D4.4.2 The GPUSimPow Simulator

Seventh Framework Programme

Document Information

No. & Title of Deliverable : D4.4.2 The GPUSimPow simulator

Grant Agreement No. : 288653

Project Website : lpgpu.org

Date : 31st July 2012

Delivery Date : 31st August 2012

Nature : P

Authors : Sohan Lal (TUB), Jan Lucas (TUB), Michael Andersch (TUB), Mauricio Alvarez-Mesa (TUB), Ben Juurlink (TUB), Paul Keir (Codeplay)

Contributors : Vasileios Spiliopoulos (Uppsala); Georgios Keramidas (ThinkSilicon), Björn Knafla (Codeplay)

Reviewers : Konstantinos Koukos (Uppsala); Georgios Keramidas (ThinkSilicon)

Acknowledgement

The research leading to these results has received funding from the EU's Seventh Framework Programme (FP7/2007-2013) under Grant Agreement No. 288653.

Executive Summary

In this task, we developed a power simulation framework named GPUSimPow that is able to accurately estimate area, peak dynamic, and static power for a given GPGPU architecture as well as runtime dynamic power for GPGPU workloads written in CUDA or OpenCL. For GPUSimPow, we combine a cycle-accurate architectural simulator, GPGPU-Sim, with an extension of the McPAT power modeling framework, GPGPU-Pow. The power model used in GPGPU-Pow consists of both analytical models created using CACTI6.5 as well as empirical models derived from industry data and our own measurements on real hardware. To perform these measurements, we developed a custom testbed which allows hardware and software developers to measure power on real GPU cards at the granularity of individual GPGPU kernels. Besides the creation of empirical models, we also use the measurement testbed to measure power during execution of a set of CUDA SDK benchmarks. We use these results to validate against a simulation of the same benchmarks using GPUSimPow and show that absolute runtime power simulation accuracy is within 11% and relative runtime power simulation accuracy is within 10% of the real hardware power consumption, thus fulfilling the requirements for this task.

1. Description of Task

There are various power/energy models available for GPUs. None of them, however, suits our needs because they are either proprietary, too high level, and/or do not allow novel architectural techniques to be integrated.

Therefore, in this task, we will integrate a power model in GPGPU-Sim, which has become the de facto standard simulator for GPGPU research.

Power has been integrated in GPGPU-Sim in two ways. For operations (especially floating point operations since they are most common) we will develop kernels that execute many of the operations in question and measure the energy consumption of the kernel. The energy per operation is then obtained by dividing the total energy consumption by the number of operations. For memory structures and accesses we will use CACTI and distinguish between the different types of memory accesses (local vs. global and coalesced vs. uncoalesced). The model will be validated using workloads with different characteristics by measuring the current drawn on a test board. The model will be used to validate the high-level macro-based simulator developed in T4.3.Furthermore, ThinkS will work together with TUB and Uppsala in order to verify and improve the accuracy of the GPUSimPow simulator (in terms of power and performance). This feedback step will increase the value of GPUSimPow, since direct comparisons with the real hardware results will be performed.This task is a joint effort of TUB, ThinkS, Codeplay and ThinkS. The development will be done by TUB while Uppsala will assist the development by providing their experience in developing power models. ThinkS will aid the validation of the simulator by providing their hardware experience and Codeplay will contribute by validating the developed power simulator against the high-level macro-based simulator developed in T4.3.

2. Relevance of the Task to the Project

This task develops “GPUSimPow”, a GPU simulator which not only estimates the performance but also the power usage of GPUs while executing kernels. The simulator will be useful for many other project tasks. Overall, there are two categories of tasks that will use GPUSimPow: Tasks which try to optimize power consumption by manipulating and improving software, and tasks which tackle GPU power consumption from a hardware, architectural point of view. In both cases, GPUSimPow enables the associated research because the techniques being developed can be evaluated without the necessity of taking measurements using real hardware. However, the requirements for software-focused tasks are slightly different from the requirements for hardware-focused tasks: For the former, the simulator must provide high *relative* power simulation accuracy regarding the workloads, i.e. the changes in real power consumption caused by modifications to a given piece of software must be accurately modeled by the simulator. For the latter type of tasks, the simulator must provide high relative accuracy as well, but with regard to architectural modifications, e.g. if doubling the SIMD width doubles the (peak/runtime) power consumption of a real GPU, this must be true for the simulator as well. It is only through this requirement that the resulting power simulator enables useful design space exploration for GPU architects.

The central role of this task in the project is also exemplified by the following output dependencies:

- Task 4.4: OpenCL-to-GPUSim system
- Task 5.1: Design and Implementation of a SIMD-MIMD GPU Architecture (This task will also involve making power and area estimations of SIMD and MIMD processor cores.)
- Task 5.2: Redundancy (The architecture will also be implemented as part of the simulator in WP4.)
- Task 5.3: Slack
- Task 5.4: Accuracy
- Task 6.1: Evaluation of OpenCL Applications, Iteration and Optimization of Tools
- Task 7.1: Evaluation and Optimization Techniques to exploit SLACK in GPUs
- Task 7.2: Evaluation and Optimization Techniques to exploit REDUNDANCY in GPUs
- Task 7.3: Evaluation and Optimization of the SIMD-MIMD Architecture

3. Work Performed

The overall goal of this activity was to develop a cycle-accurate power simulator for GPU architectures and validate this simulator using measurements of real programs on real hardware. The simulator is meant to simulate real-world GPGPU programs and deliver runtime power numbers for such programs as well as general peak dynamic and static power and area numbers for the simulated architecture. In order to achieve this, the simulator must be flexible enough to allow for experimentation in order to explore the architectural design space of low-power GPU architectures.

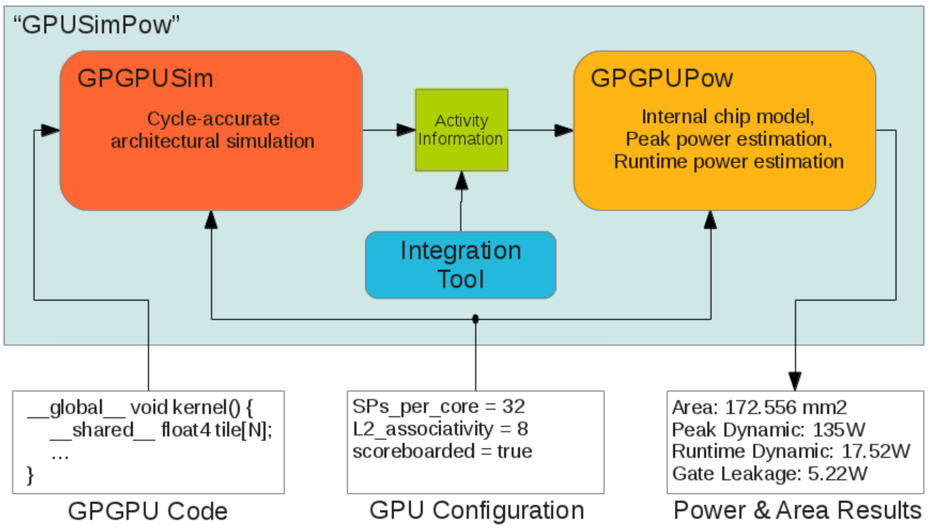
The approach in this work is described in the following sections and can be summarized as follows: We developed a power-simulation framework GPGPUSimPow (Section 3.1), consisting of a top-tier power modeling tool, McPAT (Section 3.2) and the most recent and detailed cycle-accurate GPGPU simulator, GPGPU-sim (Section 3.3) combined using our own tools and validated measurements on real hardware (Section 3.4).

3.1 Power-Simulation Framework

Generally, the power dissipation of CMOS circuits can be described by three main components: Dynamic, short-circuit, and leakage power. In modern multicores fabricated using deep-submicron technology nodes, leakage and dynamic power contribute significantly to the total power consumption of the chip, which is thus given as follows:

$$P_{\text{total}} = \alpha C V_{dd}^2 f_{\text{clk}} + V_{dd} I + V_{dd} I_{\text{leakage}} \;=\; \text{Dynamic} + \text{Short Circuit} + \text{Leakage}$$

is the fraction of total circuit capacitance being charged during a clock cycle and often referred to as *activity factor* of the circuit. In order to simulate runtime dynamic power of the GPU, we therefore must estimate an activity factor for each of the individual components inside the chip, since not all hardware capacities are equally active during execution. The total dynamic power draw is then equal to the sum of the power consumption numbers of the individual components.



```
__global__ void kernel() {
    __shared__ float4 tile[N];
    ...
}
```

```
SPs_per_core = 32
L2_associativity = 8
scoreboarded = true
```

```
Area: 172.556 mm2
Peak Dynamic: 135W
Runtime Dynamic: 17.52W
Gate Leakage: 5.22W
```

Figure 1: GPUSimPow framework

We developed a power simulation framework, called GPUSimPow, that combines the cycle-accurate simulator GPGPUSim with our power modeling tool, GPGPUPow, using a simple python integration tool. An overview is given in Figure 1_. As the figure shows, the only required inputs are the architectural configuration of the GPU as well as a GPGPU workload to execute. The framework will simulate the workload using the cycle-accurate simulator, thereby producing the activity information for all relevant hardware components. The framework will then convert the activity information into GPGPUPow format, and run the power simulation to generate numbers for peak dynamic, leakage, gate leakage, and runtime dynamic power as well as chip area.

