Fermi architecture, the number of banks is 32 and the crossbar is 32x32.

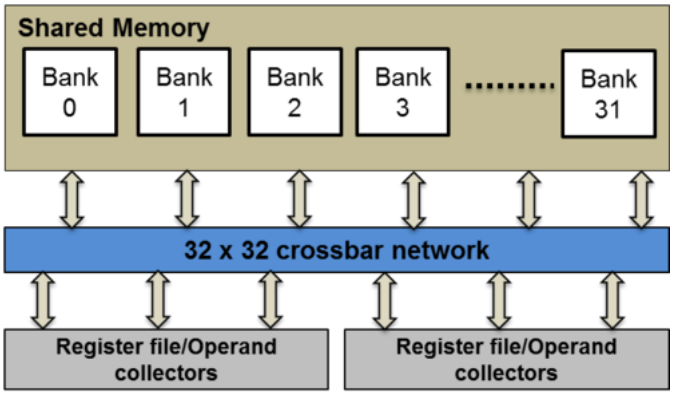

Figure 5: Shared Memory Banks.

2.2.6. Execution Units

GPU architectures typically have two kinds of execution pipelines: a) a unified pipeline for integer and floating-point (FP) operations e.g. in NVIDIA G80 or AMD GPUs, or b) seperate integer and FP pipelines e.g. in Nvidia Fermi and Kepler GPUs.

We model the FP pipeline using floating-point FMA units. We assume that two lanes of FMA units can combine to execute a double-precision (DP) operation or a 32-bit integer multiplication.

The SFU units are also modeled as double-precision FMA units. Because most instructions executed on the SFU units are transcendental operations that employ iterative algorithms to compute the result, these units' power consumption depends on the latency and throughput of instructions.

Area and power modeling of execution units are initially estimated based synthesis of their Verilog descriptions. The execution units were synthesized using a 45-nm standard cell library. The synthesized netlists are annotated with switching activity for random inputs. We use Power Compiler to estimate the power consumption of the

designs after the annotation of switching activity. All designs assume an operating nominal supply voltage of 1 V. We perform technology and voltage scaling according to ITRS projections to estimate the dynamic power consumption at the given voltage and technology node.

In order to provide estimates for both types of execution pipelines, we model both a unified integer & FP pipeline ('architecture'=2 in XML file) and separate integer and FP pipelines ('architecture'=1 in XML file)

2.2.7. NoC

We modelled the NoC as a crossbar-based network that connects SM clusters to L2 banks. The SM-to-L2 traffic and the L2-to-SM traffic utilize different crossbar networks to avoid contention.

2.2.8. DRAM

The DRAM contributes a large portion of the GPU's overall dynamic power consumption. It has large impact in the benchmarks that perform a large number of DRAM accesses. However, our power model mainly focuses on the processor power modeling, and DRAM power modeling is required for full-system validation because the DRAM is integrated on the same PC board as the GPU.

DRAM Power = (Energy for Pre-charge * Pre-charge counts + Energy for Activation * Activation counts + Energy for Write * Write counts + Energy for Read * Read counts) / Execution Time

The above equation states the empirical DRAM dynamic power model we used. We consider its dynamic power to be composed of precharge power, row buffer activation power, read power, and write power. The precharge power is calculated as the per-operation energy times the number of precharge operations divided by the execution time. Read, write, and activation powers are calculated in the same way.

The above DRAM power model results in a linear system regarding to the performance counters of each operation. We treat the access energy for each DRAM operation as an unknown variable, and we solve these unknowns by constructing a suite of microbenchmark via Least Square Estimation (LSE).

