# Microarchitecture Model

This section describes the microarchitecture modelled by GPGPU-Sim 3.x. The model is more detailed than the timing model in GPGPU-Sim 2.x. Some of the new details result from examining various NVIDIA patents. This includes the modelling of instruction fetching, scoreboard logic, and register file access. Other improvements in 3.x include a more detailed texture cache model based upon the prefetching texture cache architecture. The overall microarchitecture is first described, then the individual components including SIMT cores and clusters, interconnection network and memory partitions.

## Overview

GPGPU-Sim 3.x runs program binaries that are composed of a CPU portion and a GPU portion. However, the microarchitecture (timing) model in GPGPU-Sim 3.x reports the cycles where the GPU is busy--it does not model either CPU timing or PCI Express timing (i.e. memory transfer time between CPU and GPU). Several efforts are under way to provide combined CPU plus GPU simulators where the GPU portion is modeled by GPGPU-Sim. For example, see http://www.fusionsim.ca/.

GPGPU-Sim 3.x models GPU microarchitectures similar to those in the NVIDIA GeForce 8x, 9x, and Fermi series. The intention of GPGPU-Sim is to provide a substrate for architecture research rather than to exactly model any particular commercial GPU. That said, GPGPU-Sim 3.x has been calibrated against an NVIDIA GT 200 and NVIDIA Fermi GPU architectures.

### Accuracy

We calculated the correlation of the IPC (Instructions per Clock) versus that of real NVIDIA GPUs. When configured to use the native hardware instruction set (PTXPlus, using the -gpgpu_ptx_convert_to_ptxplus option),GPGPU-Sim 3.1.0 obtains IPC correlation of 98.3% (Figure 1) and 97.3% (Figure 2) respectively on a scaled down version of the RODINIA benchmark suite (about 260 kernel launches). All of the benchmarks described in *[Che et. al. 2009]* were included in our tests in addition to some other benchmarks from later versions of RODINIA. Each data point in Figure 1 and Figure 2 represents an individual kernel launch. Average absolute errors are 35% and 62% respectively due to some outliers.

We have included our spreadsheet used to calculate those correlations to demonstrate how the correlation coefficients were computed: File:Correlation.xls.

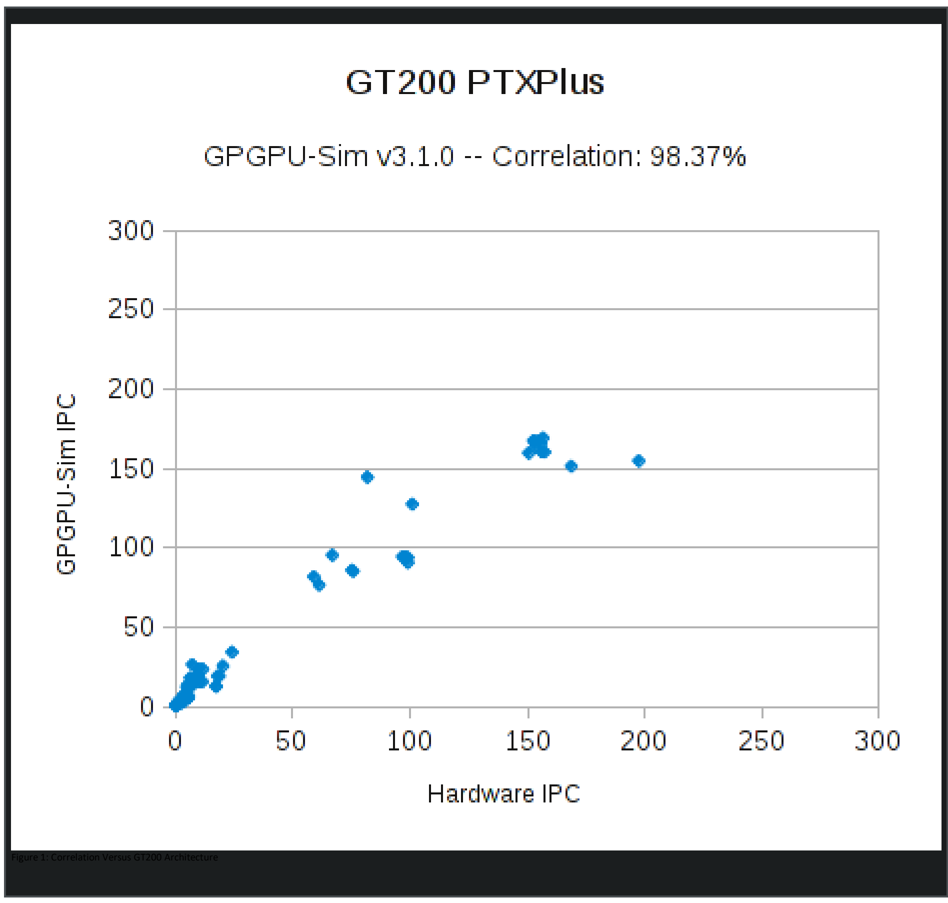


Figure 1: Correlation Versus GT200 Architecture

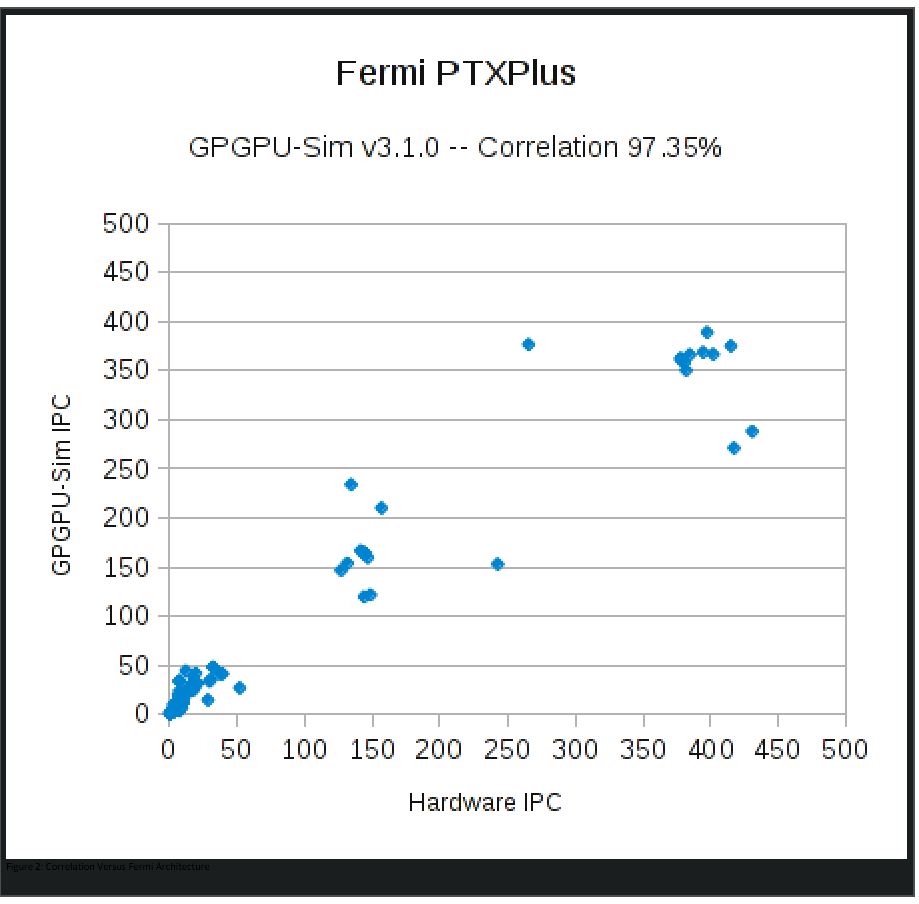


Figure 2: Correlation Versus Fermi Architecture

### Top-Level Organization

The GPU modeled by GPGPU-Sim is composed of Single Instruction Multiple Thread (SIMT) cores connected via an on-chip connection network to memory partitions that interface to graphics GDDR DRAM.

An SIMT core models a highly multithreaded pipelined SIMD processor roughly equivalent to what NVIDIA calls an Streaming Multiprocessor (SM) or what AMD calls a Compute Unit (CU). The organization of an SIMT core is described in Figure 3 below.

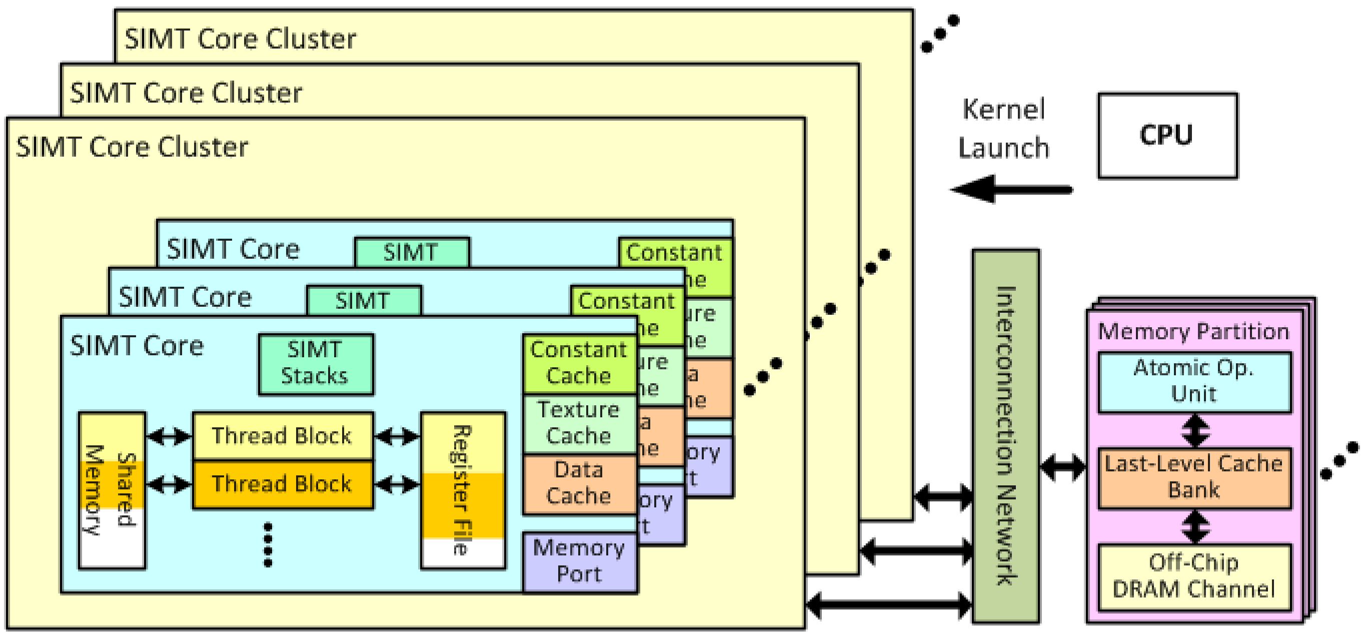


Figure 3: Overall GPU Architecture Modeled by GPGPU-Sim

### Clock Domains

GPGPU-Sim supports four independent clock domains: (1) the SIMT Core Cluster clock domain (2) the interconnection network clock domain (3) the L2 cache clock domain, which applies to all logic in the memory partition unit except DRAM, and (4) the DRAM clock domain.

Clock frequencies can have any arbitrary value (they do not need to be multiples of each other). In other words, we assume the existence of synchronizers between clock domains. In the GPGPU-Sim 3.x simulation model, units in adjacent clock domains communicate through clock crossing buffers that are filled at the source domain's clock rate and drained at the destination domain's clock rate.

## SIMT Core Clusters

The SIMT Cores are grouped into SIMT Core Clusters. The SIMT Cores in a SIMT Core Cluster share a common port to the interconnection network as shown in Figure 4.

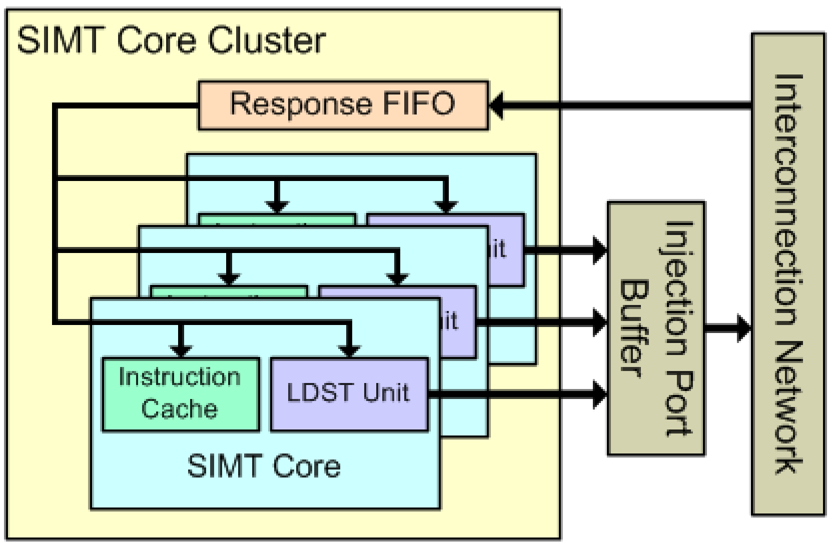


Figure 4: SIMT Core Clusters

As illustrated in Figure 4, each SIMT Core Cluster has a single *response FIFO* which holds the packets ejected from the interconnection network. The packets are directed to either a SIMT Core's instruction cache (if it is a memory response servicing an instruction fetch miss) or its memory pipeline (LDST unit). The packets exit in FIFO fashion. The response FIFO is stalled if a core is unable to accept the packet at the head of the FIFO. For generating memory requests at the LDST unit, each SIMT Core has its own injection port into the interconnection network. The injection port buffer however is shared by all the SIMT Cores in a cluster.

## SIMT Cores

Figure 5 below illustrates the SIMT core microarchitecture simulated by GPGPU-Sim 3.x. An SIMT core models a highly multithreaded pipelined SIMD processor roughly equivalent to what NVIDIA calls an Streaming Multiprocessor (SM) [1] or what AMD calls a Compute Unit (CU) [2]. A Stream Processor (SP) or a CUDA Core would correspond to a lane within an ALU pipeline in the SIMT core.

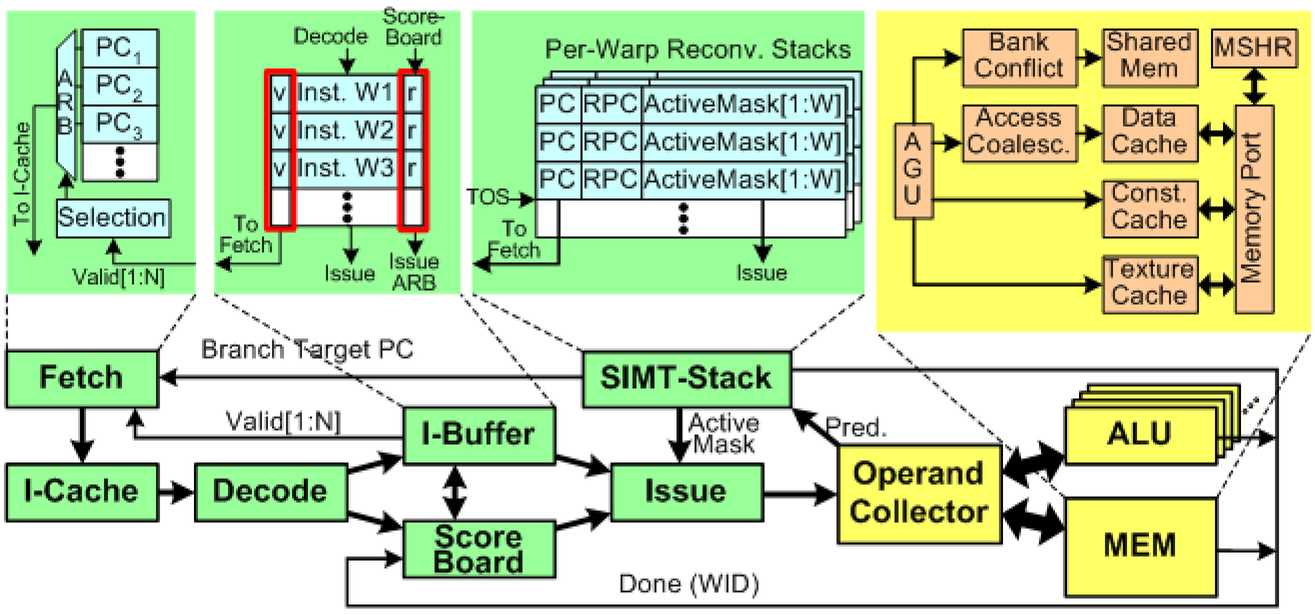


Figure 5: Detailed Microarchitecture Model of SIMT Core

This microarchitecture model contains many details not found in earlier versions of GPGPU-Sim. The main differences include:

- A new front-end that models instruction caches and separates the warp scheduling (issue) stage from the fetch and decode stage
- Scoreboard logic enabling multiple instructions from a single warp to be in the pipeline at once
- A detailed model of an operand collector that schedules operand access to single ported register file banks (used to reduce area and power of the register file)
- Flexible model that supports multiple SIMD functional units. This allows memory instructions and ALU instructions to operate in different pipelines.

The following subsections describe the details in Figure 5 by going through each stage of the pipeline.

## Front End

As described below, the major stages in the front end include instruction cache access and instruction buffering logic, scoreboard and scheduling logic, SIMT stack.

### Fetch and Decode

The instruction buffer (I-Buffer) block in Figure 5 is used to buffer instructions after they are fetched from the instruction cache. It is statically partitioned so that all warps running on SIMT core have dedicated storage to place instructions. In the current model, each warp has two I-Buffer entries. Each I-Buffer entry has a valid bit, ready bit and a single decoded instruction for this warp. The valid bit of an entry indicates that there is a non-issued decoded instruction within this entry in the I-Buffer. While the ready bit indicates that the decoded instructions of this warp are ready to be issued to the execution pipeline. Conceptually, the ready bit is set in the schedule and issue stage using the scoreboard logic and availability of hardware resources (in the simulator software, rather than actually set a ready bit, a readiness check is performed). The I-Buffer is initially empty with all valid bits and ready bits deactivated.

A warp is eligible for instruction fetch if it does not have any valid instructions within the I-Buffer. Eligible warps are scheduled to access the instruction cache in round robin order. Once selected, a read request is sent to instruction cache with the address of the next instruction in the currently scheduled warp. By default, two consecutive instructions are fetched. Once a warp is scheduled for an instruction fetch, its valid bit in the I-Buffer is activated until all the fetched instructions of this warp are issued to the execution pipeline.

The instruction cache is a read-only, non-blocking set-associative cache that can model both FIFO and LRU replacement policies with on-miss or on-fill allocation policies. A request to the instruction cache results in either a hit, miss or a reservation fail. The reservation fail results if either the miss status holding register (MSHR) is full or there are no replaceable blocks in the cache set because all block are reserved by prior pending requests (see section Caches for more details). In both cases of hit and miss the round robin fetch scheduler moves to the next warp. In case of hit, the fetched instructions are sent to the decode stage. In the case of a miss a request will be generated by the instruction cache. When the miss response is received the block is filled into the instruction cache and the warp will again need to access the instruction cache. While the miss is pending, the warp does not access the instruction cache.

A warp finishes execution and is not considered by the fetch scheduler anymore if all its threads have finished execution without any outstanding stores or pending writes to local registers. The thread block is considered done once all warps within it are finished and have no pending operations. Once all thread blocks dispatched at a kernel launch finish, then this kernel is considered done.

At the decode stage, the recent fetched instructions are decoded and stored in their corresponding entry in the I-Buffer waiting to be issued.

The simulator software design for this stage is described in Fetch and Decode.

### Instruction Issue

A second round robin arbiter chooses a warp to issue from the I-Buffer to rest of the pipeline. This round robin arbiter is decoupled from the round robin arbiter used to schedule instruction cache accesses. The issue scheduler can be configured to issue multiple instructions from the same warp per cycle. Each valid instruction (i.e. decoded and not issued) in the currently checked warp is eligible for issuing if (1) its warp is not waiting at a barrier, (2) it has valid instructions in its I-Buffer entries (valid bit is set), (3) the scoreboard check passes (see section Scoreboard for more details), and (4) the operand access stage of the instruction pipeline is not stalled.

Memory instructions (Load, store, or memory barriers) are issued to the memory pipeline. For other instructions, it always prefers the SP pipe for operations that can use both SP and SFU pipelines. However, if a control hazard is detected then instructions in the I-Buffer corresponding to this warp are flushed. The warp's next pc is updated to point to the next instruction (assuming all branches as not-taken). For more information about handling control flow, refer to SIMT Stack.

At the issue stage barrier operations are executed. Also, the SIMT stack is updated (refer to SIMT Stack for more details) and register dependencies are tracked (refer to Scoreboard for more details). Warps wait for barriers ("__syncthreads()") at the issue stage.

### SIMT Stack

A per-warp SIMT stack is used to handle the execution of branch divergence on single-instruction, multiple thread (SIMT) architectures. Since divergence reduces the efficiency of these architectures, different techniques can be adapted to reduce this effect. One of the simplest techniques is the post-dominator stack-based reconvergence mechanism. This technique synchronizes the divergent branches at the earliest guaranteed reconvergence point in order to increase the efficiency of the SIMT architecture. Like previous versions of GPGPU-Sim, GPGPU-Sim 3.x adopts this mechanism.

Entries of the SIMT stack represents a different divergence level. At each divergence branch, a new entry is pushed to the top of the stack. The top-of-stack entry is popped when the warp reaches its reconvergence point. Each entry stores the target PC of the new branch, the immediate post dominator reconvergence PC and the active mask of threads that are diverging to this branch. In our model, the SIMT stack of each warp is updated after each instruction issue of this warp. The target PC, in case of no divergence, is normally updated to the next PC. However, in case of divergence, new entries are pushed to the stack with the new target PC, the active mask that corresponds to threads that diverge to this PC and their immediate reconvergence point PC. Hence, a control hazard is detected if the next PC at top entry of the SIMT stack does not equal to the PC of the instruction currently under check.

See Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware for more details.

Note that it is known that NVIDIA and AMD actually modify the contents of their divergence stack using special instructions. These divergence stack instructions are not exposed in PTX but are visible in the actual hardware SASS instruction set (visible using decuda or NVIDIA's cuobjdump). When the current version of GPGPU-Sim 3.x is configured to execute SASS via PTXPlus (see PTX vs. PTXPlus) it ignores these low level instructions and instead a comparable control flow graph is created to identify immediate post-dominators. We plan to support execution of the low level branch instructions in a future version of GPGPU-Sim 3.x.

### Scoreboard

The Scoreboard algorithm checks for WAW and RAW dependency hazards. As explained above, the registers written to by a warp are reserved at the issue stage. The scoreboard algorithm indexed by warp IDs. It stores the required register numbers in an entry that corresponds to the warp ID. The reserved registers are released at the write back stage.

As mentioned above, the decoded instruction of a warp is not scheduled for issue until the scoreboard indicates no WAW or RAW hazards exist. The scoreboard detects WAW and RAW hazards by tracking which registers will be written to by an instruction that has issued but not yet written its results back to the register file.

## Register Access and the Operand Collector

Various NVIDIA patents describe a structure called an "operand collector". The operand collector is a set of buffers and arbitration logic used to provide the appearance of a multiported register file using multiple banks of single ported RAMs. The overall arrangement saves energy and area which is important to improving throughput. Note that AMD also uses banked register files, but the compiler is responsible for ensuring these are accessed so that no bank conflicts occur.

Figure 6 provides an illustration of the detailed way in which GPGPU-Sim 3.x models the operand collector.

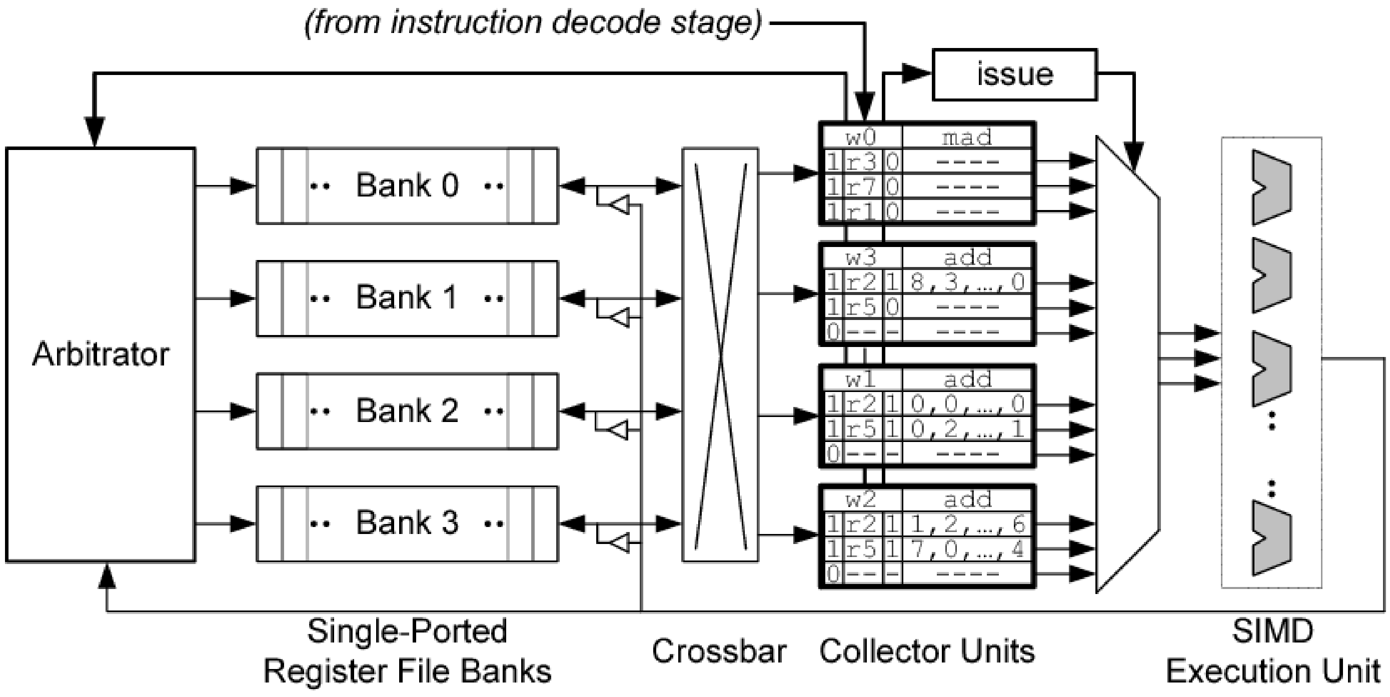


Figure 6: Operand collector microarchitecture

After an instruction is decoded, a hardware unit called a collector unit is allocated to buffer the source operands of the instruction.