3.1.1 OffloadCL Compiler Compatibility

The compiler technology developed by Codeplay as part of Task 4.1 must cooperate with GPGPU-Sim and GPUSimPow, in preparation for the impending Task 4.2. Specifically, Codeplay's single-source, extended C++ compiler, OffloadCL, must provide input compatible with the GPUSimPow toolchain. OffloadCL extends C++ in a number of ways, including a data parallel compound statement; known as an Offload block. Each Offload block is translated into an OpenCL C kernel; while the remaining host code, including newly created glue code, is lowered into the C language.

GPGPU-Sim, and GPUSimPow, are Linux applications; while our OffloadCL compiler runs under Windows. As we generate intermediate C and OpenCL C source files, we initially tested basic GPGPU-Sim compatibility with OpenCL programs using examples from the NVIDIA CUDA SDK. Having reasonable compatibilty, we consequently, developed an OS-portable version of the OffloadCL runtime library. By running the OffloadCL Windows executable under the Wine emulator, we were then able to compile OffloadCL C++ programs, and pass the output through GPGPU-Sim. Each modification of a Codeplay product requires extensive testing. In this case our in-house testing system, Testplay, itself needed some work to modify it for the new Linux test environment required.

3.2 Power modeling

For architecture-level power modeling, we chose McPAT (Multi-Core Power, Area, and Timing), a tool originally published by HP[LAS+ 09]. McPAT is an integrated and hierarchical power, area, and timing modeling framework that supports comprehensive design space exploration for multicore and manycore processor configurations ranging from 90nm to 22nm and even further. It is integrated in the sense that it analyzes power, area, and timing in a tightly coupled way that allows computer architects to explore the design space using metrics such as energy-delay-area-products. McPATs hierarchical nature allows to decouple the views of the architectural, circuit, and technological levels. With this decoupling, architects can easily examine low-level device technology on the one hand, and switch to higher abstraction levels with architectural-level configurations on the other hand.

Internally, McPAT generates a chip representation which models processor components in one of two ways:

1. Regular components such as caches or register files are modeled analytically using CACTI 6.5[TAM+ 08].
2. Highly customized components and logic blocks such as functional units (FU) are modeled empirically from measurement and industry data.

To generate power numbers for a run of a particular program, McPAT must cooperate with an architectural cycle-accurate simulator. McPAT relies on this simulator to get activity factors / access counts for the individual hardware strIuctures. Using these activity factors, McPAT then computes power numbers for the simulated architecture and executed program.

Altogether, McPAT can be regarded as an extended version of CACTI that is capable of modeling not only regular hardware structures, but the entire processor. Unfortunately, McPAT was designed for modeling contemporary CPUs only. For this reason, on the one hand, McPAT contains a variety of structures and components that are irrelevant for GPU architecture modeling, such as branch predictors, renamers, or wide-fetch frontends. On the other hand, many of the specific architectural features of modern GPUs are not present in McPAT, such as reconvergence stacks, SIMD-style ALU bundles, or memory access coalescing logic.

Consequently, in this task, we have developed GPGPUPow, a tool to model GPU area and power based on McPAT. Similar to McPAT's approach, we model regular structures such as register files and shared memory using analytic CACTI descriptions. For irregular structures, we employ an approach based on a mixture of our own measurements using real hardware and industry data. As McPAT, GPGPUPow provides power numbers for peak dynamic, runtime dynamic, leakage, and gate leakage powers as well as area numbers. Given the particular cycle-accurate simulator we use to supply McPAT with activity factors (see next section), the GPU architecture we model is roughly similar to the recent NVIDIA Fermi/Kepler[LOS09][NVI12] and AMD GCN[AMD12] architectures. This way, we are able to assert that the architecture model in GPGPUPow is sufficiently similar to the one simulated by the cycle-accurate simulator, introducing fewer modeling errors.

Following is a list of components that have been added to McPAT to develop GPGPUPow:

- Warp Control Unit (Warp Status Table, Instruction Buffers, Reconvergence Stacks, Scoreboarding Logic, Instruction Decode Logic, Schedulers)
- GPU-style Register File
- Execution Units (INT, FP32, SFU)
- Load-Store Unit (Coalescer, Bank Conflict Checker, AGU Array, Per-Core Constant Cache Slice, Shared Memory, L2 Cache)
- GDDR

A few general components could be reused from McPAT's models for the GPU architecture model:

- Memory Controller
- PCIe-Controller
- NoC

In the following, a brief overview over the architectural models for each of these components is given.

3.2.1 Warp Control Unit

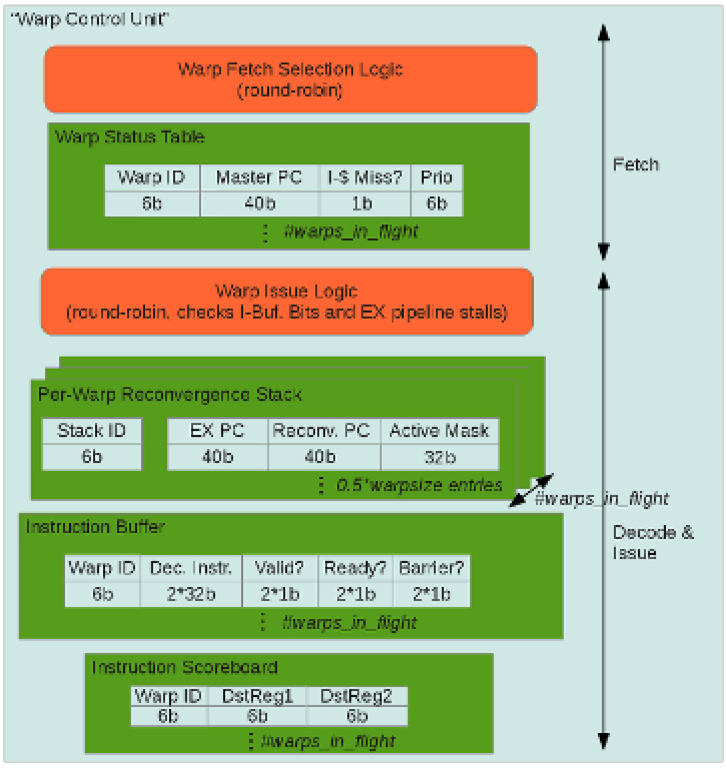

"Warp Control Unit"
Warp Fetch Selection Logic (round-robin)
Warp Status Table
Warp ID 6b | Master PC 40b | I-$ Miss? 1b | Prio 6b
#warps_in_flight
Fetch
Warp Issue Logic (round-robin, checks I-Buf. Bits and EX pipeline stalls)
Per-Warp Reconvergence Stack
Stack ID 6b | EX PC 40b | Reconv. PC 40b | Active Mask 32b
0.5*warpsize entries
#warps_in_flight
Decode & Issue
Instruction Buffer
Warp ID 6b | Dec. Instr. 2*32b | Valid? 2*1b | Ready? 2*1b | Barrier? 2*1b
#warps_in_flight
Instruction Scoreboard
Warp ID 6b | DstReg1 6b | DstReg2 6b
#warps_in_flight

Figure 2: Warp control unit

The Warp Control Unit (WCU) represents the front-end of a single shader core, from here on just referred to as core. As such, the WCU is responsible for keeping the execution back-end, i.e. the functional units and the load-store-hardware, supplied with instructions at all times. Therefore, the WCU handles thread management (i.e. formation of warps from threads and the relation of per-thread control flow under warp constraints), warp scheduling, warp instruction fetching, decoding, dependency resolution, and renaming. An example of the structures contained in the WCU, together with hypothetical storage sizes, is shown in Figure 2.

The information needed for each warp to fetch instructions and manage the warp threads is contained in a single RAM table, the Warp Status Table (WST). The WST contains one entry for each in-flight warp the core can handle. Each entry consists of a warp global ID, a warp master PC, bit information whether the warp is allowed to fetch instructions, and a priority (optional) for scheduling. The WST table is multiported to have as many read and write ports as the core has warp schedulers so each scheduler can modify the WST independently.

To select a warp to fetch an instruction for, a rotating-priority (round-robin) warp scheduler is modeled. Such schedulers consist of a set of inverters, a wide priority encoder, and a phase counter. These components have been modeled from appropriate circuit plans[KQM04] using McPAT's circuit and technology layers.

As is common for most GPGPU applications on modern GPU architectures, the individual in-flight threads often execute different dynamic instruction paths. The grouping of threads into SIMD bundles (warps) implicitly forces the thread PCs to have the same value at all times, however. If this is not the case due to the threads branching into different dynamic execution paths, the execution of threads in a single warp but with different PCs is serialized. Different branch directions are executed after each other, until all execution path within the bundle have been executed. A active mask is used to enable only the execution of instructions from threads that share the currently executed control flow path. To achieve this serialization and keep track of the thread IDs that have to execute certain branch outcomes, the hardware uses a stack memory called the reconvergence stack[US7543136]. For each individual in-flight warp, the hardware maintains a separate stack. In our model, a stack consists of tokens, each of which contains an execution PC, a reconvergence PC, and an active mask for that warp and code block. The number of tokens on the stack, i.e. the stack depth is limited to half the number of threads in a warp. While theoretically fully divergent behaviour with each warp thread taking a different dynamic program path is possible and would require a stack as deep as the warp size, such behaviour is extremely unlikely[FA11]. Therefore, we assume that in such a case the overflowing entries are spilled to memory. We also assume that buffering and arbitration of access to the reconvergence stacks can be used and was used to create a design that uses a single read and write port for the reconvergence stacks. For this reason, the stacks are modeled with a single read/write port.