2.3. Accuracy

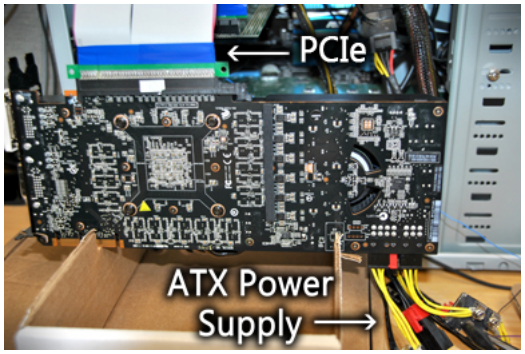

Figure 6: GPU Card

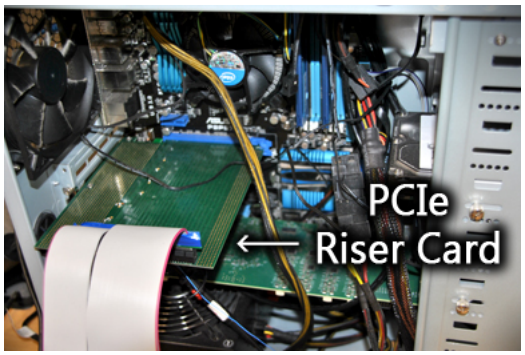

Figure 7: PCIe Riser Card

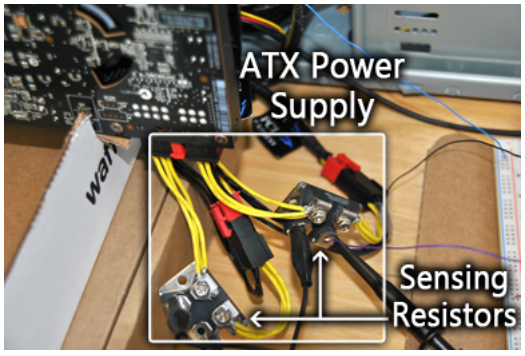

Figure 8: ATX Power Supply

Figure 9: DAQ

To improve the accuracy of our power model, we setup a hardware measurement unit such that we can compare the modeled power with the hardware power (shown in Figures 6 - 9). For the validation purpose, we used to cards with two different architectures: GTX 480 (Fermi) and Quadro FX 5600 (G80). Both the cards are connected to the PCIe slot through a PCIe riser card and powered by an ATX power supply (Figures 6 and 7). The PCIe riser card and the ATX power supply have power pins that deliver power to the GPU. For each power supply source, we measure the instantaneous current and voltage to compute power. We sense the current by measuring the voltage drop across a current sensing resistor (Figure 8). We use a National Instruments' DAQ to sample the voltage drop at a rate of 2 Million Samples/s (Figure 9). Their parameters for different components are shown in next table.

Component	GTX 480	Quadro FX 5600
Archiecture	Fermi	G80
CUDA Cores	480	120
Frequency	1.4GHz	1.2GHz
Register file	131KB	32KB
Shared memory	48KB	16KB
L1 cache	16KB	N/A
L2 cache	768KB	192KB
Texture Cache	12KB	5KB
Constant Cache	8KB	8KB
Technology node	40nm	65nm
Memory type	GDDR5	GDDR3
Memory bandwidth	177.3GB/s	76.8GB/s
Memory Controller	6	6

We constructed a suite of microbenchmarks which are carefully crafted for our power model refinement. We progressively evolve the complexity of the microbenchmarks, starting from basic microbenchmarks that access only specific microarchitectural components (i.e., functional, memory, DRAM microbenchmarks) to more complex heterogeneous microbenchmarks (Mix) that exercise combinations of instructions to concurrently stress the different microarchitectural components.

Name	Exercised Components	Counts
Functional	Integer, floating point, and special function unit	11
Memory	Instruction, L1 & L2, texture, constant caches, and shared memory	25
DRAM	DRAM	22
Mix	Mix of functional, DRAM and memory microbenchmarks	22

This suite microbenchmark ensures our power model capable of adapting to newer architectures. Depending on the extent of the differences in the target architecture from the existing power model, full or partial iterations of the refinement and validation loop may be required. In this process, components may need to be added or removed from the power and performance simulators. Activity counters may need to be added that capture the behavior of the modified architecture, and additional microbenchmarks may also be necessary to stress and refine the new components.