The collector units are not used to eliminate name dependencies via register renaming, but rather as a way to space register operand accesses out in time so that no more than one access to a bank occurs in a single cycle. In the organization shown in the figure, each of the four collector units contains three operand entries. Each operand entry has four fields: a valid bit, a register identifier, a ready bit, and operand data. Each operand data field can hold a single 128 byte source operand composed of 32 four byte elements (one four byte value for each scalar thread in a warp). In addition, the collector unit contains an identifier indicating which warp the instruction belongs to. The arbitrator contains a read request queue for each bank to hold access requests until they are granted.

When an instruction is received from the decode stage and a collector unit is available it is allocated to the instruction and the operand, warp, register identifier and valid bits are set. In addition, source operand read requests are queued in the arbiter. To simplify the design, data being written back by the execution units is always prioritized over read requests. The arbitrator selects a group of up to four non-conflicting accesses to send to the register file. To reduce crossbar and collector unit area the selection is made so that each collector unit only receives one operand per cycle.

As each operand is read out of the register file and placed into the corresponding collector unit, a “ready bit” is set. Finally, when all the operands are ready the instruction is issued to a SIMD execution unit.

In our model, each back-end pipeline (SP, SFU, MEM) has a set of dedicated collector units, and they share a pool of general collector units. The number of units available to each pipeline and the capacity of the pool of the general units are configurable.

## ALU Pipelines

GPGPU-Sim v3.x models two types of ALU functional units.

- SP units executes all types of ALU instructions except transcendentals.
- SFU units executes transcendental instructions (Sine, Cosine, Log... etc.).

Both types of units are pipelined and SIMDized. The SP unit can usually execute one warp instruction per cycle, while the SFU unit may only execute a new warp instruction every few cycles, depending on the instruction type. For example, the SFU unit can execute a sine instruction every 4 cycles or a reciprocal instruction every 2 cycles. Different types of instructions also has different execution latency.

Each SIMT core has one SP unit and one SFU unit. Each unit has an independent issue port from the operand collector. Both units share the same output pipeline register that connects to a common writeback stage. There is a result bus allocator at the output of the operand collector to ensure that the units will never be stalled due to the shared writeback. Each instruction will need to allocate a cycle slot in the result bus before being issued to either unit. Notice that the memory pipeline has its own writeback stage and is not managed by this result bus allocator.

The software design section contains more implementation detail of the model.

### Memory Pipeline (LDST unit)

GPGPU-Sim Supports the various memory spaces in CUDA as visible in PTX. In our model, each SIMT core has 4 different on-chip level 1 memories: shared memory, data cache, constant cache, and texture cache. The following table shows which on chip memories service which type of memory access:

| Core Memory | PTX Accesses |
|---|---|
| Shared memory (R/W) | CUDA shared memory (OpenCL local memory) accesses only |
| Constant cache (Read Only) | Constant memory and parameter memory |
| Texture cache (Read Only) | Texture accesses only |
| Data cache (R/W - evict-on-write for global memory, writeback for local memory) | Global and Local memory accesses (Local memory = Private data in OpenCL) |

Although these are modelled as separate physical structures, they are all components of the memory pipeline (LDST unit) and therefore they all share the same writeback stage. The following describes how each of these spaces is serviced:

1. Texture Memory - Accesses to texture memory are cached in the L1 texture cache (reserved for texture accesses only) and also in the L2 cache (if enabled). The L1 texture cache is a special design described in the 1998 paper [3]. Threads on GPU cannot write to the texture memory space, thus the L1 texture cache is read-only.
2. Shared Memory - Each SIMT core contains a configurable amount of shared scratchpad memory that can be shared by threads within a thread block. This memory space is not backed by any L2, and is explicitly managed by the programmer.
3. Constant Memory - Constants and parameter memory is cached in a read-only constant cache.
4. Parameter Memory - See above
5. Local Memory - Cached in the L1 data cache and backed by the L2. Treated in a fashion similar to global memory below except values are written back on eviction since there can be no sharing of local (private) data.
6. Global Memory - Global and local accesses are both serviced by the L1 data cache. Accesses by scalar threads from the same warp are coalesced on a half-warp basis as described in the CUDA 3.1 programming guide. [4]. These accesses are processed at a rate of 2 per SIMT core cycle, such that a memory instruction that is perfectly coalesced into 2 accesses (one per half-warp) can be serviced in a single cycle. For those instructions that generate more than 2 accesses, these will access the memory system at a rate of 2 per cycle. So, if a memory instruction generates 32 accesses (one per lane in the warp), it will take at least 16 SIMT core cycles to move the instruction to the next pipeline stage.

The subsections below describe the first level memory structures.

#### L1 Data Cache

The L1 data cache is a private, per-SIMT core, non-blocking first level cache for local and global memory accesses. The L1 cache is not banked and is able to service two coalesced memory request per SIMT core cycle. An incoming memory request must not span two or more cache lines in the L1 data cache. Note also that the L1 data caches are not coherent.

The table below summarizes the write policies for the L1 data cache.

| L1 data cache write policy | | |
|---|---|---|
| | Local Memory | Global Memory |
| Write Hit | Write-back | Write-evict |
| Write Miss | Write no-allocate | Write no-allocate |

For local memory, the L1 data cache acts as a write-back cache with write no-allocate. For global memory, write hits cause eviction of the block. This mimics the default policy for global stores as outlined in the PTX ISA specification [5].

Memory accesses that hit in the L1 data cache are serviced in one SIMT core clock cycle. Missed accesses are inserted into a FIFO miss queue. One fill request per SIMT clock cycle is generated by the L1 data cache (given the interconnection injection buffers are able to accept the request).

The cache uses Miss Status Holding Registers (MSHR) to hold the status of misses in progress. These are modeled as a fully-associative array. Redundant accesses to the memory system that take place while one request is in flight are merged in the MSHRs. The MSHR table has a fixed number of MSHR entries. Each MSHR entry can service a fixed number of miss requests for a single cache line. The number of MSHR entries and maximum number of requests per entry are configurable.

A memory request that misses in the cache is added to the MSHR table and a fill request is generated if there is no pending request for that cache line. When a fill response to the fill request is received at the cache, the cache line is inserted into the cache and the corresponding MSHR entry is marked as filled. Responses for filled MSHR entries are generated at one request per cycle. Once all the requests waiting at the filled MSHR entry have been responded to and serviced, the MSHR entry is freed.

#### Texture Cache

The texture cache model is a prefetching texture cache. Texture memory accesses exhibit mostly spatial locality and this locality has been found to be mostly captured with about 16 KB of storage. In realistic graphics usage scenarios many texture cache accesses miss. The latency to access texture in DRAM is on the order of many 100's of cycles. Given the large memory access latency and small cache size the problem of when to allocate lines in the cache becomes paramount. The prefetching texture cache solves the problem by temporally decoupling the state of the cache tags from the state of the cache blocks. The tag array represents the state the cache will be in when misses have been serviced 100's of cycles later. The data array represents the state after misses have been serviced. The key to making this decoupling work is to use a reorder buffer to ensure returning texture miss data is placed into the data array in the same order the tag array saw the access. For more details see the original paper.

#### Constant (Read only) Cache

Accesses to constant and parameter memory run through the L1 constant cache. This cache is implemented with a tag array and is like the L1 data cache with the exception that it cannot be written to.

### Thread Block / CTA / Work Group Scheduling

Thread blocks, Cooperative Thread Arrays (CTAs) in CUDA terminology or Work Groups in OpenCL terminology, are issued to SIMT Cores one at a time. Every SIMT Core clock cycle, the thread block issuing mechanism selects and cycles through the SIMT Core Clusters in a round robin fashion. For each selected SIMT Core Cluster, the SIMT Cores are selected and cycled through in a round robin fashion. For every selected SIMT Core, a single thread block will be issued to the core from the selected kernel if the there are enough resources free on that SIMT Core.

If multiple CUDA Streams or command queues are used in the application, then multiple kernels can be executed concurrently in GPGPU-Sim. Different kernels can be executed across different SIMT Cores; a single SIMT Core can only execute thread blocks from a single kernel at a time. If multiple kernels are concurrently being executed, then the selection of the kernel to issue to each SIMT Core is also round robin. Concurrent kernel execution on CUDA architectures is described in the NVIDIA CUDA Programming Guide

## Interconnection Network

Interconnection network is responsible for the communications between SIMT core clusters and Memory Partition units. To simulate the interconnection network we have interfaced the "booksim" simulator to GPGPU-Sim. Booksim is a stand alone network simulator that can be found here. Booksim is capable of simulating virtual channel based Tori and Fly networks and is highly configurable. It can be best understood by referring to "Principles and Practices of Interconnection Networks" book by Dally and Towles.

We refer to our modified version of the booksim as *Intersim*. Intersim has it own clock domain. The original booksim only supports a single interconnection network. We have made some changes to be able to simulate two interconnection networks: one for traffic from the SIMT core clusters to Memory Partitions and one network for traffic from Memory partitions back to SIMT core clusters. This is one way of avoiding circular dependencies that might cause protocol deadlocks in the system. Another way would be having dedicated virtual channels for request and response traffic on a single physical network but this capability is not fully supported in the current version of our public release. Note: A newer version of Booksim (Booksim 2.0) is now available from Stanford, but GPGPU-Sim 3.x does not yet use it.

Please note that SIMT Core Clusters do not directly communicate which each other and hence there is no notion of coherence traffic in the interconnection network. There are only four packet types: (1)read-request and (2)write-requests sent from SIMT core clusters to Memory partitions and (3)read-replys and write-acknowledges sent from Memory Partitions to SIMT Core Clusters.

#### Concentration

SIMT core Clusters act as external concentrators in GPGPU-Sim. From the interconnection network's point of view a SIMT core cluster is a single node and the routers connected to this node only have one injection and one ejection port.

#### Interface with GPGPU-Sim

The interconnection network regardless of its internal configuration provides a simple interface to communicate with SIMT core Clusters and Memory partitions that are connected to it. For injecting packets SIMT core clusters or Memory controllers first check if the network has enough buffer space to accept their packet and then send their packet to the network. For ejection they check if there is packet waiting for ejection in the network and then pop it. These action happen in each units' clock domain. The serialization of packets is handled inside the network interface, e.g. SIMT core cluster injects a packet in SIMT core cluster clock domain but the router accepts only one flit per interconnect clock cycle. More implementation details can be found in the Software design section.

## Memory Partition

The memory system in GPGPU-Sim is modelled by a set of memory partitions. As shown in Figure 7 each memory partition contain an L2 cache bank, a DRAM access scheduler and the off-chip DRAM channel. The functional execution of atomic operations also occurs in the memory partitions in the Atomic Operation Execution phase. Each memory partition is assigned a sub-set of physical memory addresses for which it is responsible. By default the global linear address space is interleaved among partitions in chunks of 256 bytes. This partitioning of the address space along with the detailed mapping of the address space to DRAM row, banks and columns in each partition is configurable and described in the Address Decoding section.

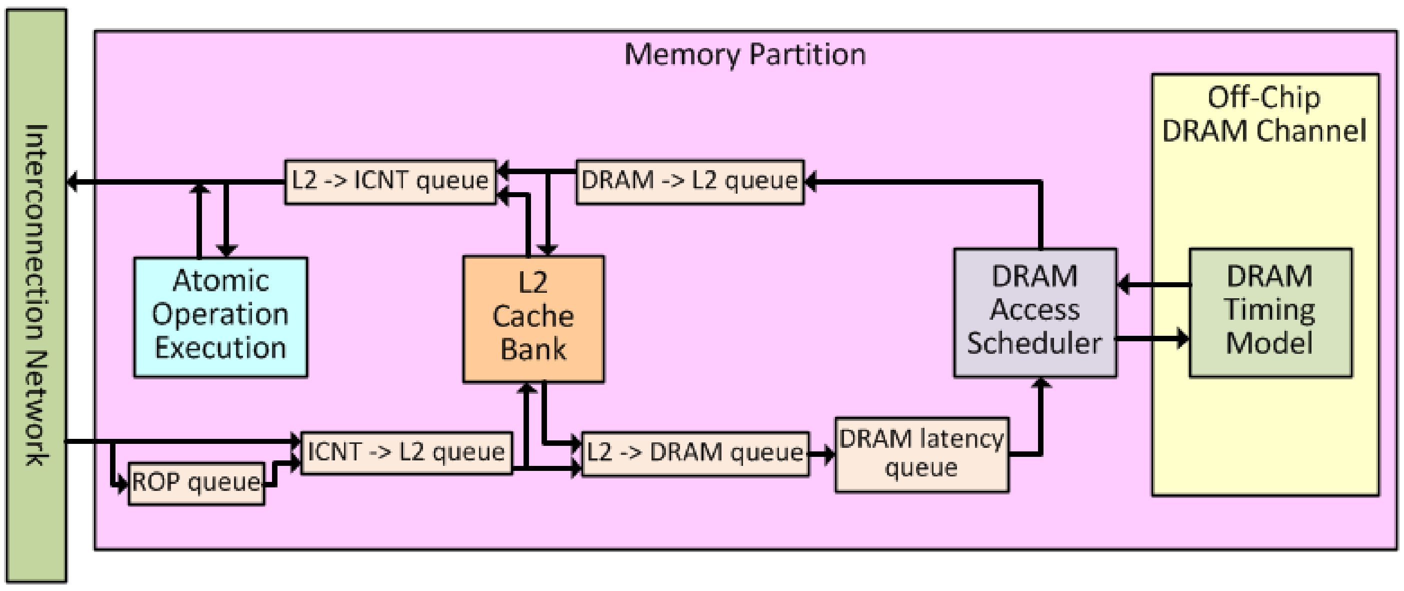


Figure 7: Memory Partition Components

The L2 cache (when enabled) services the incoming texture and (when configured to do so) non-texture memory requests. Note the Quadro FX 5800 (GT200) configuration enables the L2 for texture references only. On a cache miss, the L2 cache bank generates memory requests to the DRAM channel to be serviced by off-chip GDDR3 DRAM.

The subsections below describe in more detail how traffic flows through the memory partition along with the individual components mentioned above.

### Memory Partition Connections and Traffic Flow

Figure 7 above shows the three sub-components inside a single Memory Partition and the various FIFO queues that facilitate the flow of memory requests and responses between them.

The memory request packets enter the Memory Partition from the interconnect via the *ICNT->L2* queue. Non-texture accesses are directed through the *Raster Operations Pipeline (ROP)* queue to model a minimum pipeline latency of 460 L2 clock cycles, as observed by a GT200 micro-benchmarking study. The L2 cache bank pops one request per L2 clock cycle from the *ICNT->L2* queue for servicing. Any memory requests for the off-chip DRAM generated by the L2 are pushed into the *L2->dram* queue. If the L2 cache is disabled, packets are popped from the *ICNT->L2* and pushed directly into the *L2->DRAM* queue, still at the L2 clock frequency. Fill requests returning from off-chip DRAM are popped from *DRAM->L2* queue and consumed by the L2 cache bank. Read replies from the L2 to the SIMT core are pushed through the *L2->ICNT* queue.

The *DRAM latency* queue is a fixed latency queue that models the minimum latency difference between a L2 access and a DRAM access (an access that has missed L2 cache). This latency is observed via micro-benchmarking and this queue simply modeling this observation (instead of the real hardware that causes this delay). Requests exiting the *L2->DRAM* queue reside in the *DRAM latency* queue for a fixed number of SIMT core clock cycles. Each DRAM clock cycle, the DRAM channel can pop memory request from the *DRAM latency* queue to be serviced by off-chip DRAM, and push one serviced memory request into the *DRAM->L2* queue.

Note that ejection from the interconnect to Memory Partition (*ROP* or *ICNT->L2* queues) occurs in L2 clock domain while injection into the interconnect from Memory Partition (*L2->ICNT* queue) occurs in the interconnect (ICNT) clock domain.

### L2 Cache Model and Cache Hierarchy

The L2 cache model is very similar to the L1 data caches in SIMT cores (see that section for more details). When enabled to cache the global memory space data, the L2 acts as a read/write cache with write policies as summarized in the table below.

| L2 cache write policy | | |
|---|---|---|
| | Local Memory | Global Memory |
| Write Hit | Write-back for L1 write-backs | Write-evict |
| Write Miss | Write no-allocate | Write no-allocate |

Additionally, note that the L2 cache is a unified last level cache that is shared by all SIMT cores, whereas the L1 caches are private to each SIMT core.

The private L1 data caches are not coherent (the other L1 caches are for read only address spaces). The cache hierarchy in GPGPU-Sim is non-inclusive non-exclusive. Additionally, a non-decreasing cache line size going down the cache hierarchy (i.e. increasing cache level) is enforced. A memory request from the first level cache also cannot span across two cache lines in the L2 cache. These two restrictions ensure:

1. A request from a lower level cache can be serviced by one cache line in the higher level cache. This ensures that requests from the L1 can be serviced atomically by the L2 cache.
2. Atomic operations do not need to access multiple blocks at the L2

This restriction simplifies the cache design and prevents having to deal with live-lock related issues with servicing a request from L1 non-atomically.

### Atomic Operation Execution Phase

The Atomic Operation Execution phase is a very idealized model of atomic instruction execution. Atomic instructions with non-conflicting memory accesses that were coalesced into one memory request are executed at the memory partition in one cycle. In the performance model, we currently model an atomic operation as a global load operation that skips the L1 data cache. This generates all the necessary register writeback traffic (and data hazard stalls) within the SIMT core. At L2 cache, the atomic operation marks the accessed cache line dirty (changing its status to *modified*) to generate the writeback traffic to DRAM. If L2 cache is not enabled (or used for texture access only), then no DRAM write traffic will be generated for atomic operations (a very idealized model).

### DRAM Scheduling and Timing Model

GPGPU-Sim models both DRAM scheduling and timing. GPGPU-Sim implements two open page mode DRAM schedulers: a FIFO (First In First Out) scheduler and a FR-FCFS (First-Ready First-Come-First-Serve) scheduler, both described below. These can be selected using the configuration option -gpgpu_dram_scheduler.

#### FIFO Scheduler

The FIFO scheduler services requests in the order they are received. This will tend to cause a large number of precharges and activates and hence result in poorer performance especially for applications that generate a large amount of memory traffic relative to the amount of computation they perform.

#### FR-FCFS

The First-Row First-Come-First-Served scheduler gives higher priority to requests to a currently open row in any of the DRAM banks. The scheduler will schedule all requests in the queue to open rows first. If no such request exists it will open a new row for the oldest request. The code for this scheduler is located in dram_sched.h/.cc.

#### DRAM Timing Model

GPGPU-Sim accurately models graphics DRAM memory. Currently GPGPU-Sim 3.x models GDDR3 DRAM, though we are working on adding a detailed GDDR5. The following DRAM timing parameters can be set using the option -gpgpu_dram_timing_opt nbk:tCCD:tRRD:tRCD:tRAS:tRP:tRC:CL:WL:tCDLR:tWR. Currently, we do not model the timing of DRAM refresh operations. Please refer to GDDR3 specifications for more details about each parameter.

- nbk: number of banks
- tCCD: Column to Column Delay (RD/WR to RD/WR different banks)
- tRRD: Row to Row Delay (Active to Active different banks)
- tRCD: Row to Column Delay (Active to RD/WR/RTR/WTR/LTR)
- tRAS: Active to PRECHARGE command period
- tRP: PRECHARGE command period
- tRC: Active to Active command period (same bank)
- CL: CAS Latency
- WL: WRITE latency
- tCDLR: Last data-in to Read Delay (switching from write to read)
- tWR: WRITE recovery time

In our model, commands for each memory bank are scheduled in a round-robin fashion. The banks are arranged in a circular array with a pointer to the bank with the highest priority. The scheduler goes through the banks in order and issues commands. Whenever an activate or precharge command is issued for a bank, the priority pointer is set to the next bank guaranteeing that other pending commands for other banks will be eventually scheduled.

## Instruction Set Architecture (ISA)

### PTX and SASS

GPGPU-Sim simulates the Parallel Thread eXecution (PTX) instruction set used by NVIDIA. PTX is a pseudo-assembly instruction set; i.e. it does not execute directly on the hardware. ptxas is the assembler released by NVIDIA to assemble PTX into the native instruction set run by the hardware (SASS). Each hardware generation supports a different version of SASS. For this reason, PTX is compiled into multiple versions of SASS that correspond to different hardware generations at compile time. Despite that, the PTX code is still embedded into the binary to enable support for future hardware. At runtime, the runtime system selects the appropriate version of SASS to run based on the available hardware. If there is none, the runtime system invokes a just-in-time (JIT) compiler on the embedded PTX to compile it into the SASS corresponding to the available hardware.

### PTXPlus

GPGPU-Sim is capable of running PTX. However, since PTX is not the actual code that runs on the hardware, there is a limit to how accurate it can be. This is mainly due to compiler passes such as strength reduction, instruction scheduling, register allocation to mention a few.

To enable running SASS code in GPGPU-Sim, new features had to be added:

- New addressing modes
- More elaborate condition codes and predicates
- Additional instructions
- Additional datatypes

In order to avoid developing and maintaining two parsers and two functional execution engines (one for PTX and the other for SASS), we chose to extend PTX with the required features in order to provide a one-to-one mapping to SASS. PTX along with the extentions is called PTXPlus. To run SASS, we perform a syntax conversion from SASS to PTXPlus.

PTXPlus has a very similar syntax when compared to PTX with the addition of new addressing modes, more elaborate condition codes and predicates, additional instructions and more data types. It is important to keep in mind that PTXPlus is a superset of PTX, which means that valid PTX is also valid PTXPlus. More details about the exact differences between PTX and PTXPlus can be found in #PTX vs. PTXPlus.

### From SASS to PTXPlus

When the configuration file instructs GPGPU-Sim to run SASS, a conversion tool, cuobjdump_to_ptxplus, is used to convert the SASS embedded within the binary to PTXPlus. For the full details of the conversion process see #PTXPlus Conversion . The PTXPlus is then used in the simulation. When SASS is converted to PTXPlus, only the syntax is changed, the instructions and their order is preserved exactly as in the SASS. Thus, the effect of compiler optimizations applied to the native code is fully captured. Currently, GPGPU-Sim only supports the conversion of GT200 SASS to PTXPlus.

## Using GPGPU-Sim

Refer to the README file in the top level GPGPU-Sim directory for instructions on building and running GPGPU-Sim 3.x. This section provides other important guidance on using GPGPU-Sim 3.x, covering topics such as different simulation modes, how to modify the timing model configuration, a description of the default simulation statistics, and description of approaches for analyzing bugs at the functional level via tracing simulation state and a GDB-like interface. GPGPU-Sim 3.x also provides extensive support for debugging performance simulation bugs including both a high level microarchitecture visualizer and cycle by cycle pipeline state visualization. Next, we describe strategies for debugging GPGPU-Sim when it crashes or deadlocks in performance simulation mode. Finally, it conclude with answers to frequently asked questions.

## Simulation Modes

By default most users will want to use GPGPU-Sim 3.x to estimate the number of GPU clock cycles it takes to run an application. This is known as performance simulation mode. When trying to run a new application on GPGPU-Sim it is always possible that application may not run correctly--i.e., it is possible it may generate the wrong output. To help debugging such applications, GPGPU-Sim 3.x also supports a fast functional simulation only mode. This mode may also be helpful for compiler research and/or when making changes to the functional simulation engine. Orthogonal to the distinction between performance and functional simulation, GPGPU-Sim 3.x also support execution of the native hardware ISA on NVIDIA GPUs (currently GT200 and earlier only), via an extended PTX syntax we call PTXPlus. The following subsections describe these features in turn.

| Modes of operation for GPGPU-Sim | | | | |
|---|---|---|---|---|
| **CUDA Version** | **PTX** | **PTXPlus** | **cuobjdump+PTX** | **cuobjdump+PTXPlus** |
| 2.3 | ? | No | No | No |
| 3.1 | Yes | No | No | No |
| 4.0 | No | No | Yes | Yes |

### Performance Simulation

Performance simulation is the default mode of simulation and collects performance statistics in exchange for slower simulation speed. GPGPU-Sim simulates the microarchitecture described in the Microarchitecture Model section.

To select the performance simulation mode, add the following line to the gpgpusim.config file:

```
-gpgpu_ptx_sim_mode 0
```

For information regarding understanding the simulation output refer to the section on understanding simulation output.

### Pure Functional Simulation

Pure functional simulations run faster than performance simulations but only perform the execution of the CUDA/OpenCL program and does not collect performance statistics.

To select the pure functional simulation mode, add the following line to the gpgpusim.config file:

```
-gpgpu_ptx_sim_mode 1
```

Alternatively, you can set the environmental variable PTX_SIM_MODE_FUNC to 1. Then execute the program normally as you would do in performance simulation mode.

Simulating only the functionality of a GPU device, GPGPU-Sim pure functional simulation mode execute a CUDA/OpenCL program as if it runs on a real GPU device, so no performance measures are collected in this mode, only the regular output of a GPU program is shown. As you expect the pure simulation mode is significantly faster than the performance simulation mode (about 5~10 times faster).

This mode is very useful if you want to quickly check that your code is working correctly on GPGPU-Sim, or if you want to experience using CUDA/OpenCL without the need to have a real GPU computation device. Pure functional simulation supports the same versions of CUDA as the performance simulation (CUDA v3.1) and (CUDA v2.3) for PTX Plus. The pure functional simulation mode execute programs as a group of warps, where warps of each Cooperative Thread Array (CTA) get executed till they all finish or all wait at a barrier, in the latter case once all the warps meet at the barrier they are cleared to go ahead and cross the barrier.

Software design details for Pure Functional Simulation can be found below.

### Interactive Debugger Mode

Interactive debugger mode offers a GDB-like interface for debugging functional behavior in GPGPU-Sim. However, currently it only works with performance simulation.