After instructions have been fetched from the I-Cache, they are decoded. For this, we re-use the instruction decoder hardware models already present in McPAT.

Once an instruction has been decoded, the WCU places the instruction into an instruction buffer (IB) slot. The instruction resides in its buffer slot until it is ready to execute, that is, if its register dependencies have been resolved (scoreboarded architectures) or the previous instruction from the same warp has been committed (blocking barrel-processing architectures). The instruction buffer is a cache-like structure that is tagged by the warp ID and has an associativity greater than one, i.e. each instruction can be buffered in one of several slots tagged by its parent warp ID. The number of “sets” is equal to the number of in-flight warps in the core. The IB has a single read port per scheduler and a single write port per scheduler. Write-back from later pipeline stages, e.g. for warp barriering, is assumed to be carried out using extra state DFFs which are negligible size-wise and have therefore not been modeled.

For resolving register dependencies, GPUs (e.g. Fermi) use simple approaches based on scoreboarding[US7434032]. In our models, a scoreboard is a cache-like table tagged by the warp ID. For each cache set representing a warp, the associativity is equal to the maximum number of instructions simultaneously in-flight for this warp. A single scoreboard entry consists of state bits, i.e. if the entry is valid, as well as if the designators of destination registers of a warp instruction currently in the pipeline back-end. This way, for an incoming instruction, the scoreboard allows checking if the instruction's source registers match with the destination registers in the scoreboard entry, and if so, the instruction must wait in the instruction buffer until the scoreboard entry is cleared. The scoreboard has one read and one write port per warp scheduler.

3.2.2 GPU-style Register File

The GPU Register File modeling is based on an NVIDIA Patent[US7339592]. The GPU Register File is built from multiple single ported RAM banks. Operands are collected over multiple cycles simulating a multi-ported register file. Different threads will have their registers stored in different banks. This scheme increases the area density of the register file. Full throughput, however, can only be achieved if the scheduler is always able to interleave register file accesses from different banks. A crossbar is used to connect the different register banks to a set of operand collector units which are two-ported four entry register files and some status bits for the scheduler.

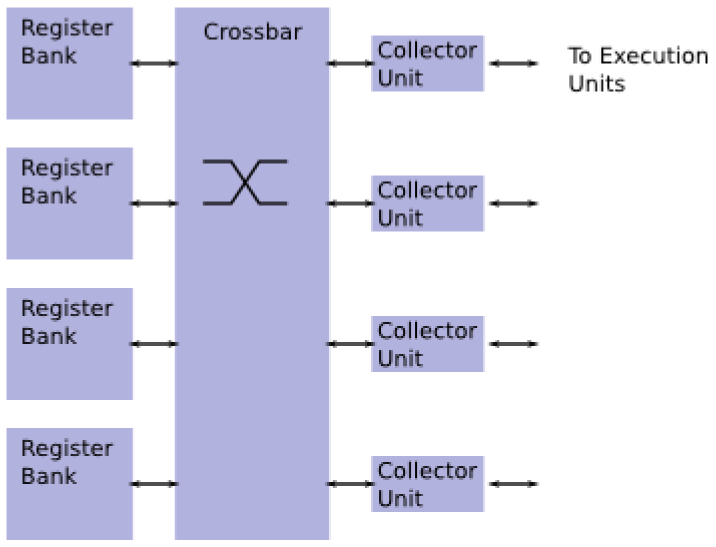


Figure 3: GPU Register File

3.2.3 Execution Units

The basic unit of execution flow in the Streaming Multiprocessor(SM) is the warp. In the SM, a warp is a collection of 32 threads and is executed in lockstep on eight Scalar Processors (SP). NVIDIA refers to this arrangement as Single-Instruction Multiple-Thread (SIMT), where every thread of a warp executes the same instruction in lockstep, but allows each thread to branch separately. The GPU has a set of SIMD execution units which execute the SIMT threads in lock step. For example, the NVIDIA GT240 has 12 SIMT cores. Each SIMT core has eight fully pipelined floating point units and two special function units (SFUs). The SFUs execute transcendental instructions such as sine, cosine,reciprocal, and square root. In our power model, we used the energy and area numbers published in the energy efficient floating point unit design by Sameh et al.[GH11]. Caro et al.[DCPS08] implemented high performance floating point special function unit and presented energy and area numbers for 0.18µm CMOS at 420 MHZ. SFU consumes 160mW @ 420 MHZ and has 0.340 mm^2 area The SFU has 6 cycles latency and 1 cycle throughput. We used their results with scaling for 40nm process technology. We also provided a measurement-based model for these execution units. This model is based on a micro-benchmark measurements on a GT240 and is scaled for process technology and frequency.

3.2.4 Load-Store Unit

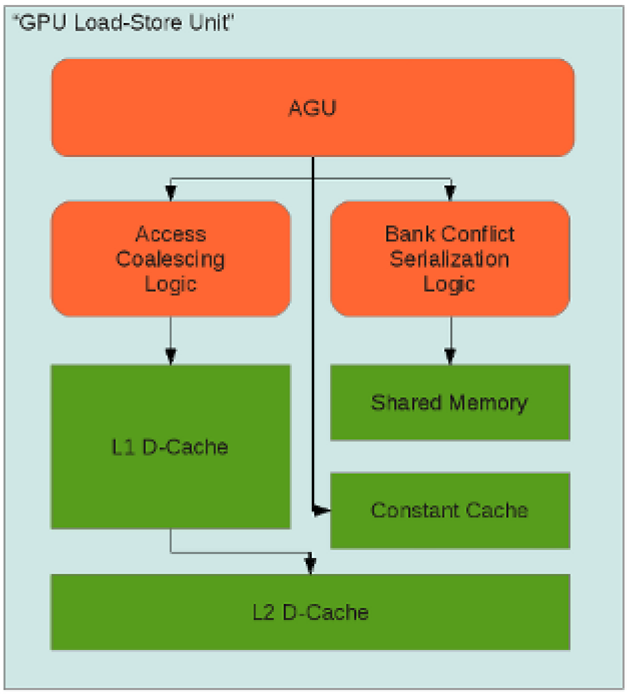


Figure 4: Load-Store Unit

The load-store unit (LDSTU) is functionally responsible for handling instructions that read or write any kind of memory. In the power model, the LDSTU encapsulates the top-tier memory structures of the core, i.e. L1/SMEM, the constant caches and the L2 caches. In a future variant of the model, the LDSTU will contain the texture caching subsystem, i.e. texture caches and texture mapping units, as well. An overall overview of the LDSTU is depicted in Figure 4.

As the figure shows, a memory access instruction for an entire warp is first passed to the address generators. Given base addresses as well as strides and offsets, the address generation unit (AGU) generates one memory address per thread in the warp. Given reasonable warp sizes of 32/64 threads in modern architectures, this requires very high bandwidth address generation units that supply the later stages of the memory subsystem with 32/64 memory addresses each cycle. We model the complete AGU as an array of parallel high-bandwidth sub-AGUs (SAGU), each of which is able to generate 8 memory addresses per cycle. Each SAGU[GGC+] consists of 8 address generators, each consisting of a 32-wide 3/2 counter array and a 32-bit 2/1 adder, to generate the 8 final addresses. Each of these 8 generators is supplied with a base address, an index and a stride generated by a hardwired stride generator (not modeled, negligible). The major hardware effort is the generation of the base addresses, performed by the main accumulator circuit. For each of the 8 base addresses, accumulator circuitry out of the following building blocks is used: 32 4/2 counters (modeled as two 3/2 counters, i.e. overall 64 full adders), 32 2-bit registers (64 DFFs), and 32 2/1 bit adders (half adders). For the adders, we employ a low-transistor-count low-power model as given by [SSSA10].

Given the memory address bundle for all threads in the warp, the address bundle is further analyzed depending on the type of memory the instruction accesses.

If the instruction accesses constant memory, the addresses are checked for equality (not modeled, negligible). The number of generated constant cache / constant memory accesses is equal to the number of *different* addres ses in the address bundle, e.g. if all addresses are equal, the memory access can be serviced with a single constant memory request, allowing for high-bandwidth operation. The constant memory segment (64KB in modern GPUs) is cached in an entire hierarchy of constant caches[WPSAM10] consisting of an L1 cache (per core), L2 cache (per TPC), and L3 cache (per GPU). The sizes, line sizes and associativities of the three cache tiers are configurable and currently modeled using CACTI according to the aforementioned paper [WPSAM10].