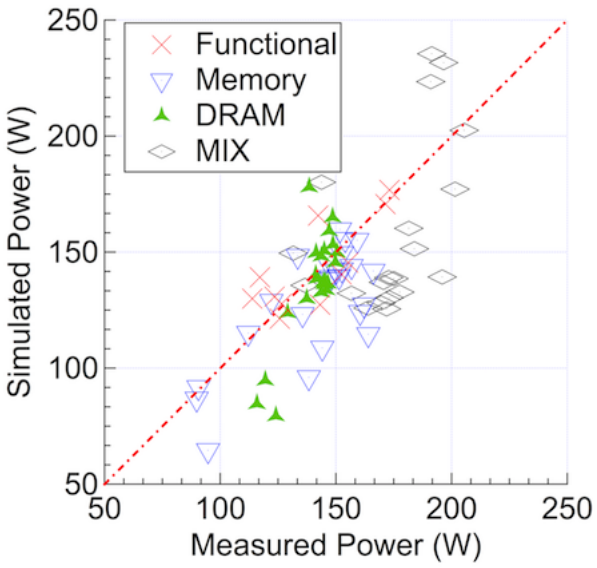

Figure 10: Power model accuracy of GTX480 over Microbenchmarks

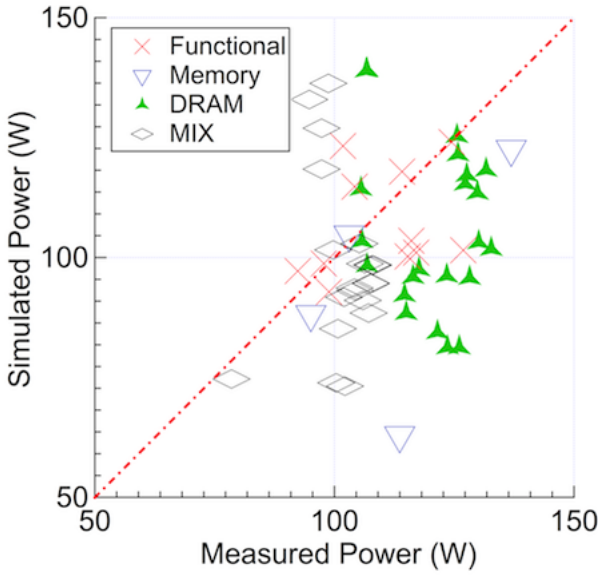

Figure 11: Power model accuracy of QuadroFX5600 over Microbenchmarks

The above figure shows the accuracy of our power model for our refinement microbenchmarks. In average, it achieves accuracy within 12.6% and 15.5% for GTX 480 and Quadro FX 5600, respectively.