To enable interactive debugger mode, set environment variable GPGPUSIM_DEBUG to 1. Here are supported commands:

| Command | Description |
|---|---|
| dp <id> | Dump pipeline: Display the state (pipeline content) of the SIMT core <id>. |
| q | Quit |
| b <file>:<line> <thread uid> | Set breakpoint at <file>:<line> for thread with <thread uid>. |
| d <uid> | Delete breakpoint. |
| s | Single step execution to next core cycle for all cores. |
| c | Continue execution without single stepping. |
| w <address> | Set watchpoint at <address>. <address> is specified as a hexadecimal number. |
| l | List PTX around current breakpoint. |

| | |
|---|---|
| h | Display help message. |

It is implemented in files debug.h and debug.cc.

## Cuobjdump Support

As of GPGPU-Sim version 3.1.0, support for using cuobjdump was added. cuobjdump is a software provided by NVidia to extract information like SASS and PTX from binaries. GPGPU-Sim supports using cuobjdump to extract the information it needs to run either SASS or PTX instead of obtaining them through the cubin files. Using cuobjdump is supported only with CUDA 4.0. cuobjdump is enabled by default if the simulator is compiled with CUDA 4.0. To enable/disable cuobjdump, add one of the following option to your configuration file:

```
# disable cuobjdump
-gpgpu_ptx_use_cuobjdump 0
# enable cuobjdump
-gpgpu_ptx_use_cuobjdump 1
```

## PTX vs. PTXPlus

By default, GPGPU-Sim 3.x simulates PTX instructions. However, when executing on an actual GPU, PTX is recompiled to a native GPU ISA (SASS). This recompilation is not fully accounted for in the simulation of normal PTX instructions. To address this issue we created PTXPlus. PTXPlus is an extended form of PTX, introduced by GPGPU-Sim 3.x, that allows for a near 1 to 1 mapping of most GT200 SASS instructions to PTXPlus instructions. It includes new instructions and addressing modes that don't exist in regular PTX. When the conversion to PTXPlus option is activated, the SASS instructions that make up the program are translated into PTXPlus instructions that can be simulated by GPGPU-Sim. Use of the PTXPlus conversion option can lead to significantly more accurate results. However, conversion to PTXPlus does not yet fully support all programs that could be simulated using normal PTX. Currently, only CUDA Toolkit later than 4.0 is supported for conversion to PTXPlus.

SASS is the term NVIDIA uses for the native instruction set used by the GPUs according to their released documentation of the instruction sets. This documentation can be found in the file "cuobjdump.pdf" released with the CUDA Toolkit.

To convert the SASS from an executable, GPGPU-Sim cuobjdump -- a software release along with the CUDA toolkit by NVIDIA that extracts PTX, SASS and other information from CUDA executables. GPGPU-Sim 3.x includes a stand alone program called cuobjdump_to_ptxplus that is invoked to convert the output of cuobjdump into PTXPlus which GPGPU-Sim can simulate. cuobjdump_to_ptxplus is a program written in C++ and its source is provided with the GPGPU-Sim distribution. See the PTXPlus Conversion section for a detailed description on the PTXPlus conversion process. Currently, cuobjdump_to_ptxplus supports the conversion of SASS for sm versions < sm_20.

To enable PTXPlus simulation, add the following line to the gpgpusim.config file:

```
-gpgpu_ptx_convert_to_ptxplus 1
```

Additionally the converted PTXPlus can be saved to files named "_#.ptxplus" by adding the following line to the gpgpusim.config file:

```
-gpgpu_ptx_save_converted_ptxplus 1
```

To turn off either feature, either remove the line or change the value from 1 to 0. More details about using PTXPlus can be found in PTXPlus support. If the option above is enabled, GPGPU-Sim will attempt to convert the SASS code to PTXPlus and then run the resulting PTXPlus. However, as mentioned before, not all programs are supported in this mode.

The subsections below describe the additions we made to PTX to obtain PTXPlus.

### Addressing Modes

To support GT200 SASS, PTXPlus increases the number of addressing modes available to most instructions. Non-load/non-store are now able to directly access memory. The following instruction adds the value in register r0 to the value store in shared memory at address 0x0010 and stores the values in register r1:

```
add.u32 $r1, s[0x0010], $r0;
```

Operands such as s[$r2] or s[$ofs1+0x0010] can also be used. PTXPlus also introduces the following addressing modes that are not present in original PTX:

- g = global memory
- s = shared memory
- ce#c# = constant memory (first number is the kernel number, second number is the constant segment)

```
g[$ofs1+$r0]     //global memory address determined by the sum of register ofs1 and register r0.
s[$ofs1+=0x0010] //shared memory address determined by value in register $ofs1. Register $ofs1 is then incremented by 0x0010.
ce1c2[$ofs1+=$r1] //first kernel's second constant segment's memory address determined by value in register $ofs1. Register $ofs1 is then incremented by the value in register $r1.
```

The implementation details of these addressing modes is described in PTXPlus Implementation.

### New Data Types

Instructions have also been upgraded to more accurately represent how 64 bit and 128 bit values are stored across multiple 32 bit registers. The least significant 32 bits are stored in the far left register while the most significant 32 bits are stored in the far right registers. The following is a list of the new data types and an example of an add instruction adding two 64 bit floating point numbers:

- .ff64 = PTXPlus version of 64 bit floating point number
- .bb64 = PTXPlus version of 64 bit untyped
- .bb128 = PTXPlus version of 128 bit untyped

```
add.rn.ff64 {$r0,$r1}, {$r2,$r3}, {$r4,$r5};
```

### PTXPlus Instructions

| PTXPlus Instructions | |
|---|---|
| nop | Do nothing |
| andn | a andn b = a and ~b |
| norn | a norn b = a nor ~b |
| orn | a orn b = a or ~b |
| nandn | a nandn b = a nand ~b |
| callp | A new call instruction added in PTXPlus. It jumps to the indicated label |
| retp | A new return instruction added in PTXPlus. It jumps back to the instruction after the previous callp instruction |
| breakaddr | Pushes the address indicated by the operand on the thread's break address stack |
| break | Jumps to the address at the top of the thread's break address stack and pops off the entry |

### PTXPlus Condition Codes and Instruction Predication

Instead of the normal true-false predicate system in PTX, SASS instructions use 4-bit condition codes to specify more complex predicate behaviour. As such, PTXPlus uses the same 4-bit predicate system. GPGPU-Sim uses the predicate translation table from decuda for simulating PTXPlus instructions.

The highest bit represents the overflow flag followed by the carry flag and sign flag. The last and lowest bit is the zero flag. Separate condition codes can be stored in separate predicate registers and instructions can indicate which predicate register to use or modify. The following instruction adds the value in register $r0 to the value in register $r1 and stores the result in register $r2. At the same the, the appropriate flags are set in predicate register $p0.

```
add.u32 $p0|$r2, $r0, $r1;
```

Different test conditions can be used on predicated instructions. For example the next instruction is only performed if the carry flag bit in predicate register $p0 is set:

```
@$p0.cf add.u32 $r2, $r0, $r1;
```

### Parameter and Thread ID (tid) Initialization

PTXPlus does not use an explicit parameter state space to store the kernel parameters. Instead, the input parameters are copied in order into shared memory starting at address 0x0010. The copying of parameters is performed during GPGPU-Sim's thread initialization process. The thread initialization process occurs when a thread block is issued to a SIMT core, as described in Thread Block / CTA / Work Group Scheduling. The Kernel Launch: Parameter Hookup section describes the implementation for this procedure. Also during this process, the values of special registers %tid.x, %tid.y and %tid.z are copied into register $r0.

```
Register $r0:
|%tid.z|  %tid.y  |  NA  |  %tid.x  |
 31  26 25     16 15  10 9       0
```

# Debugging via Prints and Traces

There are two built-in facilities for debugging gpgpu-sim. The first mechanism is through environment variables. This is useful for debugging elements of GPGPU-Sim that take place before the configuration file (gpgpusim.config) is parsed, however this can be a clumsy way to implement tracing information in the performance simulator. As of version 3.2.1 GPGPU-Sim includes a tracing system implemented in 'src/trace.h', which allows the user to turn traces on and off via the config file and enable traces by their string name. Both these systems are described below. Please note that many of the environment variable prints could be implemented via the tracing system, but exist as env variables becuase they are in legacy code. Also, GPGPU-Sim prints a large amount of information that is not controlled through the tracing system which is also a result of legacy code.

## Environment Variables for Debugging

Some behavior of GPGPU-Sim 3.x relevant to debugging can be configured via environment variables.

When debugging it may be helpful to generate additional information about what is going on in the simulator and print this out to standard output. This is done by using the following environment variable:

```
export PTX_SIM_DEBUG=<#> enable debugging and set the verbose level
```

The currently supported levels are enumerated below:

| Level | Description |
|---|---|
| 1 | Verbose logs for CTA allocation |
| 2 | Print verbose output from dominator analysis |
| 3 | Verbose logs for GPU malloc/memcpy/memset |
| 5 | Display the instruction executed |
| 6 | Display the modified register(s) by each executed instruction |
| 10 | Display the entire register file of the thread executing the instruction |
| 50 | Print verbose output from control flow analysis |
| 100 | Print the loaded PTX files |

If a benchmark does not run correctly on GPGPU-Sim you may need to debug the functional simulation engine. The way we do this is to print out the functional state of a single thread that generates an incorrect output. To enable printing out functional simulation state for a single thread, use the following environment variable (and set the appropriate level for PTX_SIM_DEBUG):

```
export PTX_SIM_DEBUG_THREAD_UID=<#> ID of thread to debug
```

Other environment configuration options:

```
export PTX_SIM_USE_PTX_FILE=<non-empty-string> override PTX embedded in the binary and revert to old strategy of looking for *.ptx files (good for hand-tweaking PTX)

export PTX_SIM_KERNELFILE=<filename> use this to specify the name of the PTX file
```

### GPGPU-Sim debug tracing

The tracing system is controlled by variables in the gpgpusim.config file:

| Variable | Values | Description |
|---|---|---|
| trace_enabled | 0 or 1 | Globally enable or disable all tracing. If enabled, then trace_components are printed. |
| trace_components | <WARP_SCHEDULER,SCOREBOARD,...> | A comma separated list of tracing elements to enable, a complete list is available in *src/trace_streams.tup* |
| trace_sampling_core | <0 through num_cores-1> | For elements associated with a given shader core (such as the warp scheduler or scoreboard), only print traces from this core |

The Code files that implement the system are:

| Variable | Description |
|---|---|
| *src/trace_streams.tup* | Lists the names of each print stream |
| *src/trace.cc* | Some setup implementation and initialization |
| *src/trace.h* | Defines all the high level interfaces for the tracing system |
| *src/gpgpu-sim/shader_trace.h* | Defines some convenient prints for debugging a specific shader core |

## Configuration Options

Configuration options are passed into GPGPU-Sim with gpgpusim.config and an interconnection configuration file (specified with option -inter_config_file inside gpgpusim.config). GPGPU-Sim 3.0.2 comes with calibrated configuration files in the configs directory for the NVIDIA GT200 (configs/QuadroFX5800/) and Fermi (configs/Fermi/).

Here is a list of the configuration options:

| **Simulation Run Configuration** | |
|---|---|
| **Option** | **Description** |
| -gpgpu_max_cycle <# cycles> | Terminate GPU simulation early after a maximum number of cycle is reached (0 = no limit) |
| -gpgpu_max_insn <# insns> | Terminate GPU simulation early after a maximum number of instructions (0 = no limit) |
| -gpgpu_ptx_sim_mode <0=performance (default), 1=functional> | Select between performance or functional simulation (note that functional simulation may incorrectly simulate some PTX code that requires each element of a warp to execute in lock-step) |
| -gpgpu_deadlock_detect <0=off, 1=on (default)> | Stop the simulation at deadlock |
| -gpgpu_max_cta | Terminates GPU simulation early (0 = no limit) |
| -gpgpu_max_concurrent_kernel | Maximum kernels that can run concurrently on GPU |
| **Statistics Collection Options** | |
| **Option** | **Description** |
| -gpgpu_ptx_instruction_classification <0=off, 1=on (default)> | Enable instruction classification |
| -gpgpu_runtime_stat <frequency>:<flag> | Display runtime statistics |
| -gpgpu_memlatency_stat | Collect memory latency statistics (0x2 enables MC, 0x4 enables queue logs) |
| -visualizer_enabled <0=off, 1=on (default)> | Turn on visualizer output (use AerialVision visualizer tool to plot data saved in log) |
| -visualizer_outputfile <filename> | Specifies the output log file for visualizer. Set to NULL for automatically generated filename (Done by default). |
| -visualizer_zlevel <compression level> | Compression level of the visualizer output log (0=no compression, 9=max compression) |
| -save_embedded_ptx | saves ptx files embedded in binary as <n>.ptx |
| -enable_ptx_file_line_stats <0=off, 1=on (default)> | Turn on PTX source line statistic profiling |
| -ptx_line_stats_filename <output file name> | Output file for PTX source line statistics. |
| -gpgpu_warpdistro_shader | Specify which shader core to collect the warp size distribution from |
| -gpgpu_cflog_interval | Interval between each snapshot in control flow logger |
| -keep | keep intermediate files created by GPGPU-Sim when interfacing with external programs |
| **High-Level Architecture Configuration (See ISPASS paper for more details on what is being modeled)** | |
| **Option** | **Description** |
| -gpgpu_n_mem <# memory controller> | Number of memory controllers (DRAM channels) in this configuration. Read #Topology Configuration before modifying this option. |
| -gpgpu_clock_domains <Core Clock>:<Interconnect Clock>:<L2 Clock>:<DRAM Clock> | Clock domain frequencies in MhZ (See #Clock Domain Configuration) |
| -gpgpu_n_clusters | Number of processing clusters |
| -gpgpu_n_cores_per_cluster | Number of SIMD cores per cluster |
| **Additional Architecture Configuration** | |
| **Option** | **Description** |
| -gpgpu_n_cluster_ejection_buffer_size | Number of packets in ejection buffer |
| -gpgpu_n_ldst_response_buffer_size | Number of response packets in LD/ST unit ejection buffer |
| -gpgpu_coalesce_arch | Coalescing arch (default = 13, anything else is off for now) |
| **Scheduler** | |
| **Option** | **Description** |
| -gpgpu_num_sched_per_core | Number of warp schedulers per core |
| -gpgpu_max_insn_issue_per_warp | Max number of instructions that can be issued per warp in one cycle by scheduler |
| **Shader Core Pipeline Configuration** | |
| **Option** | **Description** |
| -gpgpu_shader_core_pipeline <# thread/shader core>:<warp size>:<pipeline SIMD width> | Shader core pipeline configuration |
| -gpgpu_shader_registers <# registers/shader core, default=8192> | Number of registers per shader core. Limits number of concurrent CTAs. |
| -gpgpu_shader_cta <# CTA/shader core, default=8> | Maximum number of concurrent CTAs in shader |
| -gpgpu_simd_model <1=immediate post-dominator, others are not supported for now> | SIMD Branch divergence handling policy |
| -ptx_opcode_latency_int/fp/dp <ADD,MAX,MUL,MAD,DIV> | Opcode latencies |
| -ptx_opcode_initiation_int/fp/dp <ADD,MAX,MUL,MAD,DIV> | Opcode initiation period. For this number of cycles the inputs of the ALU is held constant. The ALU cannot consume new values during this time. i.e. if this value is 4, then that means this unit can consume new values once every 4 cycles. |
| **Memory Sub-System Configuration** | |
| **Option** | **Description** |
| -gpgpu_perfect_mem <0=off (default), 1=on> | Enable perfect memory mode (zero memory latency with no cache misses) |
| -gpgpu_tex_cache:l1 <nsets>:<bsize>:<assoc>,<rep>:<wr>:<alloc>:<wr_alloc>,<mshr>:<N>:<merge>,<mq>:<fifo_entry> | Texture cache (Read-Only) configuration. Evict policy: L = LRU, F = FIFO, R = Random. |
| -gpgpu_const_cache:l1 <nsets>:<bsize>:<assoc>,<rep>:<wr>:<alloc>:<wr_alloc>,<mshr>:<N>:<merge>,<mq> | Constant cache (Read-Only) configuration. Evict policy: L = LRU, F = FIFO, R = Random |
| -gpgpu_cache:il1 <nsets>:<bsize>:<assoc>,<rep>:<wr>:<alloc>:<wr_alloc>,<mshr>:<N>:<merge>,<mq> | Shader L1 instruction cache (for global and local memory) configuration. Evict policy: L = LRU, F = FIFO, R = Random |
| -gpgpu_cache:dl1 <nsets>:<bsize>:<assoc>,<rep>:<wr>:<alloc>:<wr_alloc>,<mshr>:<N>:<merge>,<mq> -- set to "none" for no DL1 -- | L1 data cache (for global and local memory) configuration. Evict policy: L = LRU, F = FIFO, R = Random |
| -gpgpu_cache:dl2 <nsets>:<bsize>:<assoc>,<rep>:<wr>:<alloc>:<wr_alloc>,<mshr>:<N>:<merge>,<mq> | Unified banked L2 data cache configuration. This specifies the configuration for the L2 cache bank in one of the memory partitions. The total L2 cache capacity = <nsets> x <bsize> x <assoc> x <# memory controller>. |
| -gpgpu_shmem_size <shared memory size, default=16kB> | Size of shared memory per SIMT core (aka shader core) |
| -gpgpu_shmem_warp_parts | Number of portions a warp is divided into for shared memory bank conflict check |
| -gpgpu_flush_cache <0=off (default), 1=on> | Flush cache at the end of each kernel call |
| -gpgpu_local_mem_map | Mapping from local memory space address to simulated GPU physical address space (default = enabled) |
| -gpgpu_num_reg_banks | Number of register banks (default = 8) |
| -gpgpu_reg_bank_use_warp_id | Use warp ID in mapping registers to banks (default = off) |
| -gpgpu_cache:dl2_texture_only | L2 cache used for texture only (0=no, 1=yes, **default=1**) |
| **Operand Collector Configuration** | |
| **Option** | **Description** |
| -gpgpu_operand_collector_num_units_sp | number of collector units (default = 4) |
| -gpgpu_operand_collector_num_units_sfu | number of collector units (default = 4) |
| -gpgpu_operand_collector_num_units_mem | number of collector units (default = 2) |
| -gpgpu_operand_collector_num_units_gen | number of collector units (default = 0) |
| -gpgpu_operand_collector_num_in_ports_sp | number of collector unit in ports (default = 1) |

| | |
|---|---|
| -gpgpu_operand_collector_num_in_ports_sfu | number of collector unit in ports (default = 1) |
| -gpgpu_operand_collector_num_in_ports_mem | number of collector unit in ports (default = 1) |
| -gpgpu_operand_collector_num_in_ports_gen | number of collector unit in ports (default = 0) |
| -gpgpu_operand_collector_num_out_ports_sp | number of collector unit in ports (default = 1) |
| -gpgpu_operand_collector_num_out_ports_sfu | number of collector unit in ports (default = 1) |
| -gpgpu_operand_collector_num_out_ports_mem | number of collector unit in ports (default = 1) |
| -gpgpu_operand_collector_num_out_ports_gen | number of collector unit in ports (default = 0) |
| **DRAM/Memory Controller Configuration** | |
| **Option** | **Description** |
| -gpgpu_dram_scheduler <0 = fifo, 1 = fr-fcfs> | DRAM scheduler type |
| -gpgpu_frfcfs_dram_sched_queue_size <# entries> | DRAM FRFCFS scheduler queue size (0 = unlimited (default); # entries per chip).<br>(Note: FIFO scheduler queue size is fixed to 2). |
| -gpgpu_dram_return_queue_size <# entries> | DRAM requests return queue size (0 = unlimited (default); # entries per chip). |
| -gpgpu_dram_buswidth <# bytes/DRAM bus cycle, default=4 bytes, i.e. 8 bytes/command clock cycle> | Bus bandwidth of a single DRAM chip at command bus frequency (default = 4 bytes (8 bytes per command clock cycle)). The number of DRAM chip per memory controller is set by option -gpgpu_n_mem_per_ctrlr. Each memory partition has (gpgpu_dram_buswidth X gpgpu_n_mem_per_ctrlr) bits of DRAM data bus pins.<br>For example, Quadro FX5800 has a 512-bit DRAM data bus, which is divided among 8 memory partitions. Each memory partition a 512/8=64 bits of DRAM data bus. This 64-bit bus is split into 2 DRAM chips for each memory partition. Each chip will have 32-bit=4-byte of DRAM bus width. We therefore set -gpgpu_dram_buswidth to 4. |
| -gpgpu_dram_burst_length <# burst per DRAM request> | Burst length of each DRAM request (default = 4 data clock cycle, which runs at 2X command clock frequency in GDDR3) |
| -gpgpu_dram_timing_opt <nbk:tCCD:tRRD:tRCD:tRAS:tRP:tRC:CL:WL:tWTR> | DRAM timing parameters:<br>■ nbk = number of banks<br>■ tCCD = CAS to CAS command delay (always = half of burst length)<br>■ tRRD = Row active to row active delay<br>■ tRCD = RAW to CAS delay<br>■ tRAS = Row active time<br>■ tRP = Row precharge time<br>■ tRC = Row cycle time<br>■ CL = CAS latency<br>■ WL = Write latency<br>■ tWTR = Write to read delay |
| -gpgpu_mem_address_mask <address decoding scheme> | **Obsolete**: Select different address decoding scheme to spread memory access across different memory banks. (0 = old addressing mask, 1 = new addressing mask, 2 = new add. mask + flipped bank sel and chip sel bits) |
| -gpgpu_mem_addr_mapping dramid@<start bit>;<memory address map> | Mapping memory address to DRAM model:<br>■ <start bit> = where the bits used to specify the DRAM channel ID starts. (This means the next Log2(#DRAM channel) bits will be used as the DRAM channel ID, and the whole address map will be shifted depending on how many bits are used.)<br>■ <memory address map> = a 64-character string specify how each bit in a memory address is decoded into row (R), column (C), bank (B) addresses. Part of the address that will be inside a single DRAM burst should be specified with (S).<br>See configuration file for Quadro FX 5800 for example. |
| -gpgpu_n_mem_per_ctrlr <# DRAM chips/memory controller> | Number of DRAM chip per memory controller (aka DRAM channel) |
| -gpgpu_dram_partition_queues | i2$:$2d:d2$:$2i |
| -rop_latency <# minimum cycle before L2 cache access> | Specify the minimum latency (in *core clock cycles*) between when a memory request arrived at the memory partition and when it accesses the L2 cache / moves into the queue to access DRAM. It models the minimum L2 hit latency. |
| -dram_latency <# minimum cycle after L2 cache access and before DRAM access> | Specify the minimum latency (in *core clock cycles*) between when a memory request has accessed the L2 cache and when it is pushed into the DRAM scheduler. This option works together with -rop_latency to model the minimum DRAM access latency (= rop_latency + dram_latency). |
| **Interconnection Configuration** | |
| **Option** | **Description** |
| -inter_config_file <Path to Interconnection Config file> | The file containing Interconnection Network simulator's options. For more details about interconnection configurations see Manual provided with the original code at [6]. NOTE that options under "4.6 Traffic" and "4.7 Simulation parameters" should not be used in our simulator. Also see #Interconnection Configuration. |
| -network_mode | Interconnection network mode to be used (default = 1). |
| **PTX Configurations** | |
| **Option** | **Description** |
| -gpgpu_ptx_use_cuobjdump | Use cuobjdump to extract ptx/sass (0=no, 1=yes) Only allowed with CUDA 4.0 |
| -gpgpu_ptx_convert_to_ptxplus | Convert embedded ptx to ptxplus (0=no, 1=yes) |
| -gpgpu_ptx_save_converted_ptxplus | Save converted ptxplus to a file (0=no, 1=yes) |
| -gpgpu_ptx_force_max_capability | Force maximum compute capability (default 0) |
| -gpgpu_ptx_inst_debug_to_file | Dump executed instructions' debug information to a file (0=no, 1=yes) |
| -gpgpu_ptx_inst_debug_file | Output file to dump debug information for executed instructions |
| -gpgpu_ptx_inst_debug_thread_uid | Thread UID for executed instructions' debug output |

## Interconnection Configuration

GPGPU-Sim 3.x uses the booksim router simulator to model the interconnection network. For the most part you will want to consult the booksim documentation for how to configure the interconnect. However, below we list special considerations that need to be taken into account to ensure your modifications work with GPGPU-Sim.

### Topology Configuration

Note that the total number of network nodes as specified in the interconnection network config file should match the total nodes in GPGPU-Sim. GPGPU-Sim's total number of nodes would be the sum of SIMT Core Cluster count and the number of Memory controllers. E.g. in the QuadroFX5800 configuration there are 10 SIMT Core Clusters and 8 Memory Controllers. That is a total of 18 nodes. Therefore in the interconnection config file the network also has 18 nodes as demonstrated below:

```
topology = fly;
k = 18;
n = 1;
routing_function = dest_tag;
```

The configuration snippet above sets up a single stage butterfly network with destination tag routing and 18 nodes. Generally, in both butterfly and mesh networks the total number of network nodes would be k*n.

Note that if you choose to use a mesh network you will want to consider configuring the memory controller placement. In the current release there are a few predefined mappings that can be enabled by setting "use_map=1;" In particular the mesh network used in our ISPASS 2009 paper paper can be configured using this setting and the following topology:

- a 6x6 mesh network (topology=mesh, k=6, n=2) : 28 SIMT cores + 8 dram channels assuming the SIMT core Cluster size is one

You can create your own mappings by modifying create_node_map() in interconnect_interface.cpp (and set use_map=1)

### Booksim options added by GPGPU-Sim

These options are specific to GPGPU-Sim and not part of the original booksim:

- perfect_icnt: if set the interconnect is not simulated all packets that are injected to the network will appear at their destination after one cycle. This is true even when multiple sources send packets to one destination.
- fixed_lat_per_hop: similar to perfect_icnt above except that the packet appears in destination after "Manhattan distance hop count times fixed_lat_per_hop" cycles.
- use_map: changes the way memory and shader cores are placed. See Topology Configuration.
- flit_size: specifies the flit_size in bytes. This is used to identify the number of flits per packet based on the size of packet as passed to icnt_push() functions.
- network_count: Number of independent interconnection networks. Should be set to 2 unless you know what you are doing.
- output_extra_latency: Adds extra cycles to each router. Used to create Figure 10 of ISPASS paper.
- enable_link_stats: prints extra statistics for each link
- input_buf_size: Input buffer size of each node in flits. If left zero the simulator will set it automatically. See "Injecting a packet from the outside world to network"
- ejection_buffer_size: ejection buffer size. If left zero the simulator will set it automatically. See "Ejecting a packet from network to the outside world"
- boundary_buffer_size: boundary buffer size. If left zero the simulator will set it automatically. See "Ejecting a packet from network to the outside world"

These four options were set using #define in original booksim but we have made them configurable via intersim's config file:

- MATLAP_OUTPUT (generates Matlab friendly outputs), DISPLAY_LAT_DIST (shows a distribution of packet latencies), DISPLAY_HOP_DIST (shows a distribution of hop counts), DISPLAY_PAIR_LATENCY (shows average latency for each source destination pair)

### Booksim Options ignored by GPGPU-Sim

Please note the following options that are part of original booksim are either ignored or should not be changed from default.