If the instruction accesses global memory, it is first *coalesced* before being passed to the L1/L2 caches and/or DRAM. The coalescing system is modeled after a corresponding NVIDIA patent[US8086806] and consists of an input queue, output queue, pending request table, and a finite state machine (not modeled, negligible). The goal of coalescing is to service the addresses requested by the memory access in as few memory requests as possible. The input queue has to buffer the incoming address bundles while the finite state machine operates to coalesce the memory access represented by a previous address bundle. Since such address bundles are large (e.g. using a warp size of 32 threads and a hypothetical memory address width of 32 bits, a single address bundle has a size of 1 Kbit), the input queue is kept small and its size is determined empirically. Once a bundle arrives at the head of the queue, it is copied into the Pending Request Table (PRT). The PRT is a very large table capable of holding one address bundle for each in-flight warp. The entries of the PRT are produced by the FSM. Once the FSM has picked a PRT entry using a simple scheduler, it selects the thread with the lowest ID in the warp the PRT entry belongs to (using a simple scheduler as well), computes the memory block which holds the address requested by that thread, and determines all other threads in the warp whose addresses target this block as well. Usually, the block size is set to the size of an L2 cache line (128B). The FSM will then generate a memory request for that block, put the request into the output queue, switch the threads that have been serviced by the request off in the PRT entry using a bit mask, and repeat the memory request generation process until all addresses in the bundle have been serviced. Then, the PRT entry is marked as invalid to be able to buffer a new entry for this warp. The output request queue, as the name implies, holds the outgoing memory requests while they are gradually dispatched to L1/L2/DRAM. Each queue entry consists of a base address, requested block size, serviced threads mask and corresponding PRT entry ID. Since the coalescing system contains buffers (input queue, PRT) that have only few but very large entries, CACTI cannot be used to model these buffers. Instead, we compute the total amount of bits which must be held in the coalescing system at any time and model the required storage using DFFs.

In several modern GPUs, shared memory (SMEM) and the L1 data cache are portions of the same physical memory structure. The distribution of physical memory to L1 and SMEM is configurable, for example, in the NVIDIA Fermi GPUs, there are 64KB of on-chip memory for usage as SMEM/L1 that can be configured to work in 16KB/48KB or 48KB/16KB SMEM/L1 partitions. Therefore, we model L1 and SMEM as an integrated physical memory structure and convert accesses to SMEM and L1 hits to accesses to that memory structure. The physical memory consists of multiple banks to be able to supply multiple accessing threads with data at a high rate. Besides the physical memory banks, the SMEM/L1 consists of interconnects for addresses and data, both modeled as crossbars, and a bank conflict checking unit. The bank conflict checking ensures that the addresses generated by the warp threads all fall into different banks. If so, the memory request can be serviced in a single transaction, otherwise, the access has to be partitioned into multiple serialized accesses. Bank conflict checking is similar to coalescing in both concept and hardware implementation and we therefore model the bank conflict checking functionality using an instance of the coalescing system described above.

The L2 cache is shared over the entire GPU and connected to the cores through the NoC. As such, the speed at which the L2 cache can store and supply data is limited by the transfer width and speed of the NoC. The line size, overall cache size and associativity of the cache are kept configurable.

3.2.5 Global Memory

The global memory in GPUs has high bandwidth but long latency. The current generation of GPUs such as Fermi use either DDR3 SDRAM or GDDR SGRAM chips to implement the global memory. The power consumed by typical DDR or GDDR chip can be divided into background, activate, read/write, termination, and refresh power [Mic07]. Background power is always consumed during the normal operation of the DRAM and can be determined by the percentage of time the DRAM is precharged or active. To allow a DRAM to read or write data, a bank and row are selected using activate (ACT) command and for each ACT command, there is a corresponding precharge (PRE) command. The PRE command closes the row. The data sheet for GDDR SGRAM [Hyn10] specifies the current IDD0 drawn during ACT-PRE cycle averaged over time with the interval between ACT commands being tRC. But, actual activate power consumed can be calculated by determining the average ACT-PRE command scheduling during the application run. The data sheet also specifies the current drawn during read/write operation. To scale the data sheet power to actual power based on command scheduling, it is calculated as the ratio of the read and write bandwidth. DRAM also consumes power for terminating the I/O signals. The termination power is system dependent. We estimate the termination power by calculating the DC power of the output drivers against the termination. Refresh is the final power component that must be calculated for the device to retain data integrity. DDR3 memory cells store data information in small capacitors that lose their charge over time and must be recharged. The process of recharging these cells is called refresh. The specification sheet specifies the current drawn during refresh assuming DRAM is continuously refreshed at minimum REFRESH-to-REFRESH command spacing,tRFC (MIN). However, REFRESH operations are typically distributed evenly over time at a refresh interval of tREFI. Thus, we scale the power according to tREFI.

3.3 GPU performance simulator

A GPU is a many-core, heavily multi-threaded processor. To handle the increasing complexity of designing the large chips which todays GPUs are, academics and industry have settled for high-level C++ simulators to experiment with and explore possible architectural design choices before building an actual chip.

For the purpose of building a framework to perform power simulation, we investigated several available architectural simulators. In order to be suitable for this activity, the simulator must be sufficiently flexible to allow reconfiguration of the simulated architecture and as accurate as possible to keep the errors before the power modeling step reasonably small. In addition, since the LPGPU project targets future GPU architectures and has a GPGPU focus, the architecture the simulator was designed for should be as modern as possible and feature GPGPU capabilities. Many well established simulators (Barra, Ocelot, Attila, multi2sim) do not meet this requirement, for example, multi2sim simulates a now-outdated VLIW5-based AMD architecture (Cypress) and Barra is a functional simulator for the 2006 NVIDIA G80 chip.

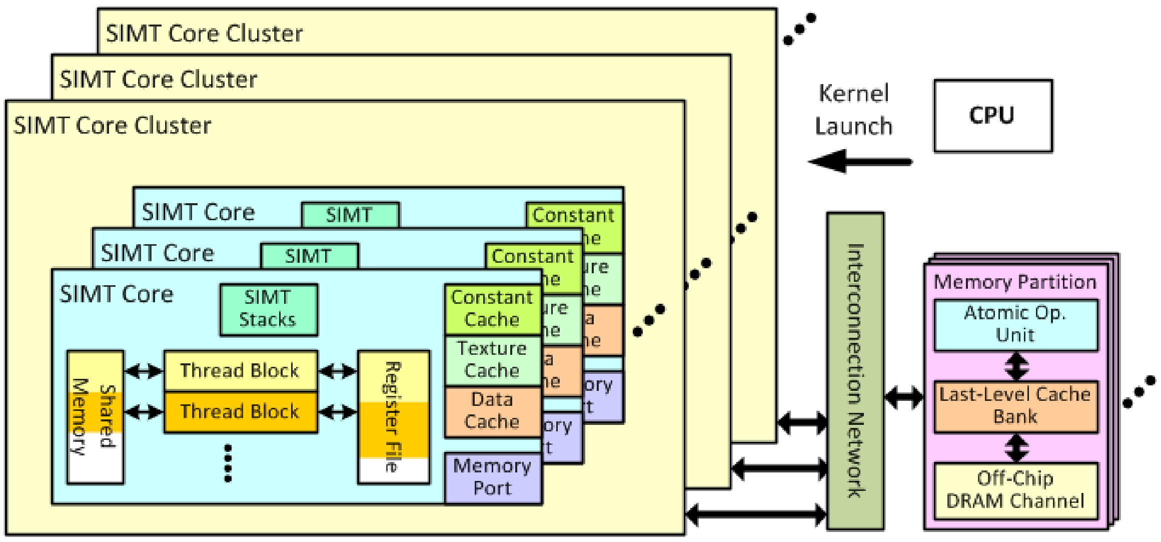


Figure 5: GPU architecture modeled by GPGPU-sim (Source: [FA12])

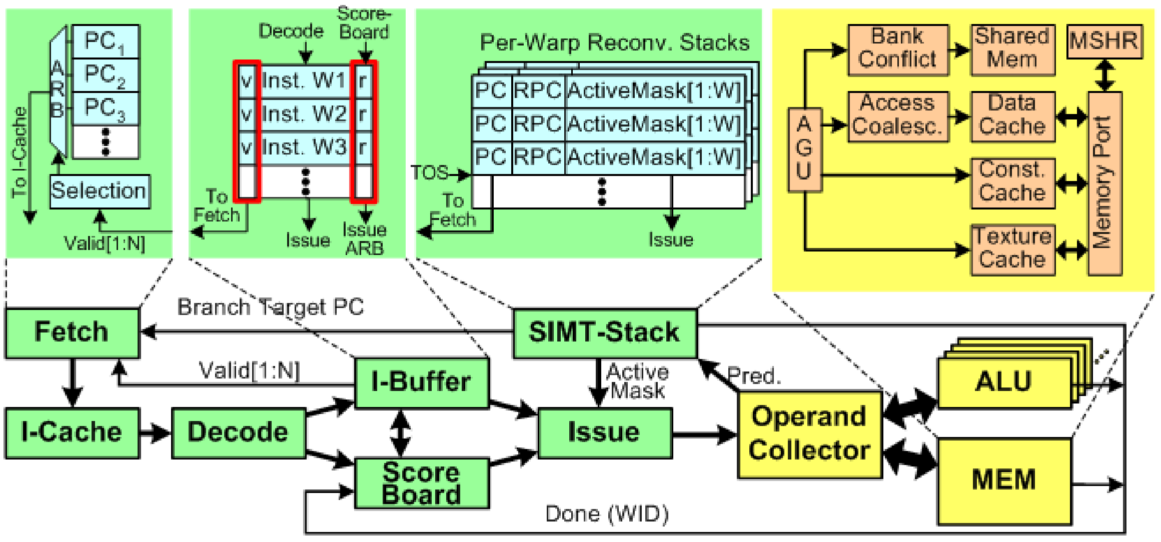


Figure 6 : Micro-architecture of SIMT core(Source: [FA12])