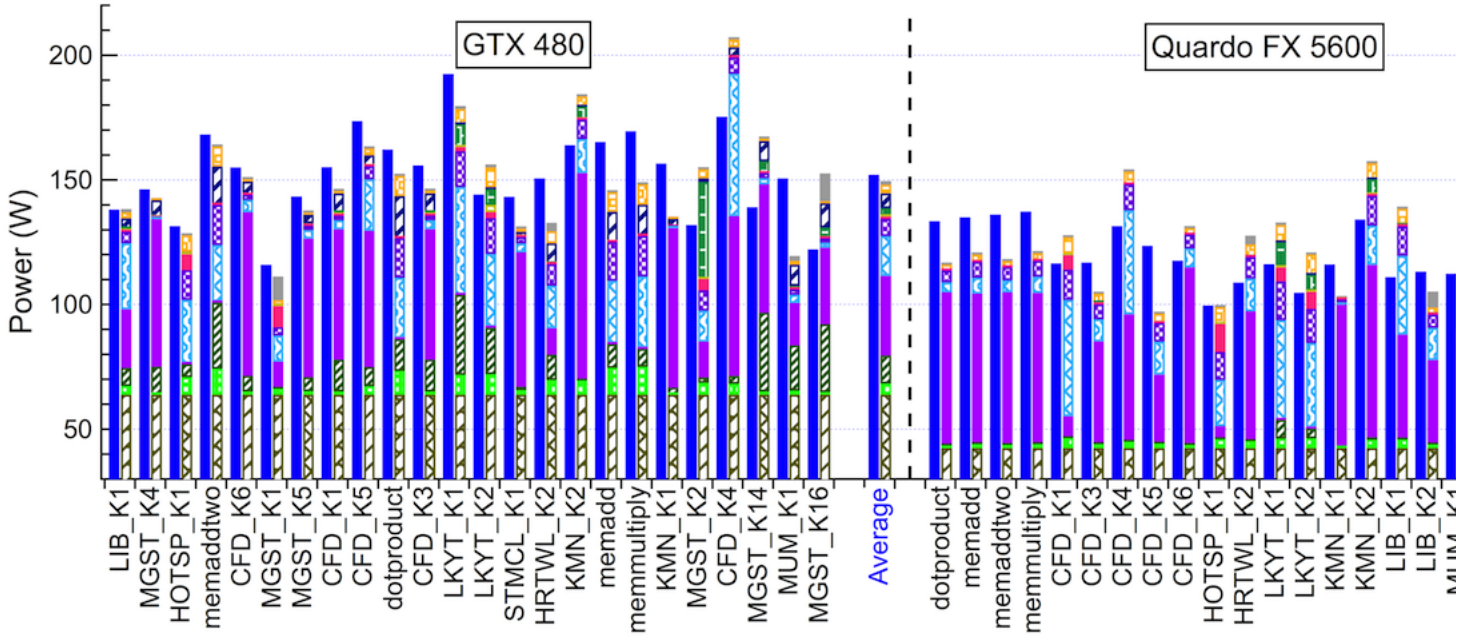

Figure 12: Power Model Accuracy for Some Rodinia and ISPASS Benchmarks

We also evaluated our power model for real benchmarks. We filter our kernels that have execution time less than 0.5ms because we found in our experiment the kernels with execution time less than that is always in the exponential increasing stage, the average power of which can not represent its actual power consumption. This execution time limitation still filters-out 56 unique kernels in the ISPASS and RODINIA benchmarks suites to the 21 kernels discussed in shown in the above graph. In addition, we study another 4 kernels from prior work by Hong and Kim. So in all we have 25 long-running kernels. The modeling error of our power model is 9.7% and 13.6% for GTX 480 and Quadro FX 5600, respectively.

3. Performance Counters Driving the Power Model

The power model can be simplified as the sum of the power components induced by the activity of GPGPU-Sim's performance counters. These performance counters are calculated within the GPGPU-Sim code and passed to the McPAT code to be used in the power calculation for each component.

3.1. Performance Counters List

Here we list the description of the performance counters used in developing the power model. However, it should be highlighted that changes in the architecture, or at a finer resolution to the power component breakdown, might imply the need for additional performance counters to correctly model the system. All the performance counters used to drive the power model are collected in a single class called "power_stats" (see Software Design of the Power Model). The abbreviations in the first column is used within the code and in the output files.

Index	Performance Counters	Description
1	I$-hits (IC_H)	Counts the number of instruction cache hits, counted per warp instruction