- Traffic Options (section 4.6 of booksim manual): injection_rate, injection_process, burst_alpha, burst_beta, "const_flit_per_packet", traffic
- Simulation parameters (section 4.7 of booksim manual): sim_type, sample_period, warmup_periods, max_samples, latency_thres, sim_count, reorder

## Clock Domain Configuration

GPGPU-Sim supports four clock domains that can be controlled by the -gpgpu_clock_domains option:

- DRAM clock domain = frequency of the real DRAM clock (command clock) and not the data clock (i.e. 2x of the command clock frequency)
- SIMT Core Cluster clock domain = frequency of the pipeline stages in a core clock (i.e. the rate at which simt_core_cluster::core_cycle() is called)

- Icnt clock domain = frequency of the interconnection network (usually this can be regarded as the *core* clock in NVIDIA GPU specs)
- L2 clock domain = frequency of the L2 cache (We usually set this equal to ICNT clock frequency)

Note that in GPGPU-Sim the width of the pipeline is equal to warp size. To compensate for this we adjust the SIMT Core Cluster clock domain. For example we model the superpipelined stages in NVIDIA's Quadro FX 5800 (GT200) SM running at the fast clock rate (1GHz+) with a single-slower pipeline stage running at 1/4 the frequency. So a 1.3GHz shader clock rate of FX 5800 corresponds to a 325MHz SIMT core clock in GPGPU-Sim.

The DRAM clock domain is specified in the frequency of the command clock. To simplify the peak memory bandwidth calculation, most GPU specs report the data clock, which runs at 2X the command clock frequency. For example, Quadro FX5800 has a memory data clock of 1600MHz, while the command clock is only running at 800MHz. Therefore, our configuration sets the DRAM clock domain at 800.0MHz.

### clock Special Register

In ptx, there is a special register %clock that reads the a clock cycle counter. On the hardware, this register is called SR1. It is a clock cycle counter that silently wraps around. In Quadro, this counter is incremented twice per scheduler clock. In Fermi, it is incremented only once per scheduler clock. GPGPU-Sim will return a value for a counter that is incremented once per scheduler clock (which is the same as the SIMT core clock).

In PTXPlus, the nvidia compiler generates the instructions accessing the %clock register as follows

```
//SASS accessing clock register
S2R R1, SR1
SHL R1, R1, 0x1
//PTXPlus accessing clock register
mov %r1, %clock
shl %r1, %r1, 0x1
```

This basically multiplies the value in the clock register by two. In PTX, however, the clock register is accessed directly. Those conditions must be taken into consideration when calculating results based on the clock register.

```
//PTX accessing clock register
mov r1, %clock
```

## Understanding Simulation Output

At the end of each CUDA grid launch, GPGPU-Sim prints out the performance statistics to the console (stdout). These performance statistics provide insights into how the CUDA application performs with the simulated GPU architecture.

Here is a brief list of the important performance statistics:

### General Simulation Statistics

| Statistic | Description |
|---|---|
| gpu_sim_cycle | Number of cycles (in Core clock) required to execute this kernel. |
| gpu_sim_insn | Number of instructions executed in this kernel. |
| gpu_ipc | gpu_sim_insn / gpu_sim_cycle |
| gpu_tot_sim_cycle | Total number of cycles (in Core clock) simulated for all the kernels launched so far. |
| gpu_tot_sim_insn | Total number of instructions executed for all the kernels launched so far. |
| gpu_tot_ipc | gpu_tot_sim_insn / gpu_tot_sim_cycle |
| gpu_total_sim_rate | gpu_tot_sim_insn / wall_time |

### Simple Bottleneck Analysis

These performance counters track stall events at different high-level parts of the GPU. In combination, they give a broad sense of how where the bottleneck is in the GPU for an application. Figure 8 illustrates a simplified flow of memory requests through the memory sub-system in GPGPU-Sim,

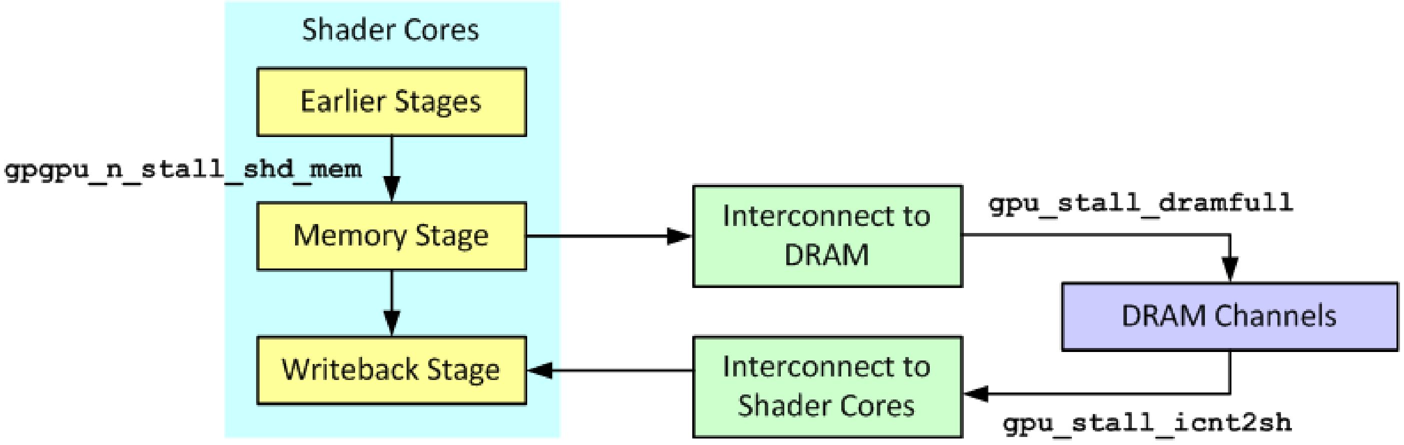


Figure 8: Memory request flow diagram

Here are the description for each counter:

| Statistic | Description |
|---|---|
| gpgpu_n_stall_shd_mem | Number of pipeline stall cycles at the memory stage caused by one of the following reasons:<br>■ shared memory bank conflict<br>■ non-coalesced memory access<br>■ serialized constant memory access |
| gpu_stall_dramfull | Number of cycles that the interconnect outputs to dram channel is stalled. |
| gpu_stall_icnt2sh | Number of cycles that the dram channels are stalled due to the interconnect congestion. |

### Memory Access Statistics

| Statistic | Description |
|---|---|
| gpgpu_n_load_insn | Number of global/local load instructions executed. |
| gpgpu_n_store_insn | Number of global/local store instructions executed. |
| gpgpu_n_shmem_insn | Number of shared memory instructions executed. |
| gpgpu_n_tex_insn | Number of texture memory instructions executed. |
| gpgpu_n_const_mem_insn | Number of constant memory instructions executed. |
| gpgpu_n_param_mem_insn | Number of parameter read instructions executed. |
| gpgpu_n_cmem_portconflict | Number of constant memory bank conflict. |
| maxmrqlatency | Maximum memory queue latency (amount of time a memory request spent in the DRAM memory queue) |
| maxmflatency | Maximum memory fetch latency (round trip latency from shader core to DRAM and back) |
| averagemflatency | Average memory fetch latency |
| max_icnt2mem_latency | Maximum latency for a memory request to traverse from a shader core to the destinated DRAM channel |
| max_icnt2sh_latency | Maximum latency for a memory request to traverse from a DRAM channel back to the specified shader core |

### Memory Sub-System Statistics

| Statistic | Description |
|---|---|
| gpgpu_n_mem_read_local | Number of local memory reads placed on the interconnect from the shader cores. |
| gpgpu_n_mem_write_local | Number of local memory writes placed on the interconnect from the shader cores. |
| gpgpu_n_mem_read_global | Number of global memory reads placed on the interconnect from the shader cores. |
| gpgpu_n_mem_write_global | Number of global memory writes placed on the interconnect from the shader cores. |
| gpgpu_n_mem_texture | Number of texture memory reads placed on the interconnect from the shader cores. |
| gpgpu_n_mem_const | Number of constant memory reads placed on the interconnect from the shader cores. |

### Control-Flow Statistics

GPGPU-Sim reports the warp occupancy distribution which measures performance penalty due to branch divergence in the CUDA application. This information is reported on a single line following the text "Warp Occupancy Distribution:". Alternatively, you may want to grep for W0_Idle. The distribution is display in format: <bin>:<cycle count>. Here is the meaning of each bin:

| Statistic | Description |
|---|---|
| Stall | The number of cycles when the shader core pipeline is stalled and cannot issue any instructions. |
| W0_Idle | The number of cycles when all available warps are issued to the pipeline and are not ready to execute the next instruction. |
| W0_Scoreboard | The number of cycles when all available warps are waiting for data from memory. |
| W*X* (where *X* = 1 to 32) | The number of cycles when a warp with *X* active threads is scheduled into the pipeline. |

Code that has no branch divergence should result in no cycles with W"X" where X is between 1 and 31. See Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware for more detail.

### DRAM Statistics

By default, GPGPU-Sim reports the following statistics for each DRAM channel:

| Statistic | Description |
|---|---|
| n_cmd | Total number of command cycles the memory controller in a DRAM channel has elapsed. The controller can issue a single command per command cycle. |
| n_nop | Total number of NOP commands issued by the memory controller. |
| n_act | Total number of Row Activation commands issued by the memory controller. |

| | |
|---|---|
| n_pre | Total number of Precharge commands issued by the memory controller. |
| n_req | Total number of memory requests processed by the DRAM channel. |
| n_rd | Total number of read commands issued by the memory controller. |
| n_write | Total number of write commands issued by the memory controller. |
| bw_util | DRAM bandwidth utilization = 2 * (n_rd + n_write) / n_cmd |
| n_activity | Total number of active cycles, or command cycles when the memory controller has a pending request at its queue. |
| dram_eff | DRAM efficiency = 2 * (n_rd + n_write) / n_activity (i.e. DRAM bandwidth utilization when there is a pending request waiting to be processed) |
| mrqq: max | Maximum memory request queue occupancy. (i.e. Maximum number of pending entries in the queue) |
| mrqq: avg | Average memory request queue occupancy. (i.e. Average number of pending entries in the queue) |

### Cache Statistics

For each cache (normal data cache, constant cache, texture cache alike), GPGPU-Sim reports the following statistics:

- Access = Total number of access to the cache
- Miss = Total number of misses to the cache. The number in parenthesis is the cache miss rate.
- PendingHit = Total number of pending hits in the cache. An pending hit access has hit a cache line in RESERVED state, which means there is already an inflight memory request sent by a previous cache miss on the same line. This access can be merged that previous memory access so that it does not produce memory traffic. The number in parenthesis is the ratio of accesses that exhibit pending hits.

Notice that pending hits are not counted as cache misses. Also, we do not count pending hits for cache that employs allocate-on-fill policy (e.g. read-only caches such as constant cache and texture cache).

GPGPU-Sim also calculates the total miss rate for all instances of the L1 data cache:

total_dl1_misses

total_dl1_accesses

total_dl1_miss_rate

Notice that data for L1 Total Miss Rate should be ignored when the l1 cache is turned off: -gpgpu_cache:dl1 none

### Interconnect Statistics

In GPGPU-Sim, the user can configure whether to run all traffic on a single interconnection network, or on two separate physical networks (one relaying data from the shader cores to the DRAM channels and the other relaying the data back). (The motivation for using two separate networks, besides increasing bandwidth, is often to avoid "protocol deadlock" which otherwise requires additional dedicated virtual channels.) GPGPU-Sim reports the following statistics for each individual interconnection network:

| Statistic | Description |
|---|---|
| average latency | Average latency for a single flit to traverse from a source node to a destination node. |
| average accepted rate | Measured average throughput of the network relative to its total input channel throughput. Notice that when using two separate networks for traffics in different directions, some nodes will never inject data into the network (i.e. the output only nodes such as DRAM channels on the cores-to-dram network). To get the real ratio, total input channel throughput from these nodes should be ignored. That means one should multiply this rate with the ratio (total # nodes / # input nodes in this network) to get the real average accepted rate. Note that by default we use two separate networks which is set by network_count option in interconnection network config file. The two networks serve to break circular dependancies that might cause deadlocks. |
| min accepted rate | Always 0, as there are nodes that do not inject flits into the network due to the fact that we simulate two separate networks for traffic in different directions. |
| latency_stat_0_freq | A histogram showing the distribution of latency of flits traversed in the network. |

Note: Accepted traffic or throughput of a network is the amount of traffic delivered to the destination terminals of the network. If the network is below saturation all the offered traffic is accepted by the network and offered traffic would be equal to throughput of the network. The interconnect simulator calculates the accepted rate of each node by dividing the total number of packets received at a node by the total network cycles.

## Visualizing High-Level GPGPU-Sim Microarchitecture Behavior

AerialVision is a GPU performance analysis tool for GPGPU-Sim that is includes with the GPGPU-Sim source code introduced in a ISPASS 2010 paper. A detailed manual describing how to use AerialVision can be found here.

Two examples of the type of high level analysis possible with AerialVision are illustrated below. Figure 9 illustrates the use of AerialVision to understand microarchitecture behavior versus time. The top row is average memory access latency versus time, the second row plots load per SIMT core versus time (vertical axis is SIMT core, color represents average instructions per cycle), the bottom row shows load per memory controller channel. Figure 10 illustrates the use of AerialVision to understand microarchitecture behavior at the source code level. This helps identify lines of code associated with uncoalesced or branch divergence.

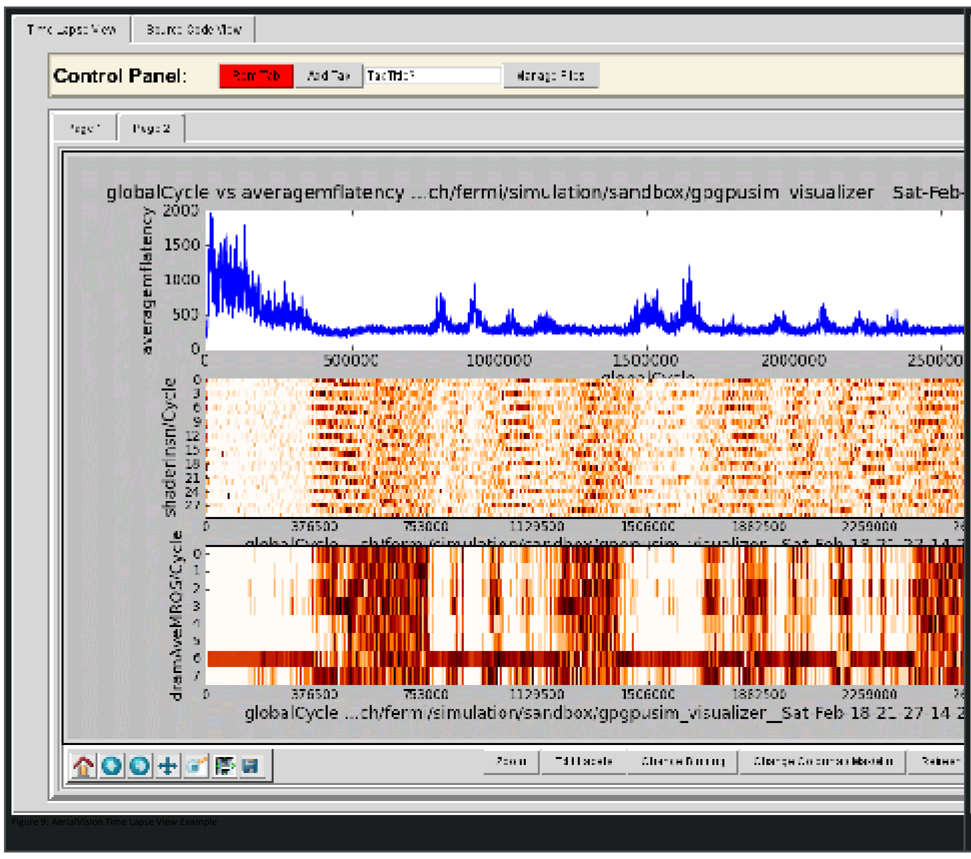


Figure 9: AerialVision Time Lapse View Example

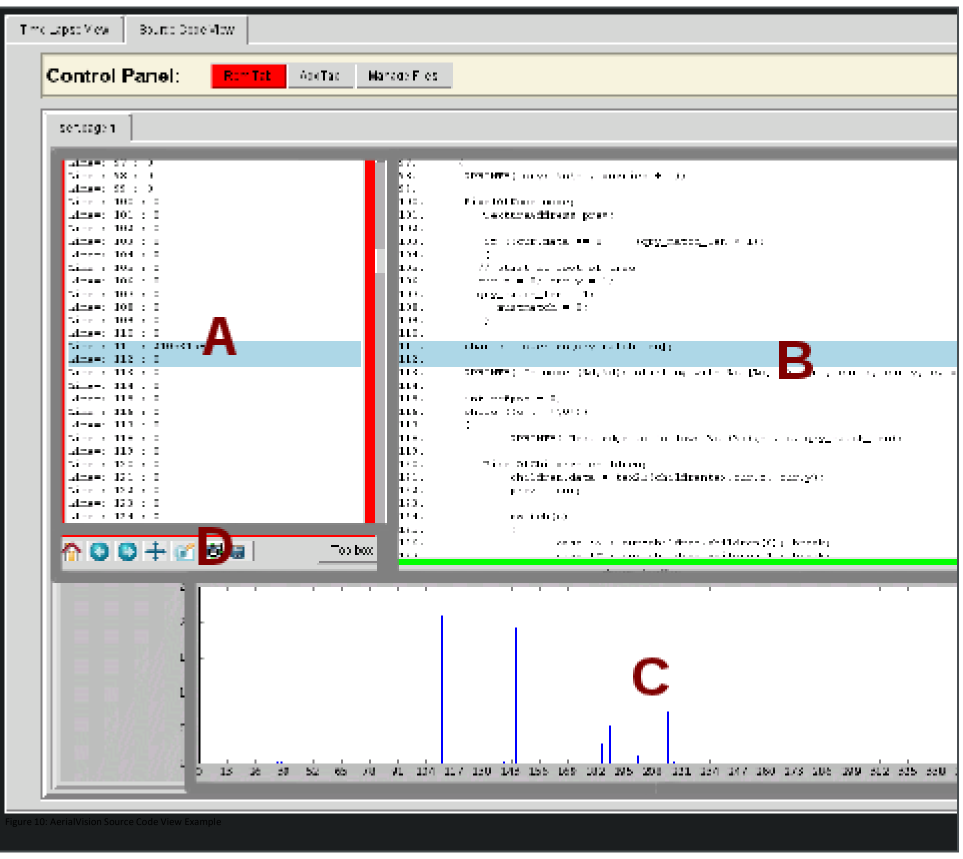


Figure 10: AerialVision Source Code View Example

To get GPGPU-Sim to generate a visualizer log file for the Time Lapse View, add the following option to gpgpusim.config:

```
-visualizer_enabled 1
```

The sampling frequency in this log file can be set by option -gpgpu_runtime_stat. One can also specify the name of the visualizer log file with option -visualizer_outputfile.

To generate the output for the Source Code Viewer, add the following option to gpgpusim.config:

```
-enable_ptx_file_line_stats 1
```

One can specify the output file name with option -ptx_line_stats_filename.

If you use plots generated by AerialVision in your publications, please cite the above linked ISPASS 2010 paper.

## Visualizing Cycle by Cycle Microarchitecture Behavior

GPGPU-Sim 3.x provides a set of GDB macros that can be used to visualize the detail states of each SIMT core and memory partition. By setting the global variable "g_single_step" in GDB to the shader clock cycle at which you would like to start "single stepping" the shader clock, these macros can be used to observe how the microarchitecture states changes cycle-by-cycle. You can use the macros anytime GDB has stopped the simulator, but the global variable "g_single_step" is used in gpu-sim.cc to trigger a call to a hard coded software breakpoint instruction placed after all shader cores have advanced simulation by one cycle. Stopping simulation here tends to lead to more easy to interpret state.

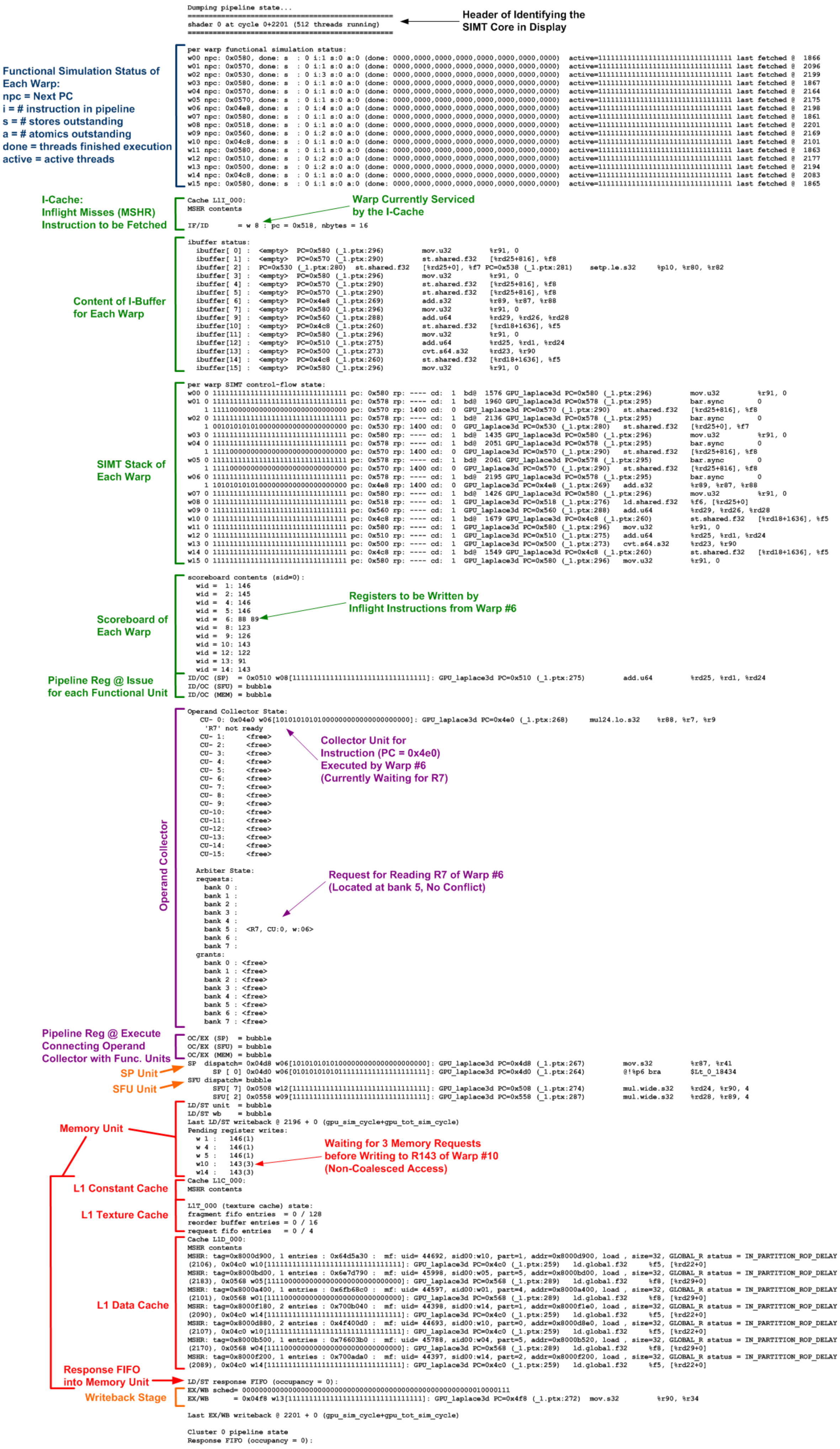
Visibility at this level is useful for debugging and can help you gain deeper insight into how the simulated GPU micro architecture works.