As a result, we chose GPGPU-Sim[BYF+09]. It is a cycle-accurate GPU performance simulator that focuses on general-purpose computation on GPUs. The micro-architecture modeled by GPGPU-sim is similar to many of the modern GPUs, such as NVIDIA G8x, G9x, GT200, Fermi, Kepler, and AMD GCN. An overview is given in Figure 5. The figure shows that the general architecture consists of a set of SIMT cores connected via an on-chip network (NoC) to memory partitions that provide the interface to the graphics memory (GDDR). A more detailed view of the GPU core pipeline and hardware structures is shown in Figure 6. Warps to fetch from are selected using a warp status table, fetched instructions are buffered in an instruction buffer, their dependencies are resolved using a simple scoreboard or barrel processing constraints, the control flow is controlled using reconvergence stacks, and the core back-end is split into functional units and a load-store system similar to the one described for GPGPUPow in Section 3.2.4. GPGPU-Sim allows reconfiguration of the key architecture parameters using a simple configuration file. The simulation is limited to the compute portion of the GPU itself, however, with no graphics (e.g. raster operators) or PCIe components modeled. The architecture GPGPU-sim was validated against is the NVIDIA GT200 GPU. Using the popular Rodinia benchmark suite, the simulator achieves 98.3% IPC correlation for this architecture. When configured to simulate the more recent Fermi micro-architecture, IPC correlation is still 97.3%[FA12]. While IPC correlation might be a good metric to cover the relative connection between real hardware IPC and simulator IPC, i.e. a workload A that executes with doubled IPC compared to a workload B on the real hardware does the same on the simulator, IPC correlation is not well suited to argue about the absolute accuracy of the simulator compared to the hardware. Besides this difference between relative and absolute accuracy, IPC correlation also effectively masks some of the bad IPC outliers. As the manual for GPGPU-Sim 3.x reveals, the IPC difference between simulated and real IPC, normalized to the real IPC, is sometimes as high as 258% (hardware IPC 7.33, simulated IPC 26.26). On average, it is still 35.5%. While these numbers may not sound convincing, GPGPU-Sim is still the most up-to-date and accurate GPU simulator publicly available, and relative accuracy is the major factor for developing a useful power simulator.

For the purpose of power simulation, the output of GPGPU-Sim are *activity factors* for the individual hardware components on a level as fine-grained as possible. In terms of GPGPU-Sim, we can estimate the activity factor for a component by counting the number of accesses to this component. GPGPU-Sim provides general statistics such as number of instructions simulated, total number of cycles as well as the activity/usage of different components such as L1, L2 cache accesses, misses and miss rate etc. However, we also need some activity factors which the simulator does not generate directly. We therefore extended GPGPU-Sim with new performance counters. For example, to calculate the power consumed by the branch divergence hardware we need the number of accesses to the warp status table and reconvergence stacks. All activity factors acquired by GPGPU-Sim during simulation are written to a human-readable XML file to be post-processed and passed to GPGPUPow. Examples of two activity factors generated for one of our benchmarks, a Black-Scholes PDE solver, are given below. The first activity factor, `af_generated_gmem_reads`, lists the total number of read requests to global memory that have been generated during the execution of the kernel. Due to coalescence, this number is usually different from the total number of instructions that read from global memory. The second activity factor, `af_rdsch_percent`, specifies the percentage of clock cycles which are outputting read data from the global memory for this particular kernel.

XML activity factor	Description
`<stat name="af_generated_gmem_reads" value="524288"/>`	Number of global memory read commands sent to the memory controllers
`<stat name="af_rdsch_percent" value="0.395083"/>`	Bus utilization by read data

Table 1: Example for XML activity factors

3.4 Measurement-based validation

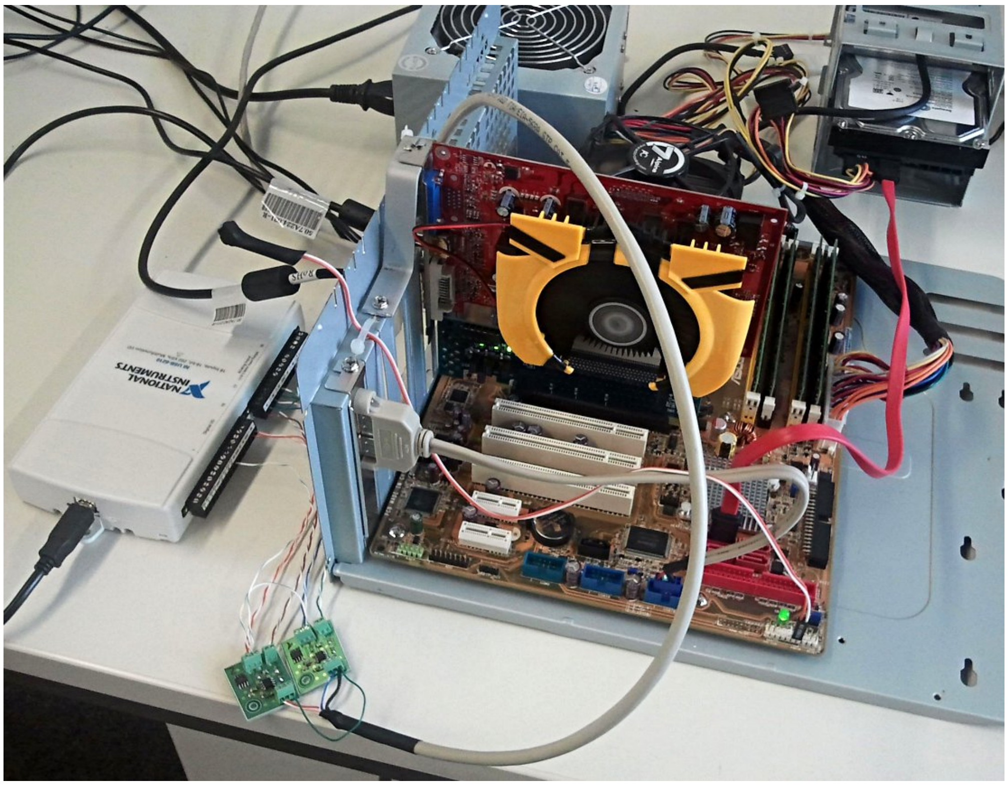


Figure 7 : Power measurement setup

We developed our own measurement setup to validate the simulator against real GPUs. Figure 7 shows the hardware part of this setup with a GT240 card. This measurement setup was also used together with custom micro benchmarks to estimate the energy per operation for different execution units. Our measurement setup consists of hardware and software components. Figure 8 shows how these parts work together. We are using a riser card with current sensing resistors to probe current and voltages going to the GPU via the PCIe slot. The GT240 has no external PCIe power connector, but our measurement setup can also measure power consumed via external PCIe power connectors in addition to the power consumed via the slot. A custom signal conditioning board amplifies the voltages drops on the measurement resistors and shifts them into 0-5 V range. The voltages are scaled also into that range. After this signal condition a NI-USB6210 DAQ is used to sample the signals. We wrote a custom measurement software that controls the DAQ and calculates power and energy values from the measured voltage and current values.

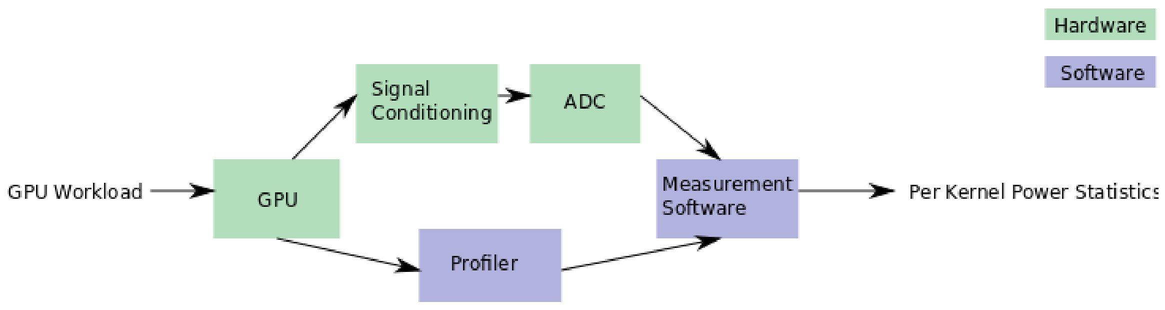


Figure 8: Power Validation framework

This measurement tool is able to use the GPU profiler to get start and stop timestamps of the kernels running on the GPU. Using that information and measured power waveform, the average power for each kernel execution cand be calculated as well as the amount of energy used. Table 2 lists a part of an example power report for running the MergeSort sample from the CUDA SDK. Figure 9 is a sample power measurement from the same benchmark. The black bars in the graph mark kernel starts and endings. The measurements here clearly show that different kernels can have very different levels of power consumption.