2	I$-misses (IC_M)	Counts the number of instruction cache misses, counted per warp instruction
3	D$-read hits (DC_RH)	Counts the number of data cache read hits, counted per memory access
4	D$-read misses (DC_RM)	Counts the number of data cache read misses, counted per memory access
5	D$-write hits (DC_WH)	Counts the number of data cache write hits, counted per memory access
6	D$-write misses (DC_WM)	Counts the number of data cache write misses, counted per memory access
7	T$-hits (TC_H)	Counts the number of texture cache hits, counted per memory access
8	T$-misses (TC_M)	Counts the number of texture cache misses, counted per memroy access
9	C$-hits (CC_H)	Counts the number of constant cache hits, counted per memory access
10	C$-misses (CC_M)	Counts the number of constant cache misses, counted per memory access
11	Shared Memory Accesses (SHRD_ACC)	Counts the number of shared memory accesses, counted per memory access
12	Register File Reads (REG_R)	Counts the number of register file reads in all instructions, counted per thread
13	Register File Writes (REG_W)	Counts the number of register file writes in all instructions, counted per thread
14	Non Register File Operands (NON_REG_OPs)	Counts the number of non register file operands (e.g. immediate), counted per thread
15	SFU accesses (SFU_ACC)	Counts the all instructions that exercise SFU pipeline (it also includes multiplications/division), counted per thread
16	SP accesses (SP_ACC)	Counts the all instructions that exercise SP pipeline with integer operands, counted per thread
17	FPU accesses (FPU_ACC)	Counts the all instructions that exercise SFU pipeline with floating-point operands, counted per thread
18	Total number of instructions (TOT_INST)	Counts the all decoded instructions, counted per warp
19	W/O operand instructions (FP_INT)	Counts the all instructions without operands (e.g., call/return)
20	DRAM reads (DRAM_RD)	Counts the dram read accesses, counted per memory access
21	DRAM writes (DRAM_WR)	Counts the dram writes accesses, counted per memory access
22	DRAM precharge (DRAM_PRE)	Counts the dram precharges accesses, counted per memory access
23	L2$-read hits (L2_RH)	Counts the number of L2 data cache read hits, counted per memory access
24	L2$-read misses (L2_RM)	Counts the number of L2 data cache read misses, counted per memory access
25	L2$-write hits (L2_WH)	Counts the number of L2 data cache write hits, counted per memory access
26	L2$-write misses (L2_WM)	Counts the number of L2 data cache write misses, counted per memory access
27	Pipeline Duty Cycle (PIPE)	Ratio of committed number of instructions to the maximum peak of committed instructions, counted per thread
28	Interconnect flit SIMT-to-Mem (NOC_A)	Counts the number of flits traveling from SIMT cluster to memory partition
29	Interconnect flit Mem-to-SIMT (NOC_A)	Counts the number of flits traveling from memory partition to SIMT cluster
30	Idle Core (IDLE_CORE_N)	Counts the average number of idle cores over cycles of each sample

4. Using the Power Model

This section describes how to compile the power model with GPGPU-Sim, configure the power model, and understand the power model's output.

4.1. Compiling GPUWattch

By default, GPGPU-Sim bypasses the GPUWattch compilation. Hence, it is not necessary to install our modified McPAT along with GPGPU-Sim. However, if the power model is required, our modified version of McPAT will be compiled alongside GPGPU-Sim into *libcudart.so* .

The location of McPAT is specified by the GPGPUSIM_POWER_MODEL environment variable and is used in GPGPU-Sim's Makefile to compile McPAT. This can either be set manually or automatically set by the *setup_evironment.sh* file in *v3.x/* if McPAT is found in the default directory (/v3.x/src/gpuwattch).

4.2. Configuration Options

This sections lists the most relevant configuration options either to describe model a specific architecture, model a certain improvement, or to produce extra data from the simulation. Some of these configurations are set in the GPGPU-Sim config files and others will be set in the GPUWattch/McPAT XML configuration files. GPGPU-Sim configurations mainly force specific changes to the performance modeling itself or the output format of data, while McPAT configurations mainly determine specific parameters for the different architectural components that are reflected in the power modeling of these components. Here, we will only mention the most relevent and/or GPU related parameters in the GPUWattch/McPAT XML configuration.

Index	GPGPUsim Configuration Option	Description
1	power_simulation_enabled	Enable the power model simulator; if enabled, an output file is genarated to include the detailed Power coefficients for the simulated configuration and the Average/Maximum/Minimum total power breakdowns for each kernel
2	gpuwattch_xml_file	The GPUWattch (McPAT) XML configuration file name; by default it is gpuwattch.xml. For the GTX480 configuration in gpgpusim.config, this is set to gpuwattch_gtx480.xml
3	gpu_stat_sample_frequency	Determines the sampling frequency (in number of GPGPU-Sim core cycles) used in the power calculations, the performance counters are reset before each samples and accumulated during the sampling period, and finaly passed to the power model (McPAT) at the end of each sample
4	power_trace_enabled	If enabled, it produces two output files that details the power breakdown values, and the accumulative performance counters values for each sample
5	power_per_cycle_dump	Dump detailed power data each sample
6	steady_power_levels_enabled	If enabled, it tracks the steady state power level throughout the execution and report the start/end values with the average power recorded for each component. The steady state is determined by (-steady_state_definition) option
7	steady_state_definition	Takes two values. First value detemines the allowed deviation within the steady state and the second value determines minimum number of samples required to assume this is a steady state power level