These GDB macros are available in the .gdbinit file that comes with the GPGPU-Sim distribution. To use these macros, either copy the file to your home directory or to the same directory where GDB is launched. GDB will automatically detect the presence of the macro file, load it and display the following message:

```
** loading GPGPU-Sim debugging macros... **
```

| Macro | Description |
|---|---|
| dp <core_id> | Display pipeline state of the SIMT core with ID=<core_id>. See below for an example of the display. |
| dpc <core_id> | Display the pipeline state, then continue to the next breakpoint.<br>This version is useful if you set "g_single_step" to trigger the hard coded breakpoint where gpu_sim_cycle is incremented in gpgpu-sim::cycle() in src/gpgpu-sim/gpu-sim.cc. Repeatedly hitting enter will advance to show the pipeline contents in successive cycles. |
| dm <partition_id> | Display the internal states of a memory partition with ID=<partition_id>. |
| ptxdis <start_pc> <end_pc> | Display PTX instructions between <start_pc> and <end_pc>. |
| ptxdis_func <core_id> <thread_id> | Display all PTX instructions inside the kernel function executed by thread <thread_id> in SIMT core <core_id>. |
| ptx_tids2pcs <thread_ids> <length> <core_id> | Display the current PCs of the threads in SIMT core <core_id> represented by an array <thread_ids> of length=<length>. |

Example of the output by dp:


Header of Identifying the SIMT Core in Display
Functional Simulation Status of Each Warp:
npc = Next PC
i = # instruction in pipeline
s = # stores outstanding
a = # atomics outstanding
done = threads finished execution
active = active threads
I-Cache:
Inflight Misses (MSHR)
Instruction to be Fetched
Warp Currently Serviced by the I-Cache
Content of I-Buffer for Each Warp
SIMT Stack of Each Warp
Scoreboard of Each Warp
Registers to be Written by Inflight Instructions from Warp #6
Pipeline Reg @ Issue for each Functional Unit
Operand Collector
Collector Unit for Instruction (PC = 0x4e0) Executed by Warp #6 (Currently Waiting for R7)
Request for Reading R7 of Warp #6 (Located at bank 5, No Conflict)
Pipeline Reg @ Execute Connecting Operand Collector with Func. Units
SP Unit
SFU Unit
Memory Unit
Waiting for 3 Memory Requests before Writing to R143 of Warp #10 (Non-Coalesced Access)
L1 Constant Cache
L1 Texture Cache
L1 Data Cache
Response FIFO into Memory Unit
Writeback Stage

```
Dumping pipeline state...
==================================================
shader 0 at cycle 0+2201 (512 threads running)
==================================================

per warp functional simulation status:
w00 npc: 0x0580, done: s  : 0 i:1 s:0 a:0 (done: 0000,0000,0000,0000,0000,0000,0000,0000)  active=11111111111111111111111111111111 last fetched @  1866
w01 npc: 0x0570, done: s  : 0 i:1 s:0 a:0 (done: 0000,0000,0000,0000,0000,0000,0000,0000)  active=11111111111111111111111111111111 last fetched @  2096
w02 npc: 0x0530, done: s  : 0 i:3 s:0 a:0 (done: 0000,0000,0000,0000,0000,0000,0000,0000)  active=11111111111111111111111111111111 last fetched @  2199
w03 npc: 0x0580, done: s  : 0 i:1 s:0 a:0 (done: 0000,0000,0000,0000,0000,0000,0000,0000)  active=11111111111111111111111111111111 last fetched @  1867
w04 npc: 0x0570, done: s  : 0 i:1 s:0 a:0 (done: 0000,0000,0000,0000,0000,0000,0000,0000)  active=11111111111111111111111111111111 last fetched @  2164
w05 npc: 0x0570, done: s  : 0 i:1 s:0 a:0 (done: 0000,0000,0000,0000,0000,0000,0000,0000)  active=11111111111111111111111111111111 last fetched @  2175
w06 npc: 0x04e8, done: s  : 0 i:4 s:0 a:0 (done: 0000,0000,0000,0000,0000,0000,0000,0000)  active=11111111111111111111111111111111 last fetched @  2198
w07 npc: 0x0580, done: s  : 0 i:1 s:0 a:0 (done: 0000,0000,0000,0000,0000,0000,0000,0000)  active=11111111111111111111111111111111 last fetched @  1861
w08 npc: 0x0518, done: s  : 0 i:1 s:0 a:0 (done: 0000,0000,0000,0000,0000,0000,0000,0000)  active=11111111111111111111111111111111 last fetched @  2201
w09 npc: 0x0560, done: s  : 0 i:2 s:0 a:0 (done: 0000,0000,0000,0000,0000,0000,0000,0000)  active=11111111111111111111111111111111 last fetched @  2169
w10 npc: 0x04c8, done: s  : 0 i:1 s:0 a:0 (done: 0000,0000,0000,0000,0000,0000,0000,0000)  active=11111111111111111111111111111111 last fetched @  2101
w11 npc: 0x0580, done: s  : 0 i:1 s:0 a:0 (done: 0000,0000,0000,0000,0000,0000,0000,0000)  active=11111111111111111111111111111111 last fetched @  1863
w12 npc: 0x0510, done: s  : 0 i:2 s:0 a:0 (done: 0000,0000,0000,0000,0000,0000,0000,0000)  active=11111111111111111111111111111111 last fetched @  2177
w13 npc: 0x0500, done: s  : 0 i:2 s:0 a:0 (done: 0000,0000,0000,0000,0000,0000,0000,0000)  active=11111111111111111111111111111111 last fetched @  2194
w14 npc: 0x04c8, done: s  : 0 i:1 s:0 a:0 (done: 0000,0000,0000,0000,0000,0000,0000,0000)  active=11111111111111111111111111111111 last fetched @  2083
w15 npc: 0x0580, done: s  : 0 i:1 s:0 a:0 (done: 0000,0000,0000,0000,0000,0000,0000,0000)  active=11111111111111111111111111111111 last fetched @  1865

Cache L1I_000:
MSHR contents

IF/ID       = w 8 : pc = 0x518, nbytes = 16

ibuffer status:
  ibuffer[ 0] :  <empty>  PC=0x580 (_1.ptx:296)         mov.u32         %r91, 0
  ibuffer[ 1] :  <empty>  PC=0x570 (_1.ptx:290)         st.shared.f32   [%rd25+816], %f8
  ibuffer[ 2] :  PC=0x530 (_1.ptx:280)  st.shared.f32   [%rd25+0], %f7 PC=0x538 (_1.ptx:281)    setp.le.s32     %p10, %r80, %r82
  ibuffer[ 3] :  <empty>  PC=0x580 (_1.ptx:296)         mov.u32         %r91, 0
  ibuffer[ 4] :  <empty>  PC=0x570 (_1.ptx:290)         st.shared.f32   [%rd25+816], %f8
  ibuffer[ 5] :  <empty>  PC=0x570 (_1.ptx:290)         st.shared.f32   [%rd25+816], %f8
  ibuffer[ 6] :  <empty>  PC=0x4e8 (_1.ptx:269)         add.s32         %r89, %r87, %r88
  ibuffer[ 7] :  <empty>  PC=0x580 (_1.ptx:296)         mov.u32         %r91, 0
  ibuffer[ 9] :  <empty>  PC=0x560 (_1.ptx:288)         add.u64         %rd29, %rd26, %rd28
  ibuffer[10] :  <empty>  PC=0x4c8 (_1.ptx:260)         st.shared.f32   [%rd18+1636], %f5
  ibuffer[11] :  <empty>  PC=0x580 (_1.ptx:296)         mov.u32         %r91, 0
  ibuffer[12] :  <empty>  PC=0x510 (_1.ptx:275)         add.u64         %rd25, %rd1, %rd24
  ibuffer[13] :  <empty>  PC=0x500 (_1.ptx:273)         cvt.s64.s32     %rd23, %r90
  ibuffer[14] :  <empty>  PC=0x4c8 (_1.ptx:260)         st.shared.f32   [%rd18+1636], %f5
  ibuffer[15] :  <empty>  PC=0x580 (_1.ptx:296)         mov.u32         %r91, 0

per warp SIMT control-flow state:
w00 0 11111111111111111111111111111111 pc: 0x580 rp: ---- cd:  1  bd@  1576 GPU_laplace3d PC=0x580 (_1.ptx:296)         mov.u32         %r91, 0
w01 0 11111111111111111111111111111111 pc: 0x578 rp: ---- cd:  1  bd@  1960 GPU_laplace3d PC=0x578 (_1.ptx:295)         bar.sync        0
    1 11110000000000000000000000000000 pc: 0x570 rp: 1400 cd:  0  GPU_laplace3d PC=0x570 (_1.ptx:290)   st.shared.f32   [%rd25+816], %f8
w02 0 11111111111111111111111111111111 pc: 0x578 rp: ---- cd:  1  bd@  2136 GPU_laplace3d PC=0x578 (_1.ptx:295)         bar.sync        0
    1 00101010101000000000000000000000 pc: 0x530 rp: 1400 cd:  0  GPU_laplace3d PC=0x530 (_1.ptx:280)   st.shared.f32   [%rd25+0], %f7
w03 0 11111111111111111111111111111111 pc: 0x580 rp: ---- cd:  1  bd@  1435 GPU_laplace3d PC=0x580 (_1.ptx:296)         mov.u32         %r91, 0
w04 0 11111111111111111111111111111111 pc: 0x578 rp: ---- cd:  1  bd@  2051 GPU_laplace3d PC=0x578 (_1.ptx:295)         bar.sync        0
    1 11110000000000000000000000000000 pc: 0x570 rp: 1400 cd:  0  GPU_laplace3d PC=0x570 (_1.ptx:290)   st.shared.f32   [%rd25+816], %f8
w05 0 11111111111111111111111111111111 pc: 0x578 rp: ---- cd:  1  bd@  2061 GPU_laplace3d PC=0x578 (_1.ptx:295)         bar.sync        0
    1 11110000000000000000000000000000 pc: 0x570 rp: 1400 cd:  0  GPU_laplace3d PC=0x570 (_1.ptx:290)   st.shared.f32   [%rd25+816], %f8
w06 0 11111111111111111111111111111111 pc: 0x578 rp: ---- cd:  1  bd@  2195 GPU_laplace3d PC=0x578 (_1.ptx:295)         bar.sync        0
    1 10101010101000000000000000000000 pc: 0x4e8 rp: 1400 cd:  0  GPU_laplace3d PC=0x4e8 (_1.ptx:269)   add.s32         %r89, %r87, %r88
w07 0 11111111111111111111111111111111 pc: 0x580 rp: ---- cd:  1  bd@  1426 GPU_laplace3d PC=0x580 (_1.ptx:296)         mov.u32         %r91, 0
w08 0 11111111111111111111111111111111 pc: 0x518 rp: ---- cd:  1  GPU_laplace3d PC=0x518 (_1.ptx:276)   ld.shared.f32   %f6, [%rd25+0]
w09 0 11111111111111111111111111111111 pc: 0x560 rp: ---- cd:  1  GPU_laplace3d PC=0x560 (_1.ptx:288)   add.u64         %rd29, %rd26, %rd28
w10 0 11111111111111111111111111111111 pc: 0x4c8 rp: ---- cd:  1  bd@  1679 GPU_laplace3d PC=0x4c8 (_1.ptx:260)         st.shared.f32   [%rd18+1636], %f5
w11 0 11111111111111111111111111111111 pc: 0x580 rp: ---- cd:  1  GPU_laplace3d PC=0x580 (_1.ptx:296)   mov.u32         %r91, 0
w12 0 11111111111111111111111111111111 pc: 0x510 rp: ---- cd:  1  GPU_laplace3d PC=0x510 (_1.ptx:275)   add.u64         %rd25, %rd1, %rd24
w13 0 11111111111111111111111111111111 pc: 0x500 rp: ---- cd:  1  GPU_laplace3d PC=0x500 (_1.ptx:273)   cvt.s64.s32     %rd23, %r90
w14 0 11111111111111111111111111111111 pc: 0x4c8 rp: ---- cd:  1  bd@  1549 GPU_laplace3d PC=0x4c8 (_1.ptx:260)         st.shared.f32   [%rd18+1636], %f5
w15 0 11111111111111111111111111111111 pc: 0x580 rp: ---- cd:  1  GPU_laplace3d PC=0x580 (_1.ptx:296)   mov.u32         %r91, 0

scoreboard contents (sid=0):
  wid =  1: 146
  wid =  2: 145
  wid =  4: 146
  wid =  5: 146
  wid =  6: 88 89
  wid =  8: 123
  wid =  9: 126
  wid = 10: 143
  wid = 12: 122
  wid = 13: 91
  wid = 14: 143
ID/OC (SP)  = 0x0510 w08[11111111111111111111111111111111]: GPU_laplace3d PC=0x510 (_1.ptx:275)         add.u64         %rd25, %rd1, %rd24
ID/OC (SFU) = bubble
ID/OC (MEM) = bubble

Operand Collector State:
   CU- 0: 0x04e0 w06[10101010101000000000000000000000]: GPU_laplace3d PC=0x4e0 (_1.ptx:268)     mul24.lo.s32    %r88, %r7, %r9
    'R7' not ready
   CU- 1:     <free>
   CU- 2:     <free>
   CU- 3:     <free>
   CU- 4:     <free>
   CU- 5:     <free>
   CU- 6:     <free>
   CU- 7:     <free>
   CU- 8:     <free>
   CU- 9:     <free>
   CU-10:     <free>
   CU-11:     <free>
   CU-12:     <free>
   CU-13:     <free>
   CU-14:     <free>
   CU-15:     <free>

  Arbiter State:
  requests:
    bank 0 :
    bank 1 :
    bank 2 :
    bank 3 :
    bank 4 :
    bank 5 :  <R7, CU:0, w:06>
    bank 6 :
    bank 7 :
  grants:
    bank 0 : <free>
    bank 1 : <free>
    bank 2 : <free>
    bank 3 : <free>
    bank 4 : <free>
    bank 5 : <free>
    bank 6 : <free>
    bank 7 : <free>
OC/EX (SP)  = bubble
OC/EX (SFU) = bubble
OC/EX (MEM) = bubble
SP  dispatch= 0x04d8 w06[10101010101000000000000000000000]: GPU_laplace3d PC=0x4d8 (_1.ptx:267)         mov.s32         %r87, %r41
      SP [ 0] 0x04d0 w06[01010101010111111111111111111111]: GPU_laplace3d PC=0x4d0 (_1.ptx:264)         @!%p6 bra       $Lt_0_18434
SFU dispatch= bubble
      SFU[ 7] 0x0508 w12[11111111111111111111111111111111]: GPU_laplace3d PC=0x508 (_1.ptx:274)         mul.wide.s32    %rd24, %r90, 4
      SFU[ 2] 0x0558 w09[11111111111111111111111111111111]: GPU_laplace3d PC=0x558 (_1.ptx:287)         mul.wide.s32    %rd28, %r89, 4
LD/ST unit  = bubble
LD/ST wb    = bubble
Last LD/ST writeback @ 2196 + 0 (gpu_sim_cycle+gpu_tot_sim_cycle)
Pending register writes:
  w 1 :   146(1)
  w 4 :   146(1)
  w 5 :   146(1)
  w10 :   143(3)
  w14 :   143(3)
Cache L1C_000:
MSHR contents

L1T_000 (texture cache) state:
fragment fifo entries  = 0 / 128
reorder buffer entries = 0 / 16
request fifo entries   = 0 / 4
Cache L1D_000:
MSHR contents
MSHR: tag=0x8000d900, 1 entries : 0x64d5a30 :  mf: uid= 44692, sid00:w10, part=1, addr=0x8000d900, load , size=32, GLOBAL_R status = IN_PARTITION_ROP_DELAY
(2106), 0x04c0 w10[11111111111111111111111111111111]: GPU_laplace3d PC=0x4c0 (_1.ptx:259)   ld.global.f32     %f5, [%rd22+0]
MSHR: tag=0x8000bd00, 1 entries : 0x6e7d790 :  mf: uid= 45998, sid00:w05, part=5, addr=0x8000bd00, load , size=32, GLOBAL_R status = IN_PARTITION_ROP_DELAY
(2183), 0x0568 w05[11110000000000000000000000000000]: GPU_laplace3d PC=0x568 (_1.ptx:289)   ld.global.f32     %f8, [%rd29+0]
MSHR: tag=0x8000a400, 1 entries : 0x6fb68c0 :  mf: uid= 44597, sid00:w01, part=4, addr=0x8000a400, load , size=32, GLOBAL_R status = IN_PARTITION_ROP_DELAY
(2101), 0x0568 w01[11110000000000000000000000000000]: GPU_laplace3d PC=0x568 (_1.ptx:289)   ld.global.f32     %f8, [%rd29+0]
MSHR: tag=0x8000f180, 2 entries : 0x700b040 :  mf: uid= 44398, sid00:w14, part=1, addr=0x8000f1e0, load , size=32, GLOBAL_R status = IN_PARTITION_ROP_DELAY
(2090), 0x04c0 w14[11111111111111111111111111111111]: GPU_laplace3d PC=0x4c0 (_1.ptx:259)   ld.global.f32     %f5, [%rd22+0]
MSHR: tag=0x8000d880, 2 entries : 0x4f400d0 :  mf: uid= 44693, sid00:w10, part=0, addr=0x8000d8e0, load , size=32, GLOBAL_R status = IN_PARTITION_ROP_DELAY
(2107), 0x04c0 w10[11111111111111111111111111111111]: GPU_laplace3d PC=0x4c0 (_1.ptx:259)   ld.global.f32     %f5, [%rd22+0]
MSHR: tag=0x8000b500, 1 entries : 0x76603b0 :  mf: uid= 45788, sid00:w04, part=5, addr=0x8000b520, load , size=32, GLOBAL_R status = IN_PARTITION_ROP_DELAY
(2170), 0x0568 w04[11110000000000000000000000000000]: GPU_laplace3d PC=0x568 (_1.ptx:289)   ld.global.f32     %f8, [%rd29+0]
MSHR: tag=0x8000f200, 1 entries : 0x700ada0 :  mf: uid= 44397, sid00:w14, part=2, addr=0x8000f200, load , size=32, GLOBAL_R status = IN_PARTITION_ROP_DELAY
(2089), 0x04c0 w14[11111111111111111111111111111111]: GPU_laplace3d PC=0x4c0 (_1.ptx:259)   ld.global.f32     %f5, [%rd22+0]

LD/ST response FIFO (occupancy = 0):
EX/WB sched= 00000000000000000000000000000000000000000000000000010000111
EX/WB      = 0x04f8 w13[11111111111111111111111111111111]: GPU_laplace3d PC=0x4f8 (_1.ptx:272)  mov.s32         %r90, %r34

Last EX/WB writeback @ 2201 + 0 (gpu_sim_cycle+gpu_tot_sim_cycle)

Cluster 0 pipeline state
Response FIFO (occupancy = 0):
```

Figure 11: Example pipeline state print out in gdb

## Debugging Errors in Performance Simulation

This section describes strategies for debugging errors that crash or deadlock GPGPU-Sim while running in performance simulation mode.

If you ran a benchmark we haven't tested and it crashed we encourage you to file a bug.

However, suppose GPGPU-Sim crashes or deadlocks after you have made your own changes? What should you do then? This section describes some generic techniques we use at UBC for debugging errors in the GPGPU-Sim timing model.

### Segmentation Faults, Aborts and Failed Assertions

### Deadlocks

## Frequently Asked Questions

**Question:** Which Linux platform is best for GPGPU-Sim?

**Answer:** Currently we use SUSE 11.3 for developing GPGPU-Sim. However, many people ran it successfully on other distributions. In principle, there should be no problems in doing so. We did very minor testing for GPGPU-Sim with Ubuntu 10.04 LTS.

**Question:** Can GPGPU-Sim simulate graphics?

**Answer:** We have some plans to model graphics in the future (no ETA on when that might be available).

**Question:** Is there an option to enable a pure functional simulation without the timing model?

**Answer:** Yes, it is available now. It was removed when we first refactored the GPGPU-Sim 2.x code base into GPGPU-Sim 3.x to simplify the development process, but now it has been reintroduced again.

**Question:** How to change configuration files?

**Answer:** GPGPU-Sim searches for a file called gpgpusim.conf in the current directory. If you need to change the configuration file on the fly, you can create a new directory and create a symbolic link to the configuration file in that directory and use it as your working directory when running GPGPU-Sim. Changing the symbolic link to another file will change the file seen by GPGPU-Sim.

**Question:** Can I run OpenCL applications on GPGPU-Sim if I don't have a GPU? Do I need a GPU to run GPGPU-Sim?

**Answer:** Building and running GPGPU-Sim does not require the presence of a GPU on your system. However, running OpenCL applications requires the NVIDIA Driver which in turn requires the presence of the graphics card. Despite that, we have included support for executing the compilation of OpenCL applications on a remote machine. This means that you only need access to a remote machine that has a graphics card, but the machine you are actually using for running GPGPU-Sim doesn't.

**Question:** I got a parsing error from GPGPU-Sim for an OpenCL program that runs fine on real GPU. What is going on?

**Answer:** Unlike most CUDA applications that contains compiled PTX code in the binary, the kernel code in an OpenCL program are compiled by the video driver at runtime. Depending on the version of the video driver, different versions of PTX may be produced from the same OpenCL kernel. GPGPU-Sim 3.x is developed with NVIDIA driver version 256.40 (see README file that comes with GPGPU-Sim). The newer driver that comes with CUDA Toolkit 4.0 or newer has introduced some new keywords in PTX. Some of these keywords are not difficult to support in GPGPU-Sim (such as .ptr), while others are not as simple (such as the use of %envreg, which is setup by the driver). For now, we would recommend downgrading the video driver, or setup a remote machine with the supported driver version for OpenCL compilation.

**Question:** Does/Will GPGPU-Sim support CUDA 4.0?

**Answer:** Supporting CUDA 4 is something we are currently working on implementing (as usual, no ETA on when it will be released). Using multiple versions of CUDA is not hard with GPGPU-Sim 3.x: The setup_environment script for GPGPU-Sim 3.x can be modified to point to the 3.1 installation so you do not need to modify your .bashrc

**Question:** Why are there two networks (reply and request)?

**Answer:** Those too networks do not necessarily need to be two different physical networks, they can be two different logical networks e.g. each one can use a dedicated set of virtual channels on a single physical network. If the request and reply networks share the same network then a "Protocol Deadlock" may happen. To understand it better read section 14.1.5 of Dally's Interconnection Network book.

**Question:** Is it normal to get 'NaN' in the simulator output?

**Answer:** You may get it with the cache miss rates when the cache module has never been accessed.

**Question:** Why do all CTAs finishes at cycle X, while gpu_sim_cycle says (X + Y)? (i.e. Why is GPGPU-Sim still simulating after all the CTAs/shader cores are done?)

**Answer:** The difference from when a CTA is considered finished by GPGPU-Sim to when GPGPU-Sim thinks the simulation is done can be due to global memory write traffic. Basically, it takes some time from issuing a write command until that command is processed by the memory system.

**Question:** How to calculate the Peak off-chip DRAM bandwidth given a GPGPU-Sim configuration?

**Answer:** Peak off-chip DRAM bandwidth = gpgpu_n_mem * gpgpu_n_mem_per_ctrlr * gpgpu_dram_buswidth * DRAM Clock * 2 (for DDR)

- gpgpu_n_mem = Number of memory channels in the GPU (each memory channel has an independent controller for DRAM command scheduling)
- gpgpu_n_mem_per_ctrlr = Number of DRAM chips attached to a memory channel (default = 2, for 64-bit memory channel)
- gpgpu_dram_buswidth = Bus width of each DRAM chip (default = 32-bit = 4 bytes)
- DRAM Clock = the real clock of the DRAM chip (as opposed to the effective clock used in marketing - See #Clock Domain Configuration)

**Question:** How to get the DRAM utilization?

**Answer:** Each memory controller prints out some statistics at the end of the simulation using "dram_print()". DRAM utilization is "bw_util". Take the average of this number across all the memory controllers (the number for each controller can differ if each DRAM channel gets a different amount of memory traffic).

Inside the simulator's code, 'bwutil' is incremented by 2 for every read or write operation because it takes two DRAM command cycles to service a single read or write operation (given burst length = 4).

**Question:** Why isn't DRAM utilization improving with more shader cores (with the same number of DRAM channels) for a memory-limited application?

**Answer:** DRAM utilization may not improve with having more inflight threads for many reasons. One reason could the DRAM precharge/activate overheads. (See e.g., Complexity Effective Memory Access Scheduling for Many-Core Accelerator Architectures)

**Question:** How to get the interconnect utilization?

**Answer:** The definition of the interconnect's untilization highly depends on the topology of the interconnection network itself, so it is quite difficult to give a single "utilization" metric that is consistent across all types of topology. If you are looking into whether the interconnection is the bottleneck of an application, you may want to look at gpu_stall_icnt2sh and gpu_stall_sh2icnt instead.

The throughput (accepted rate) is also a good indicator for the utilization of each network. Note that by default we use two separate networks for traffics from SIMT core clusters to memory partitions and the traffics heading back; therefore you will see two accepted rate numbers reported at the end of simulation (one for each network). See #Interconnect Statistics for more detail.

**Question:** Does/Will GPGPU-Sim 3.x support DWF (dynamic warp formation) or TBC (thread block compaction)?

**Answer:** The current release of GPGPU-Sim 3.x does not support DWF nor TBC. For now, only PDOM is supported. We are currently working on implementing TBC on GPGPU-Sim 3.x (evaluations of TBC in the HPCA 2011 paper was done on GPGPU-Sim 2.x). While we do not have any plan to release the implementation yet, it may be released under a separate branch in the future (this depends on how modular the final implementation is).

**Question:** Why this simulator is claimed to be timing accurate/cycle accurate? How can I verify this fact?

**Answer:** A cycle-accurate simulator reports the timing behavior of the simulated architecture - it is possible for the user to stop the simulator at cycle boundaries and observe the states. All the hardware behavior within a cycle is approximated with C/C++ (as opposed to implementing them in HDLs) to speed up the simulation time. It is also common for architectural simulator to simplify some detailed implementations covering corner cases of a hardware design to emphasize what dictates the overall performance of a system - this is what we try to achieve with GPGPU-Sim.

So, like all other cycle-accurate simulators used for architectural research/development, we do not guarantee 100% matching with real GPUs. The normal way to verify a simulator would involve comparing reported timing result of an application running on the simulator against measured runtime of the same application running on the actual hardware simulation target. With PTX-ISA, this is a little tricky, because PTX-ISA is recompiled by the GPU driver into native GPU ISA for execution on the actual GPU, whereas GPGPU-Sim execute PTX-ISA directly. Also, the limited amount of publicly available information on the actual NVIDIA GPU microarchitecture has posed a big challenge on implementing the exact matching behavior in the simulator. (i.e. We do not know what is actually implemented inside a GPU. We just implement our best guess in the simulator!)

Nevertheless, we have been continually trying to improve the accuracy of our architecture model. In our ISPASS paper in 2009, we have compared simulated timing performance of various benchmarks against their hardware runtime on a GeForce 8600GT. The correlation coefficient was calculated to be 0.899. GPGPU-Sim 3.0 has been calibrated against an NVIDIA GT 200 GPU and shows IPC correlation of 0.976 on a subset of applications from the NVIDIA CUDA SDK. We welcome feedback from users regarding the accuracy of GPGPU-Sim.

## Software Design of GPGPU-Sim

To perform substantial architecture research with GPGPU-Sim 3.x you will need to modify the source code. This section documents the high level software design of GPGPU-Sim 3.x, which differs from version 2.x. In addition to the software descriptions found here you may find it helpful to consult the doxygen generated documentation for GPGPU-Sim 3.x. Please see the README file for instructions for building the doxygen documentation from the GPGPU-Sim 3.x source.

Below we summarize the source organization, command line option parser, object oriented abstract hardware model that provides an interface between the software organization of the performance simulation engine and the software organization of the functional simulation engine. Finally, we describe the software design of the interface with CUDA/OpenCL applications.