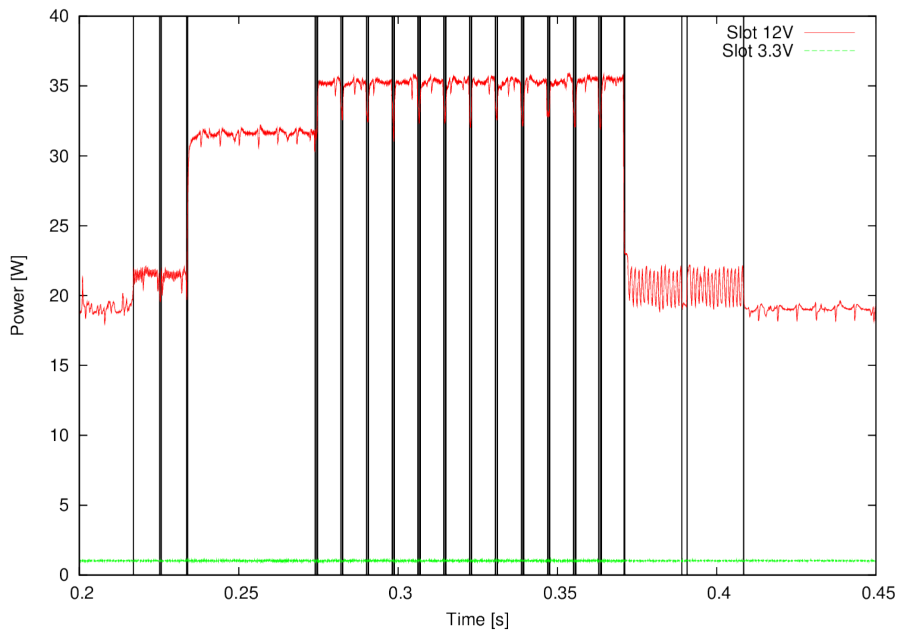


Figure 9: Sample measurement tool output

Index	Method	Start	End	Runtime	Energy	Power
0	memcpyHtoD	0.216929	0.225204	0.008275	0.186147 J	22.496014 W
1	memcpyHtoD	0.225676	0.233653	0.007977	0.178779 J	22.410513 W
2	mergeSortSharedKernel	0.233967	0.274031	0.040064	1.303852 J	32.544016 W
3	generateSampleRanksKernel	0.274035	0.274590	0.000554	0.018100 J	32.642466 W
4	mergeRanksAndIndicesKernel	0.274593	0.274667	0.000074	0.002490 J	33.539748 W
5	mergeRanksAndIndicesKernel	0.274671	0.274746	0.000075	0.002557 J	34.201843 W
6	mergeElementaryIntervalsKernel	0.274750	0.282088	0.007338	0.002557 J	34.201843 W
7	generateSampleRanksKernel	0.282091	0.282604	0.000513	0.017504 J	34.154221 W
8	mergeRanksAndIndicesKernel	0.282607	0.282657	0.000050	0.001717 J	34.213496 W
9	...					

Table 2: Example Power Report from the Measurement Tool

We then executed various benchmarks on the GPU using our measurement setup and recorded the energy and power used by each kernel. Thereafter, we simulated the same benchmarks on the simulator using a configuration file that tries to match the real hardware and compared the results.

The measurement setup was also used to build the measurement based model for the execution units. The used GPUs use a single-instruction-multiple-data (SIMD) approach to execute multiple threads at the same time. We can use this style of execution to enable different numbers of execution units, while the activity of all other units, except for the register files, stays constant. Our measurements show that integer instructions are using approximately 40 pJ while floating point instructions are using about 75 pJ per instruction. NVidia reports 50 pJ per floating point instruction[KDK+ 11].

4. Experimental Evaluation

4.1 Benchmarks

We used the simulator to simulate seven different benchmarks from the NVIDIA CUDA SDK 3.1 Samples. These benchmarks have 11 different kernels. The same benchmarks with same parameters were also executed on a GT240 in our power measurement setup. GPUSimPow was used to predict the power consumption of these benchmarks on a per kernel level. Table 3 shows the benchmarks, input sizes, number of kernels per benchmark, and calls per kernel.

Name	Description	Size (32-bit words)	No. of Kernels	Calls per Kernel
MatrixMul	Matrix-Matrix-Multiplication	A(160×80), B(80×80)	1	10
Transpose	Compute transpose of a given matrix	A(256×4096)	1	5
BlackScholes	Black-Scholes PDE solver	A(4000000)	1	5
ScalarProd	Computes the scalar product of two vectors	Vector(256),Element(4096)	1	1
VectorAdd	Computes component-wise sum of two vectors	A(500000), B(500000)	1	1
Reduction	Performs a reduce operation on a given dataset	A(16777216)	2	101,100
MergeSort	Sorting using a parallel merge-sort algorithm	A(32768)	4	12,1,12,24

Table 3: Benchmarks

4.2 Experimental Results

In this section we present our power simulation results and compare them with the measured results.

4.2.1 Performance simulation

We use the activities reported by the extended GPGPU-Sim performance simulator to calculate the runtime power. The accuracy of activities generated by performance simulator effects the accuracy of power simulation. Table 4 shows the relative error for benchmarks used. Simulated time is the time reported by GPGPU-Sim performance simulator and measured time is the time reported by the CUDA profiler. The relative error for the benchmarks shown in table is equal to or less than 20% for all benchmarks except the Transpose and generateSampleRanksKernel kernel. The simulated time reported by GPGPU-Sim is always larger than the measured time except for one kernel of the Mergesort benchmark. We discuss the effect of relative error on predicted runtime power by GPUSimPow in the power simulation subsection.

Name	Kernel	Simulated Time(ms)	Measured Time(ms)	% Relative Error
MatrixMul	matrixMul	0.0507	0.05	1.414
Transpose	transpose	1.375	0.587	134.24
BlackScholes	blackscholes	4.1542	3.6780	12.96
ScalarProd	scalarprod	0.4731	0.394	20.0
VectorAdd	vectoradd	0.269	0.271	-0.75
Reduction	reduction256	3.4283	2.859	19.91
Reduction	reduction32	0.00323	0.003	7.56
MergeSort	mergeSortSharedKernel	32.0716	40.044	-19.91
MergeSort	generateSampleRanksKernel	0.7686	0.545	41.02
MergeSort	mergeRanksAndIndicesKernel	0.0733	0.072	1.82
MergeSort	mergeElementaryIntervalsKernel	8.6667	7.342	18.04

Table 4: Relative error between simulated and measured execution time for different kernels

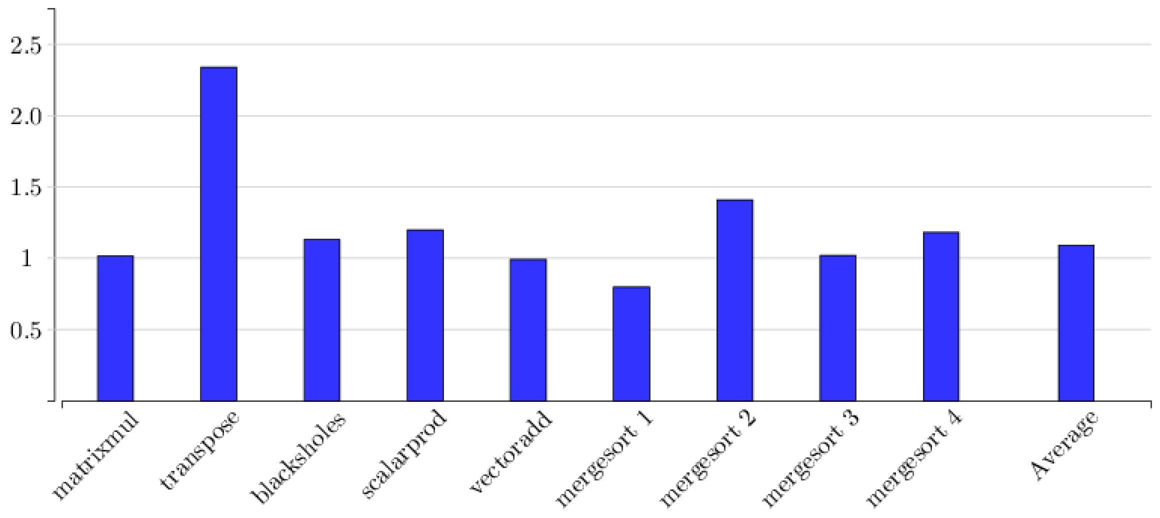


Figure 10: Simulated runtimes normalized to measured runtimes

Figure 10 shows the simulated runtimes normalized to measured runtimes. For most of the benchmarks GPGPU-Sim is able to predict the runtime within 20%. Transpose and mergesort 2,however , are outliers and use over 130% (transpose) and over 40% more time (mergesort 2) in the simulator than on the real hardware. We will later see that this also affects the accuracy of the power simulation results.