The following table is explanation of the GPUWattch/McPAT XML file configuration:

Index	XML Configuration Option	GTX 480 Value	Description
1	architecture	1	Fermi: 1, Quadro: 2
2	number_of_cores	16	Number of SMs
3	number_of_L2s	1	Number of L2 cache
4	number_of_L3s	0	Number of L3 cache
5	Number_of_NoCs	1	Number of network on chip

6	homogeneous_cores	1	1 means all cores are the same 0 means not the same
7	core_tech_node	40	Technology in nm
8	target_core_clockrate	700	Target core clock frequency in MHz
9	Temperature	380	Processor temperature in Kelvin
10	number_cache_levels	2	Number of cache levels
11	interconnect_project_type	0	0 means aggressive wire technology, 1 means conservative wire technology
12	device_type	0	0: High performance type 1: Low standby power 2: Low operating power
13	longer_channel_device	1	0: no use 1: use when possible
14	idle_core_power	159	Power consumed by an idle SM
15	number_hardware_threads	32	Hardware thread is similar to warp in GPU This means the maximum possible warps per SM
16	machine_type	1	1: in-order 0: out of order
17	issue_width	2	Maxim number of instructions can be issued to execution unit
18	pipeline_per_core	1,1	Integer pipeline and floating point pipeline If the FP pipeline is zero, it is shared with integer pipeline
19	ALU_per_core	32	Interpreted as SIMD ALU width in GPU
20	MUL_per_core	4	Interpreted as SIMD special function unit width in GPU
21	FPU_per_core	32	Interpreted as SIMD FPU width in GPU
22	rf_banks	32	
23	simd_width	32	
24	collector_units	32	
25	core_clock_ratio	2	
26	warp_size	32	
27	archi_Regs_IRF_size	32768	Number of registers per SM
28	archi_Regs_FRF_size	32	Original McPAT treats this as floating point registers GPU has unified reigster for integer and floating point
29	memory_ports	2	Number of read/write port for each cache
30	ccache_config	16384,32,4,1,1,3,8,0	Cache configuration: Capacity, block width, associativity, bank, throughput w.r.t. core clock, latency w.r.t. core clock, output width, cache policy
31	tcache_config	16384,32,4,1,1,3,8,0	
32	sharedmemory	49152,16,1,16,1,3,16,0	
33	dcache_config	16384,32,4,1,1,3,8,0	
34	L2_config	131072,256,8,1,4,23,64,1	
35	horizontal_nodes	2	NoC nodes configuration
36	vertical_nodes	1	

4.3. Understanding Simulation Output

In this section, we detail the fromat of the power simulator output. By default, if the power simulation is enabled at least one output file that reports the average/maximum/minimum power values for each kernel is produced. More outputs can be enabled if the corresponding configurations is enabled.

Index	Output File name	Configuration	Description
1	gpgpusim_power_report_(date&time).log	-power_simulation_enabled 1	Includes the detailed power coefficients for this configuration and the Average/Maximum/Minimum total power and their breakdowns for the different components for each kernel
2	gpgpusim_power_trace_(date&time).log.gz	-power_trace_enabled 1	A compressed file that has a detailed average power breakdown trace in a comma separated format
3	gpgpusim_metric_trace_(date&time).log.gz	-power_trace_enabled 1	A compressed file that has a detailed performance counters trace in a comma separated format
4	gpgpusim_steady_state_tracking_report_(date&time).log.gz	-steady_power_levels_enabled 1	It reports the steady state power level throughout the execution with the start/end values of each interval and the average power recorded for each component during this interval in a comma separated format