## File list and brief description

GPGPU-Sim 3.x consists of three major modules (each located in its own directory):

- **cuda-sim** - The functional simulator that executes PTX kernels generated by NVCC or OpenCL compiler
- **gpgpu-sim** - The performance simulator that simulates the timing behavior of a GPU (or other many core accelerator architectures)
- **intersim** - The interconnection network simulator adopted from Bill Dally's BookSim

Here are the files in each module:

### Overall/Utilities

| File name | Description |
|---|---|
| Makefile | Makefile that builds gpgpu-sim and calls other the Makefile in cuda-sim and intersim. |
| abstract_hardware_model.h<br>abstract_hardware_model.cc | Provide a set of classes that interface between functional and timing simulator. |
| debug.h<br>debug.cc | Implements the Interactive Debugger Mode. |
| gpgpusim_entrypoint.c | Contains functions that interface with the CUDA/OpenCL API stub libraries. |
| option_parser.h<br>option_parser.cc | Implements the command-line option parser. |
| stream_manager.h<br>stream_manager.cc | Implements the stream manager to support CUDA streams. |
| tr1_hash_map.h | Wrapper code for std::unordered_map in C++11. Falls back to std::map or GNU hash_map if the compile does not support C++11. |
| .gdb_init | Contains macros that simplify low-level visualization of simulation states with GDB |

### cuda-sim

| File name | Description |
|---|---|
| Makefile | Makefile for cuda-sim. Called by Makefile one level up. |
| cuda_device_print.h<br>cuda_device_print.cc | Implementation to support printf() within CUDA device functions (i.e. calling printf() within GPU kernel functions). Please note that the device printf works only with CUDA 3.1 |
| cuda-math.h | Contains interfaces to CUDA Math header files. |
| cuda-sim.h<br>cuda-sim.cc | Implements the interface between gpgpu-sim and cuda-sim. It also contains a standalone simulator for functional simulation. |
| instructions.h<br>instructions.cc | This is where the emulation code of all PTX instructions is implemented. |
| memory.h<br>memory.cc | Functional memory space emulation. |
| opcodes.def | DEF file that links between various information of each instruction (eg. string name, implementation, internal opcode...) |
| opcodes.h | Defines enum for each PTX instruction. |
| ptxinfo.l<br>ptxinfo.y | Lex and Yacc files for parsing ptxinfo file. (To obtain kernel resource requirement) |
| ptx_ir.h<br>ptx_ir.cc | Static structures in CUDA - kernels, functions, symbols... etc. Also contain code to perform static analysis for extracting immediate-post-dominators from kernels at load time. |
| ptx.l<br>ptx.y | Lex and Yacc files for parsing .ptx files and embedded cubin structure to obtain PTX code of the CUDA kernels |
| ptx_loader.h<br>ptx_loader.cc | Contains functions for loading and parsing and printing PTX and PTX info file |
| ptx_parser.h<br>ptx_parser.cc | Contains functions called by Yacc during parsing which creating infra structures needed for functional and performance simulation |
| ptx_parser_decode.def | Contains token definition of parser used in PTX extraction. |
| ptx_sim.h<br>ptx_sim.cc | Dynamic structures in CUDA - Grids, CTA, threads |
| ptx-stats.h<br>ptx-stats.cc | PTX source line profiler |

### gpgpu-sim

| File name | Description |
|---|---|
| Makefile | Makefile for gpgpu-sim. Called by Makefile one level up. |
| addrdec.h<br>addrdec.cc | Address decoder - Maps a given address to a specific row, bank, column, in a DRAM channel. |
| delayqueue.h | An implementation of a flexible pipelined queue. |
| dram_sched.h<br>dram_sched.cc | FR-FCFS DRAM request scheduler. |
| dram.h<br>dram.cc | DRAM timing model + interface to other parts of gpgpu-sim. |
| gpu-cache.h<br>gpu-cache.cc | Cache model for GPGPU-Sim |
| gpu-misc.h<br>gpu-misc.cc | Contains misc. functionality that is needed by parts of gpgpu-sim |
| gpu-sim.h<br>gpu-sim.cc | Gluing different timing models in GPGPU-Sim into one. It contains implementations to support multiple clock domains and implements the thread block dispatcher. |
| histogram.h<br>histogram.cc | Defines several classes that implement different kinds of histograms. |

| | |
|---|---|
| icnt_wrapper.h<br>icnt_wrapper.c | Interconnection network interface for gpgpu-sim. It provides a completely decoupled interface allows intersim to work as a interconnection network timing simulator for gpgpu-sim. |
| l2cache.h<br>l2cache.cc | Implements the timing model of a memory partition. It also implements the L2 cache and interfaces it with the rest of the memory partition (e.g. DRAM timing model). |
| mem_fetch.h<br>mem_fetch.cc | Defines mem_fetch, a communication structure that models a memory request. |
| mem_fetch_status.tup | Defines the status of a memory request. |
| mem_latency_stat.h<br>mem_latency_stat.cc | Contains various code for memory system statistic collection. |
| scoreboard.h<br>scoreboard.cc | Implements the scoreboard used in SIMT core. |
| shader.h<br>shader.cc | SIMT core timing model. It calls cudu-sim for functional simulation of a particular thread and cuda-sim would return with performance-sensitive information for the thread. |
| stack.h<br>stack.cc | Simple stack used by immediate post-dominator thread scheduler. (deprecated) |
| stats.h | Defines the enums that categorize memory accesses and various stall conditions at the memory pipeline. |
| stat-tool.h<br>stat-tool.cc | Implements various tools for performance measurements. |
| visualizer.h<br>visualizer.cc | Output dynamic statistics for the visualizer |

### intersim

Only files modified from original booksim are listed.

| File name | Description |
|---|---|
| booksim_config.cpp | intersim's configuration options are defined here and given a default value. |
| flit.hpp | Modified to add capability of carrying data to the flits. Flits also know which network they belong to. |
| interconnect_interface.cpp<br>interconnect_interface.h | The interface between GPGPU-Sim and intersim is implemented here. |
| iq_router.cpp<br>iq_router.hpp | Modified to add support for output_extra_latency (Used to create Figure 10 of ISPASS 2009 paper). |
| islip.cpp | Some minor edits to fix an out of array bound error. |
| Makefile | Modified to create a library instead of the standalone network simulator. |
| stats.cpp<br>stats.hpp | Stat collection functions are in this file. We have made some minor tweaks. E.g. a new function called NeverUsed is added that tell if that particular stat is ever updated or not. |
| statwraper.cpp<br>statwraper.h | A wrapper that enables using the stat collection capabilities implemented in Stat class in stats.cpp in C files. |
| trafficmanager.cpp<br>trafficmanager.hpp | Heavily modified from original booksim. Many high level operations are done here. |

## Option Parser

As you modify GPGPU-Sim for your research, you will likely add features that you want to configure differently in different simulations. GPGPU-Sim 3.x provides a generic command-line option parser that allows different software modules to register their options through a simple interface. The option parser is instantiated in gpgpu_ptx_sim_init_perf() in gpgpusim_entrypoint.cc. Options are added in gpgpu_sim_config::reg_options() using the function:

```
void option_parser_register(option_parser_t opp,
                    const char *name,
                    enum option_dtype type,
                    void *variable,
                    const char *desc,
                    const char *defaultvalue);
```

Here is the description for each parameter:

- option_parser_t opp - The option parser identifier.
- const char *name - The string the identify the command-line option.
- enum option_dtype type - Data type of the option. It can be one of the following:
  - int
  - unsigned int
  - long long
  - unsigned long long
  - bool (as int in C)
  - float
  - double
  - c-string (a.k.a. char*)
- void *variable - Pointer to the variable.
- const char *desc - Description of the option as displayed
- const char *defaultvalue - Default value of the option (the string value will be automatically parser). You can set this to NULL for this c-string variables.

Look inside gpgpu-sim/gpu-sim.cc for examples.

The option parser is implemented with the OptionParser class in option_parser.cc (exposed in the C interface as option_parser_t in option_parser.h). Here is the full C interface that is used by the rest of GPGPU-Sim:

- **option_parser_register**() - Create a binding between an option name (a string) and a variable in the simulator. The variable is passed by reference (pointer), and it will be modified when option_parser_cmdline() or option_parser_cfgfile() is called. Notice that each option can only be bind to a single variable.
- **option_parser_cmdline**() - Parse the given command line options. Calls option_parser_cmdline() for option -config <config filename>.
- **option_parser_cfgfile**() - Parse a given file containing the configuration options.
- **option_parser_print**() - Dump all the registered options and their parsed value.

Only one OptionParser object is instantiated in GPGPU-Sim, inside gpgpu_ptx_sim_init_perf() in gpgpusim_entrypoint.cc. This OptionParser object converts the simulation options in gpgpusim.config into variables values that can be accessed within the simulator. Different modules in GPGPU-Sim registers their options into the OptionParser object (i.e. specifying which option corresponds to which variable in the simulator). After that, the simulator calls option_parser_cmdline() to parse the simulation options contained in gpgpusim.config.

Internally, the OptionParser class contains as set of OptionRegistry. OptionRegistry is a template class that uses the >> operator for parsing. Each instance of OptionRegistry is responsible for parsing one option to a particular type of variable. Currently the parser only supports the following data types, but it is possible to extend supports to more complex data types by overloading the >> operator:

- 32-bit/64-bit integers
- floating points (float and doubles)
- booleans
- c-style strings (char*)

## Abstract Hardware Model

The files abstract_hardware_model{.h,.cc} provide a set of classes that interface between functional and timing simulator.

### Hardware Abstraction Model Objects

| Enum Name | Description |
|---|---|
| _memory_space_t | Memory space type (register, local, shared, global, ...) |
| uarch_op_t | Type of operation (ALU_OP, SFU_OP, LOAD_OP, STORE_OP, ...) |
| _memory_op_t | Defines whether instruction accesses (load or store) memory. |
| cudaTextureAddressMode | CUDA texture address modes. It can specify wrapping address mode, clamp to edge address mode, mirror address mode or border address mode. |
| cudaTextureFilterMode | CUDA texture filter modes (point or linear filter mode). |
| cudaTextureReadMode | CUDA texture read mode. Specifies read texture mode as element type or normalized float |
| cudaChannelFormatKind | Data type used by CUDA runtime which specifies channel format. This is an argument of cudaCreateChannelDesc(...). |
| mem_access_type | Different types of accesses in the timing simulator to different types of memories. |
| cache_operator_type | Different types of L1 data cache access behavior provided by PTX |
| divergence_support_t | Different control flow divergence handling model. (post dominator is supported) |
| **Class Name** | **Description** |
| class kernel_info_t | Holds information of a kernel. It contains information like kernel function(function_info), grid and block size and list of active threads inside that kernel (ptx_thread_info). |
| class core_t | Abstract base class of a core for both functional and performance model. shader_core_ctx (the class that implements a SIMT core in timing model) is derived from this class. |
| struct cudaChannelFormatDesc | Channel descriptor structure. It keeps channel format and number of bits of each component. |
| struct cudaArray | Structure for saving data of arrays in GPGPU-Sim. It uses whenever main program calls cuda_malloc, cuda_memcopy, cuda_free and etc. |
| struct textureReference | Data type used by cuda runtime to specifies texture references. |
| class gpgpu_functional_sim_config | Functional simulator configuration options. |
| class gpgpu_t | The top-level class that implements a functional GPU simulator. It contains the functional simulator configuration (gpgpu_functional_sim_config) and holds that actual buffer that implements global/texture memory spaces (instances of class memory_space). It has a set of member functions that provides managements to the simulated GPU memory space (malloc, memcpy, texture-bindin, ...). These member functions are called by the CUDA/OpenCL API implementations. Class gpgpu_sim (the top-level GPU timing simulation model) is derived from this class. |
| struct gpgpu_ptx_sim_kernel_info | Holds properties of a kernel function such as PTX version and target machine. Also holds amount of memory and registers using by that kernel. |
| struct gpgpu_ptx_sim_arg | Holds information of kernel arguments/paramters which is set in cudaSetupArgument(...) and _cl_kernel::bind_args(...) for CUDA and OpenCL respectively. |
| class memory_space_t | Information of a memory space like type of memory and number of banks for this memory space. |

| | |
|---|---|
| class mem_access_t | Contains information of each memory access in the timing simulator. This class has information about type of memory access, requested address, size of data and active masks of threads inside warp accessing the memory.This class is used as one of parameters of mem_fetch class, which basically instantiated for each memory access. This class is for interfacing between two different level of memory and passing through interconnects. |
| struct dram_callback_t | This class is the one who is responsible for atomic operations. The function pointer is set during functional simulation(atom_impl()) to atom_callback(...). During timing simulation this function pointer is being called when in l2cache memory controller pop memory partition unit to interconnect. This function is supposed to compute result of atomic operation and saving it in memory. |
| class inst_t | Base class of all instruction classes. This class contains information about type and size of instruction, address of instruction, inputs and outputs, latency and memory scope (memory_space_t) of the instruction. |
| class warp_inst_t | Data of instructions need during timing simulation. Each instruction (ptx_instruction) which is inherited from warp_inst_t contains data for timing and functional simulation. ptx_instruction is filled during functional simulation. After this level program needs only timing information so it casts ptx_instruction to warp_inst_t (some data is being loosed) for timing simulation. warp_inst_t inherited from inst_t. It holds warp_id, active thread mask inside the warp, list of memory accesses (mem_access_t) and information of threads inside that warp (per_thread_info) |

# GPGPU-sim - Performance Simulation Engine

In GPGPU-Sim 3.x the performance simulation engine is implemented via numerous classes defined and implemented in the files under <gpgpu-sim_root>/src/gpgpu-sim/. These classes are brought together via the top-level class gpgpu_sim, which is derived from gpgpu_t (its functional simulation counter part). In the current version of GPGPU-Sim 3.x, only one instance of gpgpu_sim, g_the_gpu, is present in the simulator. Simulation of multiple GPU simultaneously is not currently supported but may be provided in future versions.

This section describes the various classes in the performance simulation engine. These include a set of software objects that models the microarchitecture described earlier, this section also describes how the performance simulation engine interfaces with the functional simulation engine, how it interfaces with AerialVision, and various non-trivial software designs we employ in the performance simulation engine.

## Performance Model Software Objects

One of the more significant changes in GPGPU-Sim 3.x versus 2.x is the introduction of a C++ (mostly) object oriented design for the performance simulation engine. The high level design of the various classes used to implement the performance simulation engine are described in this subsection. These closely correspond to the hardware blocks described earlier.

### SIMT Core Cluster Class

The SIMT core clusters are modelled by the simt_core_cluster class. This class contains an array of SIMT core objects in m_core. The simt_core_cluster::core_cycle() method simply cycles each of the SIMT cores in order. The simt_core_cluster::icnt_cycle() method pushes memory requests into the SIMT Core Cluster's *response FIFO* from the interconnection network. It also pops the requests from the FIFO and sends them to the appropriate core's instruction cache or LDST unit. The simt_core_cluster::icnt_inject_request_packet(...) method provides the SIMT cores with an interface to inject packets into the network.

### SIMT Core Class

The SIMT core microarchitecture shown in Figure 5 is implemented with the class shader_core_ctx in shader.h/cc. Derived from class core_t (the abstract functional class for a core), this class combines all the different objects that implements various parts of the SIMT core microarchitecture model:

- A collection of shd_warp_t objects which models the simulation state of each warp in the core.
- A SIMT stack, simt_stack object, for each warp to handle branch divergence.
- A set of scheduler_unit objects, each responsible for selecting one or more instructions from its set of warps and issuing those instructions for execution.
- A Scoreboard object for detecting data hazard.
- An opndcoll_rfu_t object, which models an operand collector.
- A set of simd_function_unit objects, which implements the SP unit and the SFU unit (the ALU pipelines).
- A ldst_unit object, which implements the memory pipeline.
- A shader_memory_interface which connects the SIMT core to the corresponding SIMT core cluster. Each memory request goes through this interface to be serviced by one of the memory partitions.

Every core cycle, shader_core_ctx::cycle() is called to simulate one cycle at the SIMT core. This function calls a set of member functions that simulate the core's pipeline stages in reverse order to model the pipelining effect:

- fetch()
- decode()
- issue()
- read_operand()
- execute()
- writeback()

The various pipeline stages are connected via a set of pipeline registers which are pointers to warp_inst_t objects (with the exception of Fetch and Decode, which connects via a ifetch_buffer_t object).

Each shader_core_ctx object refers to a common shader_core_config object when accessing configuration options specific to the SIMT core. All shader_core_ctx objects also link to a common instance of a shader_core_stats object which keeps track of a set of performance measurements for all the SIMT cores.

**Fetch and Decode Software Model**

This section describes the software modelling Fetch and Decode.

The I-Buffer shown in Figure 3 is implemented as an array of shd_warp_t objects inside shader_core_ctx. Each shd_warp_t has a set m_ibuffer of I-Buffer entries (ibuffer_entry) holding a configurable number of instructions (the maximum allowable instructions to fetch in one cycle). Also, shd_warp_t has flags that are used by the schedulers to determine the eligibility of the warp for issue. The decoded instructions stored in an ibuffer_entry as a pointer to a warp_inst_t object. The warp_inst_t holds information about the type of the operation of this instruction and the operands used.

Also, in the fetch stage, the shader_core_ctx::m_inst_fetch_buffer variable acts as a pipeline register between the fetch (instruction cache access) and the decode stage.

If the decode stage is not stalled (i.e. shader_core_ctx::m_inst_fetch_buffer is free of valid instructions), the fetch unit works. The outer for loop implements the round robin scheduler, the last scheduled warp id is stored in m_last_warp_fetched. The first if-statement checks if the warp has finished execution, while inside the second if-statement, the actual fetch from the instruction cache, in case of hit or the memory access generation, in case of miss are done. The second if-statement mainly checks if there are no valid instructions already stored in the entry that corresponds the currently checked warp.

The decode stage simply checks the shader_core_ctx::m_inst_fetch_buffer and start to store the decoded instructions (current configuration decode up to two instructions per cycle) in the instruction buffer entry (m_ibuffer, an object of shd_warp_t::ibuffer_entry) that corresponds to the warp in the shader_core_ctx::m_inst_fetch_buffer.

**Schedule and Issue Software Model**

Within each core, there are a configurable number of scheduler units. The function shader_core_ctx::issue() iterates over these units where each one of them executes scheduler_unit::cycle(), where a round robin algorithm is applied on the warps. In the scheduler_unit::cycle(), the instruction is issued to its suitable execution pipeline using the function shader_core_ctx::issue_warp(). Within this function, instructions are functionally executed by calling shader_core_ctx::func_exec_inst() and the SIMT stack (m_simt_stack[warp_id]) is updated by calling simt_stack::update(). Also, in this function, the warps are held/released due to barriers by shd_warp_t:set_membar() and barrier_set_t::warp_reaches_barrier. On the other hand, registers are reserved by Scoreboard::reserveRegisters() to be used later by the scoreboard algorithm. The scheduler_unit::m_sp_out, scheduler_unit::m_sfu_out, scheduler_unit::m_mem_out points to the first pipeline register between the issue stage and the execution stage of SP, SFU and Mem pipline receptively. That is why they are checked before issuing any instruction to its corresponding pipeline using shader_core_ctx::issue_warp().

**SIMT Stack Software Model**

For each scheduler unit there is an array of SIMT stacks. Each SIMT stack corresponds to one warp. In the scheduler_unit::cycle(), the top of the stack entry for the SIMT stack of the scheduled warp determines the issued instruction. The program counter of the top of the stack entry is normally consistent with the program counter of the next instruction in the I-Buffer that corresponds to the scheduled warp (Refer to SIMT Stack). Otherwise, in case of control hazard, they will not be matched and the instructions within the I-Buffer are flushed.

The implementation of the SIMT stack is in the simt_stack class in shader.h. The SIMT stack is updated after each issue using this function simt_stack::update(...). This function implements the algorithm required at divergence and reconvergence points. Functional execution (refer to Instruction Execution) is performed at the issue stage before updating the SIMT stack. This allows the issue stage to have information of the next pc of each thread, hence, to update the SIMT stack as required.

**Scoreboard Software Model**

The scoreboard unit is instantiated in shader_core_ctx as a member object, and passed to scheduler_unit via reference (pointer). It stores both shader core id and a register table index by the warp ids. This register table stores the number of registers reserved by each warp. The functions Scoreboard::reserveRegisters(...), Scoreboard::releaseRegisters(...) and Scoreboard::checkCollision(...) are used to reserve registers, release register and to check for collision before issuing a warp respectively.

**Operand Collector Software Model**

The operand collector is modeled as one stage in the main pipeline executed by the function shader_core_ctx::cycle(). This stage is represented by the shader_core_ctx::read_operands() function. Refer to ALU Pipeline for more details about the interfaces of the operand collector.

The class opndcoll_rfu_t models the operand collector based register file unit. It contains classes that abstracts the collector unit sets, the arbiter and the dispatch units.

The opndcoll_rfu_t::allocate_cu(...) is responsible to allocate warp_inst_t to a free operand collector unit within its assigned sets of operand collectors. Also it adds a read requests for all source operands in their corresponding bank queues in the arbitrator.

However, opndcoll_rfu_t::allocate_reads(...) processes read requests that do not have conflicts, in other words, the read requests that are in different register banks and do not go to the same operand collector are popped from the arbitrator queues. This accounts for write request priority over read requests.

The function opndcoll_rfu_t::dispatch_ready_cu() dispatches the operand registers of ready operand collectors (with all operands are collected) to the execute stage.

The function opndcoll_rfu_t::writeback( const warp_inst_t &inst ) is called at the write back stage of the memory pipeline. It is responsible to the allocation of writes.

This summarizes the highlights of the main functions used to model the operand collector, however, more details are in the implementations of the opndcoll_rfu_t class in both shader.cc and shader.h.

**ALU Pipeline Software Model**

The timing model of SP unit and SFU unit are mostly implemented in the pipelined_simd_unit class defined in shader.h. The specific classes modelling the units (sp_unit and sfu class) are derived from this class with overridden can_issue() member function to specify the types of instruction executable by the unit.

The SP unit is connected to the operation collector unit via the OC_EX_SP pipeline register; the SFU unit is connected to the operand collector unit via the OC_EX_SFU pipeline register. Both units shares a common writeback stage via the WB_EX pipeline register. To prevent two units from stalling for writeback stage conflict, each instruction going into either unit has to allocate a slot in the result bus (m_result_bus) before it is issued into the destined unit (see shader_core_ctx::execute()).

The following figure provides an overview to how pipelined_simd_unit models the throughput and latency for different types of instruction.

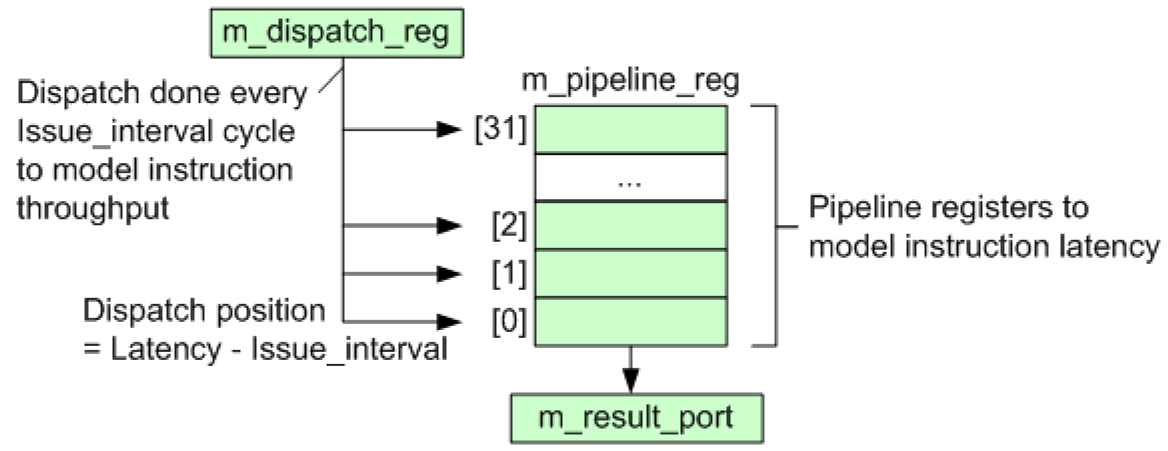

m_dispatch_reg
Dispatch done every Issue_interval cycle to model instruction throughput
m_pipeline_reg
[31]
...
[2]
[1]
[0]
Pipeline registers to model instruction latency
Dispatch position = Latency - Issue_interval
m_result_port

Figure 12: Software Design of Pipelined SIMD Unit

In each pipelined_simd_unit, the issue(warp_inst_t*&) member function moves the contents of the given pipeline registers into m_dispatch_reg. The instruction then waits at m_dispatch_reg for initiation_interval cycles. In the meantime, no other instruction can be issued into this unit, so this wait models the throughput of the instruction. After the wait, the instruction is dispatched to the internal pipeline registers m_pipeline_reg for latency modelling. The dispatching position is determined so that time spent in m_dispatch_reg are accounted towards the latency as well. Every cycle, the instructions will advances through the pipeline registers and eventually into m_result_port, which is the shared pipeline register leading to the common writeback stage for both SP and SFU units.

The throughput and latency of each type of instruction are specified at ptx_instruction::set_opcode_and_latency() in cuda-sim.cc. This function is called during pre-decode.

**Memory Stage Software Model**

The ldst_unit class inside shader.cc implements the memory stage of the shader pipeline. The class instantiates and operates on all the in-shader memories: texture (m_L1T), constant (m_L1C) and data (m_L1D). ldst_unit::cycle() implements the guts of the unit's operation and is pumped m_config->mem_warp_parts times pre core cycle. This is so fully coalesced memory accesses can be processed in one shader cycle. ldst_unit::cycle() processes the memory responses from the interconnect (stored in m_response_fifo), filling the caches and marking stores as complete. The function also cycles the caches so they can send their requests for missed data to the interconnect.

Cache accesses to each type of L1 memory are done in shared_cycle(), constant_cycle(), texture_cycle() and memory_cycle() respectively. memory_cycle is used to access the L1 data cache. Each of these functions then calls process_memory_access_queue() which is a universal function that pulls an access off the instructions internal access queue and sends this request to the cache. If this access cannot be processed in this cycle (i.e. it neither misses nor hits in the cache which can happen when various system queues are full or when all the lines in a particular way have been reserved and are not yet filled) then the access is attempted again next cycle.

It is worth noting that not all instructions reach the writeback stage of the unit. All store instructions and load instructions where all requested cache blocks hit exit the pipeline in the cycle function. This is because they do not have to wait for a response from the interconnect and can by-pass the writeback logic that book-keeps the cache lines requested by the instruction and those that have been returned.