4.2.2 Power Simulation

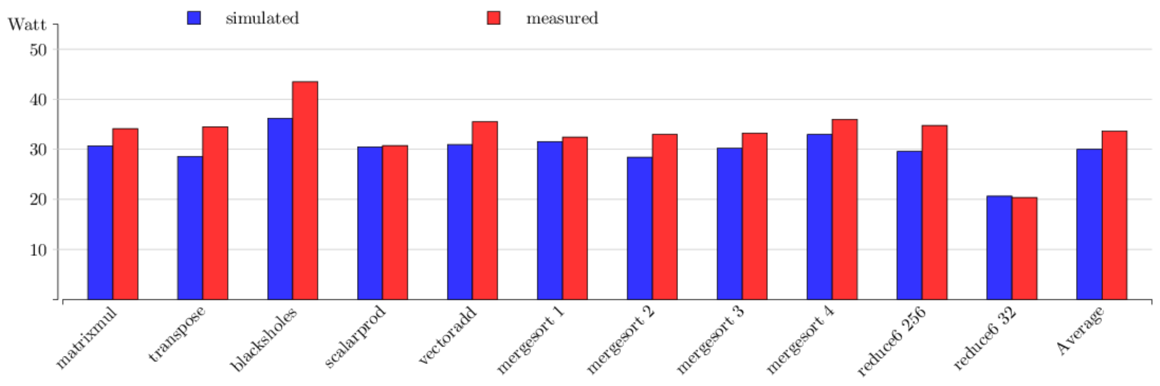


Figure 11: Simulated vs measured power for GT240

Figure 11 depicts the simulated and measured power for a subset of CUDA SDK benchmarks for GT240. The simulated power is reported by GPUSimPow and we measured the real hardware power using our power measurement infrastructure. The absolute error for simulated power w.r.t measured power varies from about 1% to 21% and absolute error on average is 10.75% for the benchmarks shown in Figure 10. McPAT also has absolute error in the range of 20% and just like our simulator it also constantly underestimates the absolute amount of power as seen in Figure 11.

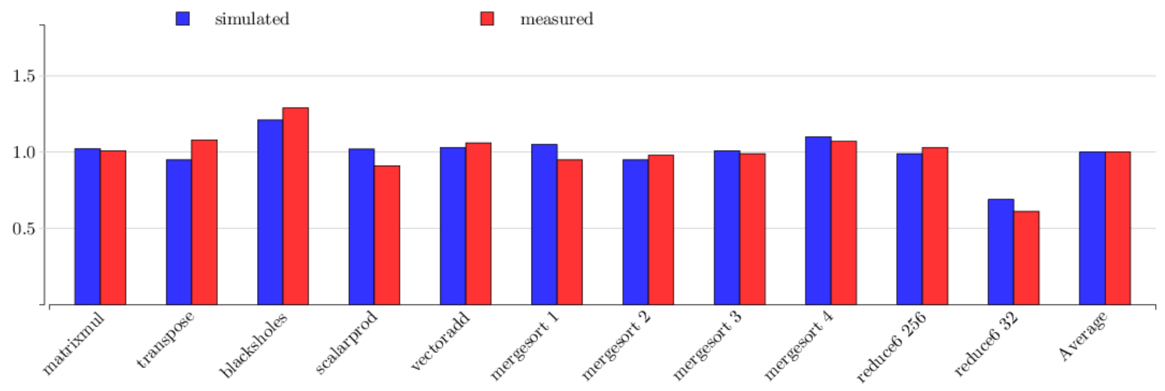


Figure 12: Normalized simulated and measured power

As discussed in the performance simulation section the accuracy of the simulated runtime power depends upon the accuracy of the simulated runtime reported by performance simulator. This is reflected by the results shown in Figure 11. The power simulation relative error is higher for benchmarks with higher simulated runtime error. For instance, the Transpose benchmark has highest relative error for simulated runtime which is clearly reflected in its simulated power. The relative error for simulated power for Transpose benchmark is more than 21% which is also highest among all benchmarks used. One might expect a equally huge error in power estimation as the runtime estimation error made by the performance simulator, but this is not the case as a large amount of the total power is constant leakage power and dynamic power used by units that are always active. The error in runtime estimation made by the performance simulator causes a large error in the estimation of the dynamic part of the power but not in the other parts of the power.

For an architectural power simulator, however, absolute accuracy is not its top priority. More important is that such a simulator is able to predict the relative changes caused by different workloads and different architectures as accurate as possible. In Figure 12, we plot simulated and measured power normalized to their average values. The graph shows how accurately the GPUSimPow simulator models the relative power differences caused by different workloads. It can be seen that the simulated relative power matches the measured power closely for most benchmarks where the performance simulator produces accurate runtime time numbers: matrixmul, vectoradd, and mergesort 3. Reduce 6 32 is an extremely short kernel, so the accuracy of the measurement is likely the problem here.

4.2.3 Architecture comparison

	GT240		GTX580		GTX580/GT240	
	Peak Power(W)	**Area(mm²)**	**Peak Power(W)**	**Area(mm²)**	**Power Ratio**	**Area Ratio**
GPUSimPow	94.44	92.98	287.5	292.78	3.05	3.15
NVIDIA Specification	69.00	139	244	520	3.54	3.74
				Relative Error	-13.84%	-15.78%

Table 5: Shows the peak power and area numbers for NVIDIA GT240 and GTX580 GPUs.

Table 5 shows how GPUSimPow predicts the peak power and area for two different NVIDIA cards with different architectures. Peak Power is constantly overestimated. We estimate peak power by estimating peak power for every component and adding them together. This cannot happen in real workloads. It is therefore not a surprise that the simulator constantly overestimates the power. With regard to area, it can be said that as our power simulator does not model some units not related to GPGPU workloads such as ROPs, triangle setup engines or fixed-function video codec circuits, the underestimation of area compared to the real chips is expected behaviour as well.

5. Status

The requirements for this task were the following:

- Minimum results: Accurate power estimations of within 25% of real power consumption.
- Expected results: Accurate power estimations of within 10% of real power consumption.
- Exceeding expectation: Accurate power estimations of within 5% of real power consumption.

As we have shown in the evaluation section, the simulation framework generates average runtime power numbers within 11% of the absolute power measured on real hardware. The minimum results for this activity were within 25%. Thus, for absolute runtime power consumption, we achieved the minimum results for this activity. Moreover, the results show that the transpose benchmark has 21.3% absolute error for simulated power. This error is quite high compared to other benchmarks. The reason for this is that the performance simulator is not able to accurately predict the runtime. The average absolute error calculated by excluding the transpose benchmark is 9.6% which is well within 10% and thus we also achieve the expected results.

Besides absolute power, an important metric is the relative variation in power, i.e. if workload A consumes two times as much power as workload B on the real hardware, this relative variation should also be modeled accurately by the simulator. For this relative power metric, our evaluation has shown that the simulation framework is accurate within 10% of the behavior on real hardware, thus achieving the expected results for the activity.

For architectural design space explorations, the accurate simulation of relative power variations caused by changes to the architecture is another important metric, i.e. if architectural configuration A consumes twice as much power as architectural configuration B, the simulator results should capture this relationship. As Table 5 in the previous section reveals, the relative errors for scaling a GT240-like architecture to the capabilities of a GTX580 GPU are about 13 (power) and 15 (area) percent compared to the scaling in real hardware. Thus, we conclude that the goal of being accurate within 25% has been achieved for these numbers.

Overall, the power simulator implementation we provided achieves the minimum or expected results for this activity, depending on the chosen metric.

6. Advances over State of the Art

Our results advance the state of the art in several ways:

- We combine architectural and measurement based power modeling techniques to allow for a flexible yet accurate power simulation framework.
- We natively model power for some of the most modern GPU architectures to date.
- We proposed and developed an improved measurement setup to be able to perform reliable power measuring experiments on real hardware.

We have built a power model that combines power estimations derived from analytical models of memory and logic structures with measured values derived through microbenchmarking of real GPUs. This combination is a new approach. Previous GPU power models[HK10][MDZD] have either relied completely on measurements and were not able to make predictions about GPUs with changed architectural parameters, or they only used analytical models and numbers from RTL power simulation tools, but no measurements of actual power used [KRS07] and thus showed poor correlation with actual industry data. Only by combining analytical and empirical representations we achieve the desired level of (relative) accuracy as well as maintaining a model that is flexible enough to generalize to architectural modifications. This way, our simulator enables computer architects to perform design-space explorations for future GPU architectures in a way that integrates performance and power viewpoints. This integrated approach has become much more important in the recent years as processors in general and GPUs in particular approach the *power wall*.

From an architecture point of view, most previous approaches to the modeling of GPU power have used architectures which are outdated from today's point of view. The most commonly used architectures are similar to the NVIDIA GT200 (2008) and AMD RV870/Cypress (2009) GPUs. While the former is outdated with regard to its GPGPU capabilities (e.g. no caches, poor FP64 performance, slow kernel switching …), the latter features a VLIW5 architecture that has nowadays been surpassed by wide scalar SIMT architectures. Overall, none of the previously published power modeling approaches take into account the most recent trends in GPU architecture design. Our approach improves upon those shortcomings by natively addressing a general architecture similar to the NVIDIA Fermi/Kepler and AMD GCN GPUs.

We also estimate the power of the complete GPU system. This does not only include the power consumed by the main GPU chip but also the power consumed by the external memory and its interface and the power supply circuits on the GPU board.