4.4. Example

- Copy the configuration files from v3.x/configs/GTX480/ to your working directory (config_fermi_islip.icnt, gpgpusim.config, gpuwattch_gtx480.xml)
- Add or set the following configuration options to **gpgpusim.config**

```
-power_simulation_enabled 1 # Enable power model
-gpuwattch_xml_file gpuwattch_gtx480.xml # Default is "gpuwattch.xml", only necessary if not the default
-power_trace_enabled 1 # Enable output: detailed average power traces
-steady_power_levels_enabled 1  # Enable output: steady state average power levels and corresponding performance counters
```

- Run your application
- This will produce the following output files

```
gpgpusim_power_report_.log
gpgpusim_power_trace_report_.log.gz
gpgpusim_metric_trace_report_.log.gz
gpgpusim_steady_state_tracking_report_.log.gz
```

- Decompress the necessary output files (gzip -d < filename >.log.gz)

Sample of Output files

==========
gpgpusim_power_report_.log
==========

This file shows the Average, Max, and Min power stats for each component, as well as the corresponding performance counters use to generate these values.

Example of average breakdown:
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
...

The "P" at the end signifies power (in Watts), whereas no "P" signifies the counter value. I.e., gpu_avg_IBP represents the average instruction buffer power, whereas gpu_avg_IC_H and gpu_avg_IC_M represents the instruction cache Hit and Miss performance counters respectively that were recorded for this average power value.

After the detailed breakdown, the total stats accumulated over all previous kernels are reported:

Example:
Accumulative Power Statistics Over Previous Kernels:
gpu_tot_avg_power = 41.6007
gpu_tot_max_power = 47.968
gpu_tot_min_power = 13.6755

This format is then repeated for each kernel in the application.

==========
gpgpusim_power_trace_report_.log.gz
==========

This file shows a detailed dynamic power breakdown (in Watts) of each component per sampling period (default 500 cycles) in a CSV format.

Example:
power,IBP,ICP,DCP,TCP,CCP,SHRDP,RFP,SPP,SFUP,FPUP,SCHEDP,L2CP,MCP,NOCP,DRAMP,PIPEP,IDLE_COREP,CONST_DYNAMICP
22.499146,0.245070,0.275922,0.000000,0.000000,0.266508,0.000000,0.043781,0.000000,0.000000,0.000000,0.000738,0.117159,1.242679,0.049753,0.278432,0.000000,1
38.357532,1.272785,0.707513,0.133254,0.000000,1.045434,0.000000,2.088517,0.021102,0.000000,0.000000,0.003642,0.229445,1.365898,0.092398,0.453405,10.990476,
...
The first line displays the name of each component (power=cumulative power, IBP=instruction buffer power, DCP=data cache power, etc...). The following lines show the dynamic power breakdown (in Watts) of each component. The sum of all components equals the first value (power).

==========
gpgpusim_metric_trace_report_.log.gz
==========

This file shows a detailed metric breakdown (performance counter) of each component per sampling period (default 500 cycles) in a CSV format.

TOT_INST,FP_INT,IC_H,IC_M,DC_RH,DC_RM,DC_WH,DC_WM,TC_H,TC_M,CC_H,CC_M,SHRD_ACC,REG_RD,REG_WR,NON_REG_OPs,SP_ACC,SFU_ACC,FP
496.000000,496.000000,248.000000,128.000000,0.000000,0.000000,0.000000,0.000000,0.000000,0.000000,0.000000,0.000000,256.000000,0.000000,0.000000,0.000000,8704.000
2576.000000,2448.000000,1288.000000,128.000000,0.000000,128.000000,0.000000,0.000000,0.000000,0.000000,640.000000,0.000000,0.000000,61440.000000,61184.00
...