**Cache Software Model**

gpu-cache.h implements all the caches used by the ldst_unit. Both the constant cache and the data cache contain a member tag_array object which implements the reservation and replacement logic. The probe() function checks for a block address without effecting the LRU position of the data in question, while access() is meant to model a look-up that effects the LRU position and is the function that generates the miss and access statistics. MSHR's are modeled with the mshr_table class emulates a fully associative table with a finite number of merged requests. Requests are released from the MSHR through the next_access() function.

The read_only_cache class is used for the constant cache and as the base-class for the data_cache class. This hierarchy can be somewhat confusing because R/W data cache extends from the read_only_cache. The only reason for this is that they share much of the same functionality, with the exception of the access function which deals has to deal with writes in the data_cache. The L2 cache is also implemented with the data_cache class.

The tex_cache class implements the texture cache outlined in the architectural description above. It does not use the tag_array or mshr_table since it's operation is significantly different from that of a conventional cache.

**Thread Block / CTA / Work Group Scheduling**

The scheduling of Thread Blocks to SIMT cores occurs in shader_core_ctx::issue_block2core(...). The maximum number of thread blocks (or CTAs or Work Groups) that can be concurrently scheduled on a core is calculated by the function shader_core_config::max_cta(...). This function determines the maximum number of thread blocks that can be concurrently assigned to a single SIMT core based on the number of threads per thread block specified by the program, the per-thread register usage, the shared memory usage, and the configured limit on maximum number of thread blocks per core. Specifically, the number of thread blocks that could be assigned to a SIMT core if each of the above criteria was the limiting factor is computed. The minimum of these is the maximum number of thread blocks that can be assigned to the SIMT core.

In shader_core_ctx::issue_block2core(...), the thread block size is first padded to be an exact multiple of the warp size. Then a range of free hardware thread ids is determined. The functional state for each thread is initialized by calling ptx_sim_init_thread. The SIMT stacks and warp states are initialized by calling shader_core_ctx::init_warps.

When each thread finishes, the SIMT core calls register_cta_thread_exit(...) to update the active thread block's state. When all threads in a thread block have finished, the same function decreases the count of thread blocks active on the core, allowing more thread blocks to be scheduled in the next cycle. New thread blocks to be scheduled are selected from pending kernels.

### Interconnection Network

The interconnection network interface has a few functions as follows. These function are implemented in interconnect_interface.cpp. These function are wrapped in icnt_wrapper.cpp. The original intention for having icnt_wrapper.cpp was to allow other network simulators to hook up to GPGPU-Sim.

- init_interconnect(): Initialize the network simulator. Its inputs are the interconnection network's configuration file and the number of SIMT core clusters and memory nodes.
- interconnect_push(): which specifies a source node, a destination node, a pointer to the packet to be transmitted and the packet size (in bytes).
- interconnect_pop(): gets a node number as input and it returns a pointer to the packet that was waiting to be ejected at that node. If there is no packet it returns NULL.
- interconnect_has_buffer(): gets a node number and the packet size to be sent as input and returns one(true) if the input buffer of the source node has enough space.
- advance_interconnect(): Should be called every interconnection clock cycle. As name says it perform all the internal steps of the network for one cycle.
- interconnect_busy(): Returns one(true) if there is a packet in flight inside the network.
- interconnect_stats(): Prints network statistics.

### Clock domain crossing for intersim

#### Ejecting a packet from network

We effectively have a two stage buffer per virtual channel at the output, the first stage contains a buffer per virtual channel that has the same space as the buffers internal to the network, the next stage buffer per virtual channel is where we cross from one clock domain to the other--we push flits into the second stage buffer in the interconnect clock domain, and remove whole packets from the second stage buffer in the shader or L2/DRAM clock domain. We return a credit only when we are able to move a flit from the first stage buffer to the second stage buffer (and this occurs at the interconnect clock frequency).

#### Ejection interface details

Here is a more detailed explanation of the clock boundary implementation: At the ejection port of each router we have as many buffers as the number of Virtual Channels. Size of each buffer is exactly equal to VC buffer size. These are the first stage of buffers mentioned above. Let's call the second stage of buffers (again as many as VCs) boundary buffers. These buffers are sized to hold 16-flits each by default (this is a configurable option called boudry_buf_size). When a router tries to eject a flit, the flit is put in the corresponding first stage buffers based on the VC it's coming from ( No credit is sent back yet). Then the boundary buffers are checked to see if they have space; a flit is popped from the corresponding ejection buffer and pushed to the boundary buffer if it has space (this is done for all buffers in the same cycle). At this point the flit is also pushed to a credit return queue. Routers can pop 1 flit per network cycle from this credit return queue and generate its corresponding credit. The shader (or L2/DRAM) side pops the boundary buffer every shader or (DRAM/L2 cycle) and gets a full "Packet". i.e. If the packet is 4 flits it frees up 4 slots in the boundary buffer;if it's 1 flit it only frees up 1 flit. Since boundary buffers are as many as VCs shader (or DRAM) pops them in round robin. (It can only get 1 packet per cycle) In this design the first stage buffer always has space for the flits coming from router and as boundary buffers get full the flow of credits backwards will stop.

Note that the implementation described above is just our way of implementing the interface logic in the simulator and not necessarily the way the network interface is actually implemented in real hardware.

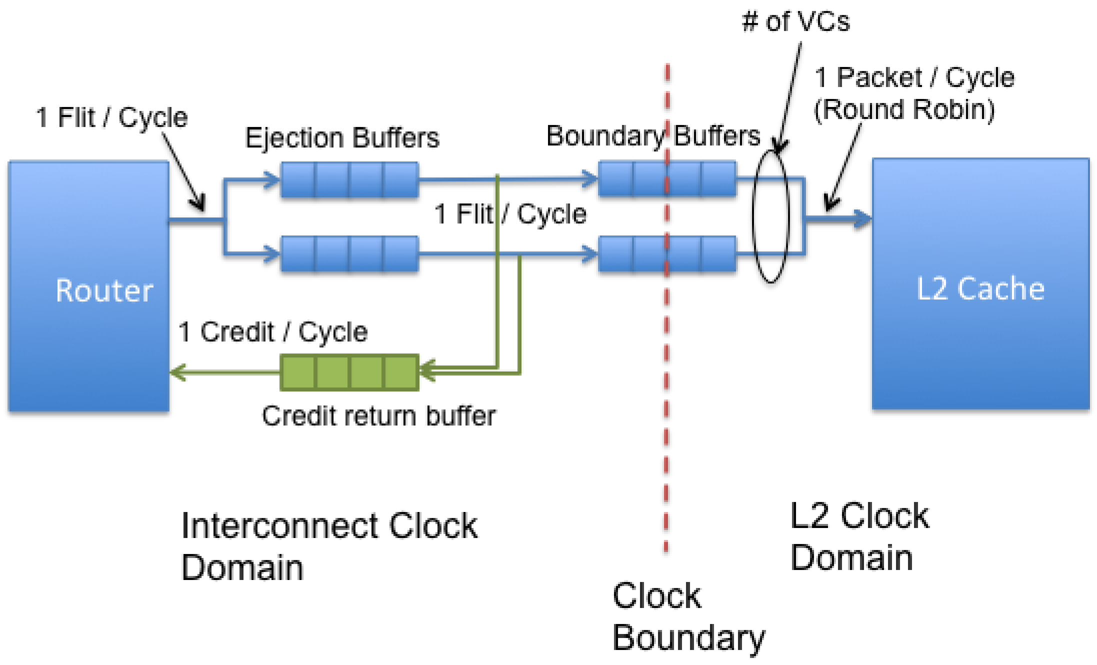


Figure 13: Clock Boundary Implementation

#### Injecting a packet to the network

Each node of the network has an input buffer. This input buffer size is configurable via input_buffer_size option in the interconnect config file. In order to inject a packet into the interconnect first the input buffer capacity is checked by calling interconnect_has_buffer(). If there is enough space the packet will be pushed to interconnect by calling interconnect_push(). These steps are done in the shader clock domain (in the memory stage) and in the interconnect clock domain for memory nodes.

Every-time advance_interconnect() function is called (in the interconnect clock domain) flits are taken out of the input buffer on each node and actually start traveling in the network (if possible).

### Memory Partition

The Memory Partition is modelled by the memory_partition_unit class defined inside l2cache.h and l2cache.cc. These files also define an extended version of the mem_fetch_allocator, partition_mf_allocator, for generation of mem_fetch objects (memory requests) by the Memory Partition and L2 cache.

From the sub-components described in the Memory Partition micro-architecture model section, the member object of type data_cache models the L2 cache and type dram_t the off-chip DRAM channel. The various queues are modelled using the fifo_pipeline class. The minimum latency ROP queue is modelled as a queue of rop_delay_t structs. The rop_delay_t structs store the minimum time at which each memory request can exit the ROP queue (push time + constant ROP delay). The m_request_tracket object tracks all in-flight requests not fully serviced yet by the Memory Partition to determine if the Memory Partition is currently active.

The Atomic Operation Unit does not have have an associated class. This component is modelled simply by functionally executing the atomic operations of memory requests leaving the *L2->icnt queue*. The next section presents further details.

#### Memory Partition Connections and Traffic Flow

The gpgpu_sim::cycle() method clock all the architectural components in GPGPU-Sim, including the Memory Partition's queues, DRAM channel and L2 cache bank.

The code segment

```
::icnt_push( m_shader_config->mem2device(i), mf->get_tpc(), mf, response_size );
m_memory_partition_unit[i]->pop();
```

injects memory requests into the interconnect from the Memory Partition's *L2->icnt* queue. The call to memory_partition_unit::pop() functionally executes atomic instructions. The request tracker also discards the entry for that memory request here indicating that the Memory Partition is done servicing this request.

The call to memory_partition_unit::dram_cycle() moves memory requests from *L2->dram* queue to the DRAM channel, DRAM channel to *dram->L2* queue, and cycles the off-chip GDDR3 DRAM memory.

The call to memory_partition_unit::push() ejects packets from the interconnection network and passes them to the Memory Partition. The request tracker is notified of the request. Texture accesses are pushed directly into the *icnt->L2* queue, while non-texture accesses are pushed into the minimum latency *ROP* queue. Note that the push operations into both the *icnt->L2* and *ROP* queues are throttled by the size of *icnt->L2* queue as defined in the memory_partition_unit::full() method.

The call to memory_partition_unit::cache_cycle() clocks the L2 cache bank and moves requests into or out of the L2 cache. The next section describes the internals of memory_partition_unit::cache_cycle().

#### L2 Cache Model

Inside memory_partition_unit::cache_cycle(), the call

```
mem_fetch *mf = m_L2cache->next_access();
```

generates replies for memory requests waiting in filled MSHR entries, as described in the MSHR description. Fill responses, i.e. response messages to memory requests generated by the L2 on read misses, are passed to the L2 cache by popping from the *dram->L2* queue and calling

```
m_L2cache->fill(mf,gpu_sim_cycle+gpu_tot_sim_cycle);
```

Fill requests that are generated by the L2 due to read misses are popped from the L2's miss queue and pushed into the *L2->dram* queue by calling

```
m_L2cache->cycle();
```

L2 access for memory request exiting the *icnt->L2* queue is done by the call

```
enum cache_request_status status = m_L2cache->access(mf->get_partition_addr(),mf,gpu_sim_cycle+gpu_tot_sim_cycle,events);
```

On a L2 cache hit, a response is immediately generated and pushed into the *L2->icnt* queue. On a miss, no request is generated here as the code internal to the cache class has generated a memory request in its miss queue. If the L2 cache is disabled, then memory requests are pushed straight from the *icnt->L2* queue to the *L2->dram* queue.

Also in memory_partition_unit::cache_cycle(), memory requests are popped from the *ROP* queue and inserted into the *icnt->L2* queue.

#### DRAM Scheduling and Timing Model

The DRAM timing model is implemented in the files dram.h and dram.cc. The timing model also includes an implementation of a FIFO scheduler. The more complicated FRFCFS scheduler is located in dram_sched.h and dram_sched.cc.

The function dram_t::cycle() represents a DRAM cycle. In each cycle, the DRAM pops a request from the request queue then calls the scheduler function to allow the scheduler to select a request to be serviced based on the scheduling policy. Before the requests are sent to the scheduler, they wait in the *DRAM latency* queue for a fixed number of SIMT core cycles. This functionality is also implemented inside dram_t::cycle().

```
case DRAM_FIFO: scheduler_fifo(); break;
case DRAM_FRFCFS: scheduler_frfcfs(); break;
```

The DRAM timing model then checks if any bank is ready to issue a new request based on the different timing constraints specified in the configuration file. Those constraints are represented in the DRAM model by variables similar to this one

```
unsigned int CCDc; //Column to Column Delay
```

Those variables are decremented at the end of each cycle. An action is only taken when all of its constraint variables have reached zero. Each taken action resets a set of constraint variables to their original configured values. For example, when a column is activated, the variable CCDc is reset to its original configured value, then decremented by one every cycle. We cannot scheduler a new column until this variable reaches zero. The Macro DEC2ZERO decrements a variable until it reaches zero, and then it keeps it at zero until another action resets it.

## Interface between CUDA-Sim and GPGPU-Sim

The timing simulator (GPGPU-Sim) interfaces with the functional simulator (CUDA-sim) through the ptx_thread_info class. The m_thread member variable is an array of ptx_thread_info in the SIMT core class shader_core_ctx and maintains a functional state of all threads active in that SIMT core. The timing model communicates with the functional model through the warp_inst_t class which represents a dynamic instance of an instruction being executed by a single warp.

The timing model communicates with the functional model at the following three stages of simulation.

### Decoding

In the decoding stage at shader_core_ctx::decode(), the timing simulator obtains the instruction from the functional simulator given a PC. This is done by calling the ptx_fetch_inst function.

### Instruction execution

1. Functional execution: The timing model advances the functional state of a thread by one instruction by calling the ptx_exec_inst method of class ptx_thread_info. This is done inside core_t::execute_warp_inst_t. The timing simulator passes the dynamic instance of the instruction to execute, and the functional model advances the thread's state accordingly.
2. SIMT stack update: After functional execution of an instruction for a warp, the timing model updates the next PC in the SIMT stack by requesting it from the functional model. This happens inside simt_stack::update.
3. Atomic callback: If the instruction is an atomic operation, then functional execution of the instruction does not take place in core_t::execute_warp_inst_t. Instead, in the functional execution stage the functional simulator stores a pointer to the atomic instruction in the warp_inst_t object by calling warp_inst_t::add_callback. The timing simulator executes this callback function as the request is leaving the L2 cache (see Memory Partition Connections and Traffic Flow).

### Launching Thread Blocks

When new thread blocks are launched in shader_core_ctx::issue_block2core, the timing simulator initializes the per-thread functional state by calling the functional model method ptx_sim_init_thread. Additionally, the timing model also initializes the SIMT stack and warp states by fetching the starting PC from the functional model.

## Address Decoding

Address decoding is responsible for translating linear addresses to raw addresses, which are used to access the appropriate row, column, and bank in DRAM. Address decoding is also responsible for determining which memory controller to send the memory request to. The code for address decoding is found in **addrdec.h** and **addrdec.cc**; located in "gpgpu-sim_root/src/gpgpu-sim/". When a load or store instruction is encountered in the kernel code, a "memory fetch" object is created (defined in mem_fetch.h/mem_fetch.cc). Upon creation, the mem_fetch object decodes the linear address by calling ddrdec_tlx(new_addr_type addr /*linear address*/, addrdec_t *tlx /*raw address struct*/).

The interpretation of the linear address can be set to one of 13 predefined configurations by setting "-gpgpu_mem_address_mask" in a "gpgpusim.config" file to one of (0, 1, 2, 3, 5, 6, 14, 15, 16, 100, 103, 106, 160). These configurations specify the bit masks used to extract the chip (memory controller), row, col, bank, and burst from the linear address. A custom mapping can be chosen by setting "-gpgpu_mem_addr_mapping" in a "gpgpusim.config" file to a desired mapping, such as

```
    -gpgpu_mem_addr_mapping dramid@8;00000000.00000000.00000000.00000000.0000RRRR.RRRRRRRR.RRBBBCCC.CCCSSSSS
```

Where R(r)=row, B(b)=bank, C(c)=column, S(s)=Burst, and D(d) [not shown]=chip.

Also, dramid@<#> means that the address decoder will insert the dram/chip ID starting at bit <#> (counting from LSB) -- i.e., dramid@8 will start at bit 8.

## Output to AerialVision Performance Visualizer

In gpgpu_sim::cycle(), gpgpu_sim::visualizer_printstat() (in gpgpu-sim/visualizer.cc) is called every sampling interval to append a snapshot of the monitored performance metrics to a log file. This log file is the input for the time-lapse view in AerialVision. The log file is compressed via zlib as it is created to minimize disk usage. The sampling interval can be configured by the option -gpgpu_runtime_stat.

gpgpu_sim::visualizer_printstat() calls a set of functions to sample the performance metrics in various modules:

- cflog_visualizer_gzprint(): Generating data for PC-Histogram (See ISPASS 2010 paper for detail).
- shader_CTA_count_visualizer_gzprint(): The number of CTAs active in each SIMT core.
- shader_core_stats::visualizer_print(): Performance metrics for SIMT cores, including the warp cycle breakdown.
- memory_stats_t::visualizer_print(): Performance metric for memory accesses.
- memory_partition_unit::visualizer_print(): Performance metric for each memory partition. Calls dram_t::visualizer_print().
- time_vector_print_interval2gzfile(): Latency distribution for memory accesses.

The PC-Histogram is implemented using two classes: thread_insn_span and thread_CFlocality. Both classes can be found in gpgpu-sim/stat-tool.{h,cc}. It is interfaced to the SIMT cores via a C interface:

- create_thread_CFlogger(): Create one thread_CFlocality object for each SIMT core.
- cflog_update_thread_pc(): Update PC of a thread. The new PC will be added to the list of PC that was touched by this thread.
- cflog_visualizer_gzprint(): Output the PC-Histogram of this current sampling interval to the log file.

## Histogram

GPGPU-Sim provides several types of histogram data types that simplifies generation of value breakdown for any metric. These histogram classes are implemented in histogram.{h,cc}:

- **binned_histogram**: The base histogram with each unique integer value occupying a bin.
- **pow2_histogram**: A Power-Of-Two histogram with each bin representing log2 of the input value. This is useful when the value of a metric can span a large range (differs by orders of magnitude).
- **linear_histogram**: A histogram with each bin representing a range of values specified by the stride.

All of the histogram classes offer the same interface:

- constructor([stride], name, nbins, [bins]): Create a histogram with a given name, with nbins # of bins, and with bins located by the given pointer (optional). The stride option is only available to **linear_histogram**.
- reset_bins(): Reset all the bins to zero.
- add2bin(sample): Add a sample to the histogram.
- fprint(fout): Print the histogram to the given file handle. Here is the output format:

```
<name> = <number of samples in each bin> max=<maximum among the samples> avg=<average value of the samples>
```

## Dump Pipeline

See #Visualizing_Cycle_by_Cycle_Microarchitecture_Behavior for how this is used.

The top level dump pipeline code is implemented in gpgpu_sim::pipeline(...) in gpgpu-sim/gpu-sim.cc. It calls shader_core_ctx::display_pipeline(...) in gpgpu-sim/shader.cc for the pipeline states in each SIMT core. For memory partition states, it calls memory_partition_unit::print(...) in gpgpu-sim/l2cache.cc.

# CUDA-sim - Functional Simulation Engine

The src/cuda-sim directory contains files that implement the functional simulation engine used by GPGPU-Sim. For increased flexibility the functional simulation engine interprets instructions at the level of individual scalar operations per vector lane.

## Key Objects Descriptions

**kernel_info** (<gpgpu-sim_root>/src/abstract_hardware_model.h/cc):

- The *kernel_info_t* object contains the GPU grid and block dimensions, the *function_info* object associated with the kernel entry point, and memory allocated for the kernel arguments in *param* memory.

**ptx_cta_info** (<gpgpu-sim_root>/src/ptx_sim.h/cc):

- Contains a the thread state (ptx_thread_info) for the set of threads within a cooperative thread array (CTA) (or workgroup in OpenCL).

**ptx_thread_info** (<gpgpu-sim_root>/src/ptx_sim.h/cc):

- Contains functional simulation state for a single scalar thread (work item in OpenCL). This includes the following:
    - Register value storage
    - Local memory storage (private memory in OpenCL)
    - Shared memory storage (local memory in OpenCL). Notice that all scalar threads from the same thread block/workgroup accesses the same shared memory storage.
    - Program counter (PC)
    - Call stack
    - Thread IDs (the software ID within a grid launch, and the hardware ID indicating which hardware thread slot it occupies in timing model)

The current functional simulation engine was developed to support NVIDIA's PTX. PTX is essentially a low level compiler intermediate representation but not the actual machine representation used by NVIDIA hardware (which is known as SASS). Since PTX does not define a binary representation, GPGPU-Sim does not store a binary view of instructions (e.g., as you would learn about when studying instruction set design in an undergraduate computer architecture course). Instead, the text representation of PTX is parsed into a list of objects somewhat akin to a low level compiler intermediate representation.

Individual PTX instructions are found inside of PTX functions that are either kernel entry points or subroutines that can be called on the GPU. Each PTX function has a function_info object:

**function_info** (<gpgpu-sim_root>/src/cuda-sim/ptx_ir.h/cc):

- Contains a list of static PTX instructions (ptx_instruction's) that can be functionally simulated.
- For kernel entry points, stores each of the kernel arguments in a map; *m_ptx_kernel_param_info*; however, this might not always be the case for OpenCL applications. In OpenCL, the associated constant memory space can be allocated in two ways: It can be explicitly initialized in the .ptx file where it is declared, or it can be allocated using the clCreateBuffer on the host. In this later case, the .ptx file will contain a global declaration of the parameter, but it will have an unknown array size. Thus, the symbol's address will not be set and need to be set in the *function_info::add_param_data(...)* function before executing the PTX. In this case, the address of the kernel argument is stored in a symbol table in the *function_info* object.

The list below describe the class hierarchy used to represent instructions in GPGPU-Sim 3.x. The hierarchy was designed to support future expansion of instruction sets beyond PTX and to isolate functional simulation objects from the timing model.

**inst_t** (<gpgpu-sim_root>/src/abstract_hardware_model.h/cc):

- Contains an abstract view of a static instruction relevant to the microarchitecture. This includes the opcode type, source and destination register identifiers, instruction address, instruction size, reconvergence point instruction address, instruction latency and initiation interval, and for memory operations, the memory space accessed.

**warp_inst_t** (<gpgpu-sim_root>/src/abstract_hardware_model.h/cc) (derived from inst_t):

- Contains the view of a dynamic instruction relevant to the microarchitecture. This includes per lane dynamic information such as mask status and memory address accessed. To support accurate functional execution of global memory atomic operations this class includes a callback interface to functionally simulate atomic memory operations when they reach the DRAM interface in performance simulation.

**ptx_instruction** (<gpgpu-sim_root>/src/cuda-sim/ptx_ir.h/cc) (derived from warp_inst_t):

- Contains the full state of a dynamic instruction including the interfaces required for functional simulation.

To support functional simulation GPGPU-Sim must access data in the various memory spaces defined in the CUDA and OpenCL memory models. This requires both a way to name locations and a place to store the values in those locations.

For naming locations, GPGPU-Sim initially builds up a "symbol table" representation while parsing the input PTX. This is done using the following classes:

**symbol_table** (<gpgpu-sim_root>/src/cuda-sim/ptx_ir.h/cc):

- Contains a mapping from the textual representation of a memory location in PTX (e.g., "%r2", "input_data", etc...) to a symbol object that contains information about the data type and location.

**symbol** (<gpgpu-sim_root>/src/cuda-sim/ptx_ir.h/cc):

- Contains information about the name and type of data and its location (address or register identifier) in the simulated GPU memory space. Also tracks where the name was declared in the PTX source.

**type_info** and **type_info_key** (<gpgpu-sim_root>/src/cuda-sim/ptx_ir.h/cc):

- Contains information about the type of a data object (used during instruction interpretation).

**operand_info** (<gpgpu-sim_root>/src/cuda-sim/ptx_ir.h/cc):

- A wrapper class containing a source operand for an instruction which may be either a register identifier, memory operand (including displacement mode information), or immediate operand.

Storage of dynamic data values used in functional simulation uses different classes for registers and memory spaces. Register values are contained in ptx_thread_info::m_regs which is a mapping from symbol pointer to a C union called **ptx_reg_t**. Registers are accessed using the method ptx_thread_info::get_operand_value() which uses operand_info as input. For memory operands this method returns the effective address of the memory operand. Each memory space in the programming model is contained in an object of type **memory_space**. Memory spaces visible to all threads in the GPU are contained in **gpgpu_t** and accessed via interfaces in ptx_thread_info (e.g., ptx_thread_info::get_global_memory).

**memory_space** (<gpgpu-sim_root>/src/cuda-sim/memory.h/cc):

- Abstract base class for implementing memory storage for functional simulation state.

**memory_space_impl** (<gpgpu-sim_root>/src/cuda-sim/memory.h/cc):

- To optimize functional simulation performance, memory is implemented using a hash table. The hash table block size is a template argument for the template class memory_space_impl.

## PTX extraction

Depending on the configuration file, PTX is extracted either from cubin files or using cuobjdump. This section describes the flow of information for extracting PTX and other information. Figure 14 shows the possible flows for the extraction.

### From cubin

__cudaRegisterFatBinary( void *fatCubin ) in the cuda_runtime_api.cc is the function which is responsible for extracting PTX. This function is called by program for each CUDA file. FatbinCubin is a structure which contains different versions of PTX and cubin corresponded to that CUDA file. GPGPU-Sim extract the newest version of PTX which is not newer than forced_max_capability (defines in simulation parameters).

### Using cuobjdump

In CUDA version 4.0 and later, the fat cubin file used to extract the ptx and sass is not available any more. Instead, cuobjdump is used. cuobjdump is a tool provided by NVidia along with the toolkit that can extract the PTX, SASS as well as other information from the executable. If the option -gpgpu_ptx_use_cuobjdump is set to "1" then GPGPU-Sim will invoke cuobjdump to extract the PTX, SASS and other information from the binary. If conversion to PTXPlus is enabled, the simulator will invoke cuobjdump_to_ptxplus to convert the SASS to PTXPlus. The resulting program is then loaded.