During our literature research we also found that many existing power models are based on inaccurate measurement methodologies. Some of the power models are based on measuring the entire PC power and subtracting the power that was measured when the PC was idle[HK10]. This is a highly naive way of measuring the power: The power used by the remaining PC components is not constant and the measurement results will also contain ATX power supply losses. Large bypass capacitors inside the ATX power supply also prevent accurate power measurement on common kernels with kernel runtimes below 50 ms. Other papers use improved measurement methodologies, but still exhibit multiple weaknesses. These published methodologies either fail to measure all power sources and do not measure the power provided via the graphics card slot[WR11], measure only current and expect constant voltages[NMN+ 10], or use low sampling frequencies that prevent them from measuring short-term power variations[WR11][MDZD].
We improve upon that by measuring all power rails available to the card and measuring currents and voltages for all rails and sampling at an extremely high speed.

7. Conclusions and Future Work

In this task, we have developed a power simulation framework for the compute capabilities of contemporary GPUs called GPUSimPow. The framework consists of a cycle-accurate simulator loosely coupled with a power estimation tool. Given a specification of the architecture core parameters as well as a GPGPU workload, the framework is able to deliver numbers for chip area, static and peak dynamic power, as well as runtime dynamic power for the specified workload. Our evaluation of the simulation framework has shown that when compared to NVIDIA GT240 and GTX580 GPUs, the chip area, static, and peak dynamic power numbers are within the accuracy requirements for this task. On the GT240, we have evaluated both absolute and relative accuracy of the runtime power consumption reported by the simulation framework. Using our own measurement setup specifically created for the task of accurately measuring power consumed by contemporary GPUs during load, we found that absolute runtime power simulations are within 11% and relative runtime power simulations are within 10% of the real hardware power consumption.

Future work opportunities for this task are twofold: On the one hand, the existing simulation framework can be used to conduct both architectural and software studies focusing on GPU power, and on the other hand, the current state of the power simulator allows for a series of improvements.

Regarding usage of the power simulator as it is, multiple other tasks in the LPGPU project will study applications and application transformations using the simulator. In its current state, the relative accuracy of the results reported by the simulator is sufficiently high that such application studies can be conducted to produce meaningful results. Besides exploring changes to applications, the simulator will to be used to perform design space exploration for low-power GPU architecture. While the state of the simulator enables such studies in general, the empirical nature of some parts of the simulator currently limits its usefulness for studying modifications to these parts since their empirical models have not yet been validated against a sufficiently large number of real architectures. For studies that target architectural components that have not been modeled empirically but analytically, such as the warp scheduler, caches, or register file, the simulator can already provide accurate power predictions for architectural modifications. The first task to use this capability is the integration of SIMD-MIMD architectural enhancements into the simulator.

Regarding the improvement of the simulator, as mentioned before, a first step will be to validate the runtime simulations against measurements on other types of architectures than GT200, e.g. Fermi, Kepler, or GCN. Besides other architectures, the simulator must also be tested with and validated using a larger number of applications and benchmarks. While we previously considered relative accuracy only with regard to modifications of applications or architecture, an interesting opportunity would be to investigate the relative power consumption of individual components inside the architecture. Such an analysis could provide us with useful insights to start further low-power architectural studies from.

8. References

[AMD12] AMD. AMD Graphics Core Next (GCN) Architecture White Paper, 2012.

[BYF+ 09] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on, April 2009.

[CL09] Brett W. Coon and John E. Lindholm. System and method for managing divergent threads using synchronization tokens and program instructions that include set-synchronization bits, 06 2009.

[CMOS08] Brett W. Coon, Peter C. Mills, Stuart F. Oberman, and Ming Y. Sui. Tracking register usage during multithreaded processing using a scoreboard having separate memory regions and storing sequential register size indicators, 08 2008.

[DCPS08] D. De Caro, N. Petra, and A.G.M. Strollo. A high performance floating-point special function unit using constrained piecewise quadratic approximation. In Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on, pages 472–475, may 2008.

[FA11] W.W.L. Fung and T.M. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on, pages 25 –36, Feb. 2011.

[FA12] W.W.L. Fung and Tor Aamodt. GPGPU-Sim 3.x Manual, 2012.

[GGC+] Carlo Galuzzi, Chunyang Gou, Humberto Calderon, Georgi N. Gaydadjiev, and Stamatis Vassiliadis. High-bandwidth Address Generation Unit. J. Signal Process. Syst.

[GH11] Sameh Galal and Mark Horowitz. Energy-efficient floating-point unit design. IEEE Transactions on Computers, 60:913–922, 2011.

[HK10] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. In Proceedings of the 37th annual
international symposium on Computer architecture, ISCA ’10, pages 280–289, New York, NY, USA, 2010. ACM.

[Hyn10] Hynix. Hynix 1 Gb (32Mx32) GDDR5 SGRAM H5GQ1H24AFR, 2010. http://www.hynix.com/datasheet/pdf/graphics/H5GQ1H24AFR(Rev1.0).pdf.

[KDK+ 11] Stephen W. Keckler, William J. Dally, Brucek Khailany,Michael Garland, and David Glasco. GPUs and the Future of Parallel Computing. Micro, IEEE, 31(5):7–17, 2011.

[KQM04] Cheong Kun, Shaolei Quan, and Andrew Mason. A Power-Optimized 64-Bit Priority Encoder Utilizing Parallel Priority Look-Ahead. In ISCAS, pages 753–756, 2004.

[KRS07] A Ibrahim K Ramani and D Shimizu. Powerred: A flexible power modeling framework for power efficiency exploration in gpus, 2007.

[LAS+ 09] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 469–480, 2009.

[LOS09] N. Leischner, V. Osipov, and P Sanders. Fermi Architecture White Paper, 2009.

[MDZD] Xiaohan Ma, Mian Dong, Lin Zhong, and Zhigang Deng. Statistical power consumption analysis and modeling for gpu-based computing.

[Mic07] Micron. Calculating Memory System Power for DDR3, 2007. http://www.micron.com/products/dram/ddr3-sdram.

[NMN+ 10] Hitoshi Nagasaka, Naoya Maruyama, Akira Nukada, Toshio Endo, and Satoshi Matsuoka. Statistical power modeling of gpu kernels using performance counters. In Green Computing Conference, pages 115–122, 2010.

[NNHM11] L. Nyland, J.R. Nickolls, G. Hirota, and T. Mandal. Systems and methods for coalescing memory accesses of parallel threads, 12 2011.

[NVI12] NVIDIA. Kepler GK110 White Paper, 2012.

[SSSA10] Tripti Sharma, B. P. Singh, K. G. Sharma, and Neha Arora. High Speed, Low Power 8T Full Adder Cell with 45Threshold Loss Problem. In Proceedings of the 12th international conference on Networking, VLSI and signal processing, 2010.

[TAM+ 08] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B. Brockman, and Norman P. Jouppi. A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies. In Proceedings of the 35th Annual International Symposium on Computer Architecture, pages 51–62, 2008.

[US7339592] Lindholm, J. E. and Siu, M. Y. and Moy, S. S. and Liu S. and Nickolls, J.R. Simulating multiported memories using lower port count memories. Patent. US 7339592. March 2008.

[US7434032] Coon, Brett W. and Mills, Peter C. and Oberman, Stuart F. and Sui, Ming Y. Tracking Register Usage During Multithreaded Processing Using a Scoreboard Having Separate Memory Regions and Storing Sequential Register Size Indicators. Patent. US 7434032. August 2008.

[US7543136] Coon, Brett W. and Lindholm, John E. System and Method for Managing Divergent Threads Using Synchronization Tokens and Program Instructions that Include Set-Synchronization Bits. Patent. US 7543136. June 2009.

[US8086806] Nyland, L. and Nickolls, J.R. and Hirota, G. and Mandal, T. Systems and Methods for Coalescing Memory Accesses of Parallel Threads. Patent. US 8086806. December 2011.

[WPSAM10] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on,March 2010.

[WR11] Yue Wang and Nagarajan Ranganathan. An instruction-level energy estimation and optimization methodology for gpu. In Proceedings of the 2011 IEEE 11th International Conference on Computer and Information Technology, CIT ’11, pages 621–628, Washington, DC, USA, 2011. IEEE Computer Society.

[ZHLP11] Ying Zhang, Yue Hu, Bin Li, and Lu Peng. Performance and power analysis of ati gpu: A statistical approach. Networking, Architecture, and Storage, International Conference on, 0:149–158, 2011.

[US7339592] Lindholm, J. E. and Siu, M. Y. and Moy, S. S. and Liu S. and Nickolls, J.R. Simulating multiported memories using lower port count memories. Patent. US 7339592. March 2008.

[US7434032] Coon, Brett W. and Mills, Peter C. and Oberman, Stuart F. and Sui, Ming Y. Tracking Register Usage During Multithreaded Processing Using a Scoreboard Having Separate Memory Regions and Storing Sequential Register Size Indicators. Patent. US 7434032. August 2008.

[US7543136] Coon, Brett W. and Lindholm, John E. System and Method for Managing Divergent Threads Using Synchronization Tokens and Program Instructions that Include Set-Synchronization Bits. Patent. US 7543136. June 2009.

[US8086806] Nyland, L. and Nickolls, J.R. and Hirota, G. and Mandal, T. Systems and Methods for Coalescing Memory Accesses of Parallel Threads. Patent. US 8086806. December 2011.