The first line displays the name of each component (TOT_INST=total decoded instructions, IC_H=instruction cache hit, REG_WR=register file writes, etc...). The following lines show the performance counter values per sampling period (e.g., the first line shows 496 total instructions decoded, 248 instruction cache hits, 128 instruction cache misses, etc). These performance counters were used to generate the power values in the **gpgpusim_power_trace_report_.log.gz** output file. This file is useful when wanting to calculate the power offline using the coefficients provided by McPAT.

==========
gpgpusim_steady_state_tracking_report_.log.gz
==========

This file shows the steady state regions in dynamic power during each kernel as well as the per-component average dynamic power breakdown during that steady state region.

*start,end,power,IPC,TOT_INST,FP_INT,IC_H,IC_M,DC_RH,DC_RM,DC_WH,DC_WM,TC_H,TC_M,CC_H,CC_M,SHRD_ACC,REG_RD,REG_WR,NON_REG_OPs,SP_*
4,12,47.697173,0.063750,31.750000,28.000000,15.875000,0.000000,4.750000,1.250000,0.000000,2.500000,0.000000,0.000000,0.000000,0.000000,0.000000,1132.00000
16,22,35.842763,0.587667,284.000000,249.333333,142.000000,0.000000,42.500000,12.333333,0.000000,19.666667,0.000000,0.000000,0.000000,0.000000,0.000000,995

The first line displays the start and end sampling number (e.g., The first steady state region, *(4,12)*, starts at sample 4 and finishes at sample 12. Here each sampling period is the default 500 cycles). The steady state region is defined by a configurable threshold constant that specifies the maximum amount the dynamic power can vary while still being considered steady (e.g., ±5 Watts). The 3rd column is the steady state average dynamic power the (start, end) region. The remaining columns show the per-component average dynamic power breakdown during the steady state (start, end). The sum of these components equals the *power* term.

To be a steady state, a configurable number of steady state values must be recoded (i.e., 5 samples not exceeding the threshold). If not enough samples were recorded, "ERROR! Not enough steady state points to generate average" is printed.

5. Software Design of the Power Model

This section presents the software design of the power model.

5.1. File List and Brief Description

In this section, we briefly describe the files that were added to interface GPGPU-Sim with McPAT.

Index	File Name	Directory	Description
1	power_stat.cc/h	src/gpgpu-sim/	These files contain the main structures used for recording GPGPU-Sim performance counters: *power_core_stat_t* (for all core related counters) and *power_mem_stat_t* (for all memory related counters), which are contained in the wrapper *power_stat_t* object. The *core* and *mem* stat structures contain multiple counter pointer arrays with 2 locations per counter (e.g. unsigned *m_counter[2]): [0] -> pointer to counter with the current value, [1] -> previous sampled value. The difference, [0]-[1], is used to get the per-sample estimated power in McPAT.
2	gpgpu_sim_wrapper.cc/h	src/gpuwattch/	These files contain the *gpgpu_sim_wrapper* class that contains all of the McPAT structures (such as Processor, ParseXML, etc), manages the power output files, and passes the GPGPU-Sim performance counters (described in power_stat.cc/h (1)) into McPAT. The gpgpu_sim_wrapper structure is used in *power_interface.cc/h (3)* to separate the McPAT structures and interface from GPGPU-Sim.
3	power_interface.cc/h	src/gpgpu-sim/	These files are used to interface GPGPU-Sim with McPAT via two main functions: *init_mcpat()* and *mcpat_cycle()* . *init_mcpat()* is called from gpgpu_sim::init() in gpu-sim.cc and through the *gpgpu_sim_wrapper* object, initializes all of the power related structures in GPGPU-Sim and McPAT. Similarly, *mcpat_cycle()* is called from gpgpu_sim::cycle() in gpu-sim.cc, which passes all of the performance counters to McPAT (through the *gpgpu_sim_wrapper* object).
4	gpgpu_sim.verify	src/gpuwattch/	This file is distributed with our modified version of McPAT to ensure the correct McPAT version is used with GPGPU-Sim.

6. Support and Bug Reports

If you wish to report a bug for GPUWattch, please submit a bug report here and specify *GPUWattch* under the Component section.