## PTX/PTXPlus loading

When the PTX/PTXPlus program is ready, gpgpu_ptx_sim_load_ptx_from_string(...) is called. This function basically use Lex/Yacc to parse the PTX code and create symbol table for that PTX file. Then add_binary(...) is called. This function add created symbol table to CUctx structure which saves all function and symbol table information. Function gpgpu_ptxinfo_load_from_string(...) is invoked in order to extract some information from PTXinfo file. This function run ptxas (the PTX assembler tool from CUDA Toolkit) on PTX file and parse the output file using Lex and Yacc. It extract some information like number of registers using by each kernel from ptxinfo file. Also gpgpu_ptx_sim_convert_ptx_to_ptxplus(...) invoked to to create PTXPlus.

__cudaRegisterFunction(...) function invoked by application for each device function. This function is generate a mapping between device and host functions. Inside register_function(...) GPGPU-sim searches for symbol table associated with that fatCubin in which device function is located. This function generate a map between kernel entry point and CUDA application function address (host function).

## PTXPlus support

This subsection describes how PTXPlus is implemented in GPGPU-Sim 3.x.

### PTXPlus Conversion

GPGPU-Sim version 3.1.0 and later implement support for native hardware ISA execution (PTXPlus) by using NVIDIA's 'cuobjdump' utility. Currently, PTXPlus is only supported with CUDA 4.0. When PTXPlus is enabled, the simulator uses cuobjdump to extract into text format the embedded SASS (NVIDIA's hardware ISA) image included in CUDA binaries. This text representation of SASS is then converted to our own extension of PTX, called PTXPlus, using a separate executable called cuobjdump_to_ptxplus. In the conversion process, more information is needed than available in the SASS text representation. This information is acquired from the ELF and PTX code also extracted using cuobjdump. cuobjdump_to_ptxplus bundles all this information into a single PTXPlus file. Figure 14 depicts the slight differences in run time execution flow when using PTXPlus. Note that there are no changes required in the compilation process of CUDA executables. The conversion process is completely handled by GPGPU-Sim at run-time. Note this flow illustrated in Figure 14 is different than the one illustrated in Figure 3(b) of our ISPASS 2009 paper.

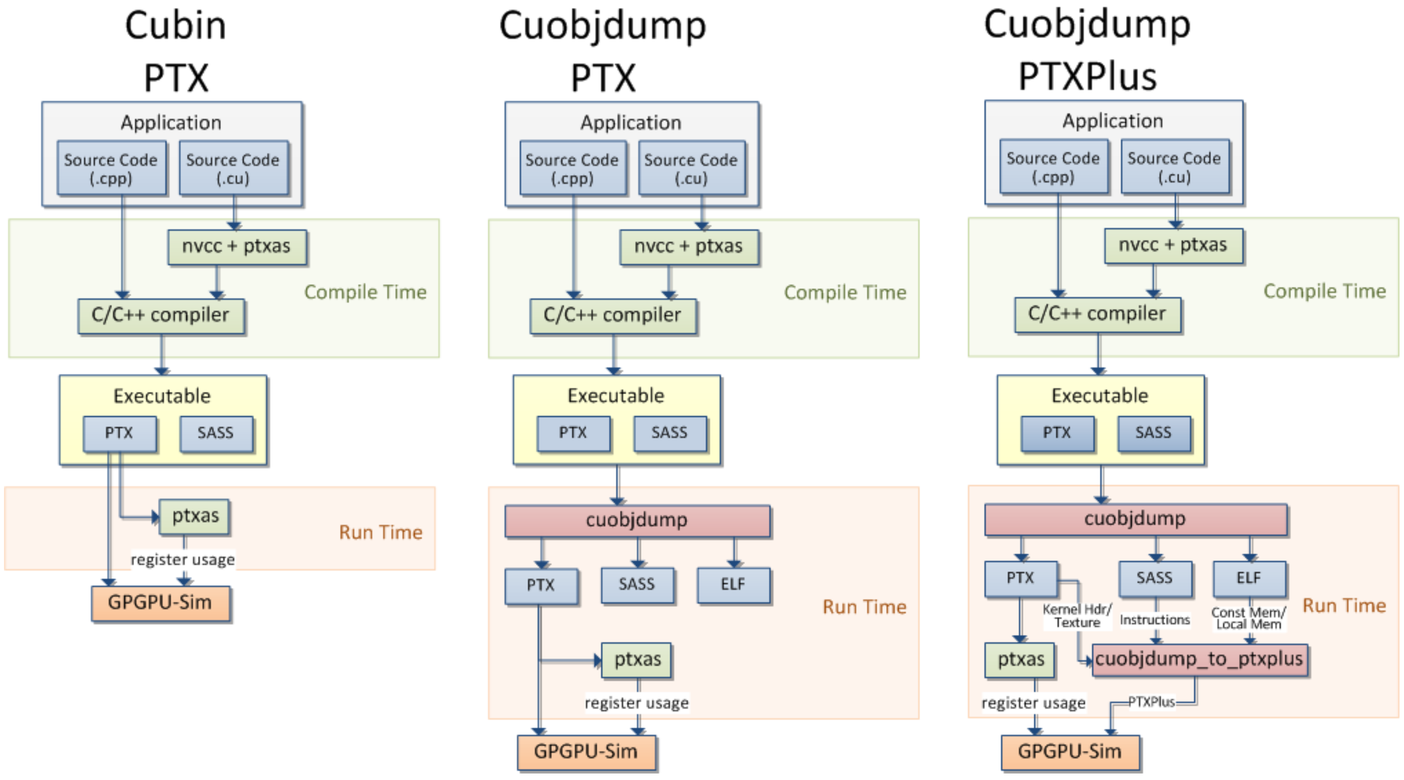


Figure 14: PTX vs PTXPlus Compile Flow

The translation from PTX to PTXPlus is performed by gpgpu_ptx_sim_convert_ptx_and_sass_to_ptxplus() located in ptx_loader.cc. gpgpu_ptx_sim_convert_ptx_and_sass_to_ptxplus() is called by called by usecuobjdump() which passes in the SASS, PTX and ELF information. gpgpu_ptx_sim_convert_ptx_and_sass_to_ptxplus() calls cuobjdump_to_ptxplus on those inputs. cuobjdump_to_ptxplus uses the three inputs to create the final PTXPlus version of the original program and this is returned from gpgpu_ptx_sim_convert_ptx_and_sass_to_ptxplus().

### Operation of cuobjdump_to_ptxplus

cuobjdump_to_ptxplus uses three files to generate PTXPlus. First, before cuobjdump_to_ptxplus is executed, GPGPU-Sim parses the information output by NVIDIA's cuobjdump and merely divides that information into multiple files. For each section (a section corresponds to one CUDA binary), three files are generated: .ptx, .sass and .elf. Those files are merely a split of the output of cuobjdump so it can be easily handled by cuobjdump_to_ptxplus. A description of each is provided below:

- .ptx: contains the PTX code corresponding to the CUDA binary
- .sass: contains the SASS generated by building the PTX code
- .elf: contains a textual dump of the ELF object

cuobjdump_to_ptxplus takes the three files corresponding to a single binary as input and generates a PTXPlus file. Multiple calls are made to cuobjdump_to_ptxplus to convert multiple binaries as needed. Each of the files are parsed, and an elaborate intermediate representation is generated. Multiple functions are then called to output this representation in the form of a PTXPlus file. Below is a description of the information extracted from each of the files:

- .ptx: The ptx file is used to extract information about the available kernels, their function signatures, and information about textures.
- .sass: The sass file is used to extract the actual instructions that will be converted to PTXPlus instructions in the output PTXPlus file.
- .elf: The elf file is used to extract constant and local memory values as well as constant memory pointers.

### PTXPlus Implementation

ptx_thread_info::get_operand_value() in instructions.cc determines the current value of an input operand. The following extensions to get_operand_value are meant for PTXPlus execution.

- If a register operands has a ".lo" modifier, only the lower 16 bits are read. If a register operands has a ".hi" modifier, only the higher 16 bits are read. This information is stored in the m_operand_lohi property of the operand. A value of 0 is default while a value of 1 means a ".lo" modifier and a value of 2 means a ".hi" modifier.
- For PTXPlus style 64bit or 128bit operands, the get_reg() function is passed each register name and the final result is constructed by combining the data in each register.
- The return value from get_double_operand_type() indicates the use of one of the new ways of determining the memory address.
    - If it's 1, the value of two registers must be added together.
    - If it's 2, the address is stored in a register and the value in the register is postincremented by a value in a second register.
    - If it's 3, the address is stored in a register and the value in the register is postincremented by a constant.
- For memory operands, the first half of the get_operand_value() function calculates the memory address to access. This value is stored in result. result is used as the address and the appropriate data is fetched from the address space indicated by the operand and returned. For postincrementing memory accesses, the register holding the address is also incremented in get_operand_value().
- If it isn't a memory operand, the value of the register is returned.
- get_operand_value checks for a negative sign on the operand and takes the negative of finalResult if necessary before returning it.

## Control Flow Analysis + Pre-decode

Each kernel function is analyzed and *pre-decoded* as it is loaded into GPGPU-Sim. When the PTX parser detects the end of a kernel function, it calls function_info::ptx_assemble() (in cuda-sim/cuda-sim.cc). This function does the following:

- Assign each instruction in the function with a unique PC
- Resolve each branch label in the function to a corresponding instruction/PC
    - I.e. Determine the branch target for each branch instruction
- Create control flow graph for the function
- Perform control-flow analysis
- Pre-decode each instruction (to speed up simulation)

Creation of the control flow graph is done via two member functions in function_info:

- create_basic_blocks() groups individual instructions into basic block (basic_block_t).
- connect_basic_blocks() connects the basic blocks to form a control flow graph.

After creating the control flow graph, two control flow analysis will be done.

- Determine the target of each break instruction:
    - This is a make-shift solution to support break instruction in PTXPlus, which implements break-statements in while-loops. A long-term solution is to extend the SIMT stack with proper break entries.
    - The goal is to determine the latest breakaddr instruction that precedes each break instruction. Assuming that the code has structured control flow. This information can be determined by traversing upstream through the dominator tree (constructed by calling member functions find_dominators() and find_idominators()). However, the control flow graph changes after break instructions are connected to their targets. The current solution is to perform this analysis iteratively until both the dominator tree and the break targets become stable.
    - The algorithm for finding dominators are described in Muchnick's Adv. Compiler Design and Implementation (Figure 7.14 and Figure 7.15).
- Find immediate post-dominator of each branch instruction:
    - This information is used by the SIMT stack for reconvergence point at a divergent branch.
    - The analysis is done by calling member functions find_postdominators() and find_ipostdominators(). The algorithm is described in Muchnick's Adv. Compiler Design and Implementation (Figure 7.14 and Figure 7.15).

Pre-decode is performed by calling ptx_instruction::pre_decode() for each instruction. It extracts information that is useful to the timing simulator.

- Detect LD/ST instruction.
- Determine if the instruction writes to a destination register.
- Obtain the reconvergence PC if this is a branch instruction.
- Extract the register operands of the instruction.
- Detect predicated instruction.

The extracted information is stored inside the ptx_instruction object corresponding to the instruction. This speeds up simulation because all scalar threads in a kernel launch executes the same kernel function. Extracting these information once as the function is loaded is significantly more efficient than repeating the same extraction for each thread during simulation.

## Memory Space Buffer

In CUDA-sim, the various memory spaces in CUDA/OpenCL are implemented functionally with memory space buffers (the memory_space_impl class in memory.h).

- All of global, texture, constant memory spaces are implemented with a single memory_space object inside the top-level gpgpu_t class (as member object m_global_memory).
- The local memory space of each thread is contained in the ptx_thread_info object corresponding to each thread.
- The shared memory space is common to the entire CTA (thread block), and a unique memory_space object is allocated for each CTA when it is dispatched for execution (in function ptx_sim_init_thread()). The object is deallocated when the CTA has completed execution.

The memory_space_impl class implements the read-write interface defined by abstract class memory_space. Internally, each memory_space_impl object contains a set of memory pages (implemented by class template mem_storage). It uses a STL unordered map (reverts to STL map if unordered map is not available) to associate pages with their corresponding addresses. Each mem_storage object is an array of bytes with read and write functions. Initially, each memory_space object is empty, and pages are allocated on demand as an address corresponding to the individual page in the memory space is accessed (either via an LD/ST instruction or cudaMemcpy()).

The implementation of memory_space, memory_space_impl and mem_storage can be found in files memory.h and memory.cc.

## Global/Constant Memory Initialization

In CUDA, programmer can declare device variables that are accessible to all kernel/device functions. These variables can be in global (e.g. x_d) or constant memory space (e.g. y_c):

```
__device__ int x_d = 100;
__constant__ int y_c = 70;
```

These variables, and their initial values, are compiled into PTX variables.

In GPGPU-Sim, these variables are parsed via the PTX parser into (symbol, value) pairs. After the all the PTX are loaded, two functions, load_stat_globals(...) and load_constants(...) are called to assign each variable with a memory address in the simulated global memory space, and copy the initial value to the assigned memory location. The two functions are located inside cuda_runtime_api.cc.

These variables can also be declared as __global__ in CUDA. In this case, they are accessible by both the host (CPU) and the device (GPU). CUDA accomplish this by having two copies of the same variable in both host memory and device memory. The linkage between the two copies is established using function __cudaRegisterVar(...). GPGPU-Sim intercept call to this function to acquire this information, and establish a similar linkage by calling functions gpgpu_ptx_sim_register_const_variable(...) or gpgpu_ptx_sim_register_global_variable(...) (implemented in cuda-sim/cuda-sim.cc). With this linkage established, the host may call cudaMemcpyToSymbol() or cudaMemcpyFromSymbol() to access these __global__ variables. GPGPU-Sim implements these functions with gpgpu_ptx_memcpy_symbol(...) in cuda-sim/cuda-sim.cc.

Notice that __cudaRegisterVar(...) is not part of CUDA Runtime API, and future versions of CUDA may implement __global__ variables in a different way. In that case, GPGPU-Sim will need to be modified to support the new implementation.

## Kernel Launch: Parameter Hookup

Kernel parameters in GPGPU-Sim are set using the same methods as in regular CUDA and OpenCL applications;

```
kernel_name <<x,y>> (param1, param2, ..., paramN) and
clSetKernelArg(kernel_name, arg_index, arg_size, arg_value)
```

respectively.

Another method to pass the kernel arguments in CUDA is with the use of *cudaSetupArgument(void* arg, size_t count, size_t offset)* . This function pushes *count* bytes of the argument pointed to by *arg* at *offset bytes* from the start of the parameter passing area, which starts at offset 0. The arguments are stored in the top of the execution stack. For example, if a CUDA kernel is to have 3 arguments, *a*, *b*, and *c* (in this order), the offset for *a* is 0, offset for *b* is sizeof(a), and offset for *c* is sizeof(a)+sizeof(b).

For both CUDA and OpenCL, GPGPU-Sim creates a *gpgpu_ptx_sim_arg* object per kernel argument and maintains a list of all kernel arguments. Prior to executing the kernel, an initialization function is called to setup the GPU grid dimensions and parameters: *gpgpu_cuda_ptx_sim_init_grid(...)* or *gpgpu_opencl_ptx_sim_init_grid(...)*. Two main objects are used within these functions, the *function_info* and *kernel_info_t* objects, which are described above. In the init_grid functions, the kernel arguments are added to the *function_info* object by calling *function_info_object::add_param_data(arg #, gpgpu_ptx_sim_arg *)*.

After adding all of the parameters to the *function_info* object, *function_info::finalize(...)* is called, which copies over the kernel arguments, stored in the *function_info::m_ptx_kernel_param_info* map, into the parameter memory allocated in the *kernel_info_t* object mentioned above. If it was not done previously in *function_info::add_param_data(...)*, the address of each kernel argument is added to the symbol table in the *function_info* object.

PTXPlus support requires copying kernel parameters to shared memory. The kernel parameters can be copied from *Param* memory to *Shared* memory by calling the *function_info::param_to_shared(shared_mem_ptr, symbol_table)* function. This function iterates over the kernel parameters stored in the *function_info::m_ptx_kernel_param_info* map and copies each parameter from *Param* memory to the appropriate location in *Shared* memory pointed to by *ptx_thread_info::shared_mem_ptr*.

The *function_info::add_param_data(...), function_info::finalize(...), function_info::param_to_shared(...),* and *gpgpu_opencl_ptx_sim_init_grid(...)* functions are defined in <gpgpu-sim_root>/distribution/src/cuda-sim/cuda-sim.cc. The *gpgpu_cuda_ptx_sim_init_grid(...)* function is implemented in <gpgpu-sim_root>/distribution/libcuda/cuda_runtime_api.cc.

## Generic Memory Space

Generic addressing is a feature that was introduced in NVIDIA's PTX 2.0, with generic addressing supported by instructions ld, ldu, st, prefetch, prefetchu, isspacep, cvta, atom, and red. In generic addressing, an address maps to global memory unless it falls within the local memory window or the shared memory window. Within these windows, an address maps to the corresponding location in local or shared memory, i.e. to the address formed by subtracting the window base from the generic address to form the offset in the implied state space. So an instruction can use generic addressing to deal with addresses that may correspond to global, local or shared memory space.

The generic addressing in GPGPU-Sim is supported with instructions ld, ldu, st, isspacep and cvta. Functions generic_to_{shared, local, global}, {shared, local, global}_to_generic and isspace_{shared, local, global} (which are all defined in "cuda_sim.cc") are used to support the generic addressing in GPGPU-Sim with the previously mentioned instructions.

Identifiers (SHARED_GENERIC_START, SHARED_MEM_SIZE_MAX, LOCAL_GENERIC_START, TOTAL_LOCAL_MEM_PER_SM, LOCAL_MEM_SIZE_MAX, GLOBAL_HEAP_START and STATIC_ALLOC_LIMIT) which are defined in "abstract_hardware_model.h" are used to define the different memory spaces boundaries (windows). These identifiers are used to derive and generate different address spaces which are needed to support generic addressing.

The follwing table shows an example of how the spaces are defined in the code:

| Identifier | Value |
|---|---|
| GLOBAL_HEAP_START | 0x80000000 |
| SHARED_MEM_SIZE_MAX | 64*1024 |
| LOCAL_MEM_SIZE_MAX | 8*1024 |
| MAX_STREAMING_MULTIPROCESSORS | 64 |
| MAX_THREAD_PER_SM | 2048 |
| TOTAL_LOCAL_MEM_PER_SM | MAX_THREAD_PER_SM*LOCAL_MEM_SIZE_MAX |
| TOTAL_SHARED_MEM | MAX_STREAMING_MULTIPROCESSORS*SHARED_MEM_SIZE_MAX |
| TOTAL_LOCAL_MEM | MAX_STREAMING_MULTIPROCESSORS*MAX_THREAD_PER_SM*LOCAL_MEM_SIZE_MAX |
| SHARED_GENERIC_START | GLOBAL_HEAP_START-TOTAL_SHARED_MEM |
| LOCAL_GENERIC_START | SHARED_GENERIC_START-TOTAL_LOCAL_MEM |
| STATIC_ALLOC_LIMIT | GLOBAL_HEAP_START - (TOTAL_LOCAL_MEM+TOTAL_SHARED_MEM) |

Notice that with this address space partitioning, each thread may only have up to 8kB of local memory (LOCAL_MEM_SIZE_MAX). With CUDA compute capability 1.3 and below, each thread can have up to 16kB of local memory. With CUDA compute capability 2.0, this limit has increased to 512kB [7]. The user may increase LOCAL_MEM_SIZE_MAX to support applications that require more than 8kB of local memory per thread. However, one should always ensure that GLOBAL_HEAP_START > (TOTAL_LOCAL_MEM + TOTAL_SHARED_MEM). Failure to do so may result in erroneous simulation behavior.

To get more information about the generic addressing in general refer to NVIDIA's PTX: Parallel Thread Execution ISA Version 2.0 manual[8].

## Instruction Execution

After parsing, instructions used for functional execution are represented as a ptx_instruction object contained within a function_info object (see cuda-sim/ptx_ir.{h,cc}). Each scalar thread is represented by a ptx_thread_info object. Executing an instruction (functionally) is mainly accomplished by calling the ptx_thread_info::ptx_exec_inst().

The abstract class core_t has the most basic data structures and procedures required for instruction execution functionally. This class is the base class for the shader_core_ctx and functionalSimCore which are used for performance and pure functional simulation respectively. core_t most important members are objects of types simt_stack and ptx_thread_info, which are used in the functional exectuion to keep track of the warps branch divergence and to handle the threads' instructions execution.

Executing instruction simply starts by initializing scalar threads using the function ptx_sim_init_thread (in cuda-sim.cc), then we execute the scalar threads in warps using the function_info::ptx_exec_inst(). In this version, keeping track of threads as warps is done using a simt_stack object for each warp of scalar threads (this is the assumed model here and other models can be used instead), the simt_stack tells which threads are active and which instruction to execute at each cycle so we can execute the scalar threads in warps.

In ptx_thread_info::ptx_exec_inst, is where acutally the instructions get functionally executed. We check the instruction opcode and call the corresponding funciton, the file opcodes.def contains the functions used to execute each instruction. Every instruction function takes two parameters of type ptx_instruction and ptx_thread_info which hold the data for the instruction and the thread in execution receptively.

Information are communicated back from the execution of ptx_exec_inst to the function that executes the warps through modifying warp_inst_t parameter that is passed to the ptx_exec_inst by reference, so for atomics we indicate that the executed warp instruction is atomic and add a call back to the warp_inst_t and which set the atomic flag, the flag is then checked by the warp execution function in order to do the callbacks which are used to execute the atomics (check functionalCoreSim::executeWarp in cuda-sim.cc).

As you might have expected more communication is made in the performance simulation than the pure functional simulation. The pure functional execution with functionalSimCore (in cuda-sim{.h,.cc}) can be checked to get more details on the functional execution.

## Interface to Source Code View in AerialVision

Source Code View in AerialVision is a view where it is possible to plot different kinds of metrics vs. ptx source code. For example, one could plot the DRAM traffic generated by each line in ptx source code. GPGPU-Sim exports the statistics needed to construct the Source Code View in AerialVision to statistics files that are read by AerialVision.

If the options "-enable_ptx_file_line_stats 1" and "-visualizer_enabled 1" are defined, GPGPU-Sim will save files with the statistics. The name of the file can be specified using the option "-ptx_line_stats_filename filename".

For each line in the executed ptx files, one line is added to the line stats file in the following format:

```
kernel line : count latency dram_traffic smem_bk_conflicts smem_warp gmem_access_generated gmem_warp exposed_latency warp_divergence
```

Using AerialVision, one could plot/view the statistics collected by this interface in different ways. AerialVision can also map those statistics to CUDA C++ source files. Please refer the AerialVision manual for more details about how to do that.

This functionality is implemented in src/cuda-sim/ptx-stats.h(.cc). The stats for each ptx line are held in an instance of the class *ptx_file_line_stats*. The function *void ptx_file_line_stats_write_file()* is responsible for printing the statistics to the statistics file in the above format. A number of other functions, similar to *void ptx_file_line_stats_add_dram_traffic(unsigned pc, unsigned dram_traffic) are called by the rest of the simulator to record different statistics about ptx source code lines.*

### Pure Functional Simulation

Pure functional simulation (bypassing performance simulation) is implemented in files cuda-sim{.h,.cc}, in function gpgpu_cuda_ptx_sim_main_func(...) and using the functionalCoreSim class. The functionalCoreSim class is inherited from the core_t abstract class, which contains many of the functional simulation data structures and procedures that are used by the pure functional simulation as well as performance simulation.

## Interface with outside world

GPGPU-Sim is compiled into stub libraries that dynamically link at runtime to the CUDA/OpenCL application. Those libraries intercept the call intended to be executed by the CUDA runtime environment and instead initialize the simulator and run the kernels on it instead of on the hardware

### Entry Point and Stream Manager

GPGPU-Sim is initialized by the function GPGPUSim_Init() which is called when the CUDA or OpenCL application performs its first CUDA/OpenCL API call. Our implementation of the CUDA/OpenCL API function implementations either directly call GPGPUSim_Init() or they call GPGPUSim_Context() which in turn calls GPGPUSim_Init(). An example API call that calls GPGPUSim_Context() is cudaMalloc(). Note that by utilizing static variables GPGPUSim_Init() is not called every time a cudaMalloc() is called.

First call to GPGPUSim_Init() would call the function gpgpu_ptx_sim_init_perf() located in gpgpusim_entrypoint.cc. Inside gpgpu_ptx_sim_init_perf() all the environmental variables, command line parameters and configuration files are processed. Based on the options a gpgpu_sim object is instantiated and assigned to the global variable g_the_gpu. Also a stream_manager object is instantiated and assigned to the global variable g_stream_manager.

GPGPUSim_Init() also calls start_sim_thread() function located in gpgpusim_entrypoint.cc. start_sim_thread() creates starts a new pthread which is actually responsible for running the simulation. For OpenCL application, the simulator pthread runs gpgpu_sim_thread_sequential(), which simulates the execution of kernels one at a time. For CUDA application, the simulator pthread runs gpgpu_sim_thread_concurrent(), which simulates the concurrent execution of multiple kernels. The maximum number of kernels that may concurrently execute on the simulated GPU is configured by the option '-gpgpu_max_concurrent_kernel'.

gpgpu_sim_thread_sequential() will wait for a start signal to start the simulation (g_sim_signal_start). The start signal is set by the gpgpu_opencl_ptx_sim_main_perf() function used to start OpenCL performance simulation.

gpgpu_sim_thread_concurrent() initializes the performance simulator structures once and then enters a loop waiting for a job from the stream manager (implemented by class stream_manager in stream_manager.h/cc). The stream manager itself gets the jobs from CUDA API calls to functions such as cudaStreamCreate() and cudaMemcpy() and kernel launches.

### CUDA runtime library (libcudart)

When building a CUDA application, NVIDIA's nvcc translates each kernel launch into a series of API calls to the CUDA runtime API, which prepares the GPU hardware for kernel execution. libcudart.so is the library provided by NVIDIA that implements this API. In order for GPGPU-Sim to intercept those calls and run the kernels on the simulator, GPGPU-Sim also implements this library. The implementation resides in libcuda/cuda_runtime_api.cc. The resulting shared object resides in gpgpu-sim_root/lib/<build_type>/libcudart.so. By including this path into your LD_LIBRARY_PATH, you instruct your system to dynamically link against GPGPU-Sim at runtime instead of the NVIDIA provided library, thus allowing the simulator to run your code. Setting your LD_LIBRARY_PATH should be done through the setup_environment script as instructed in the README file distributed with GPGPU-Sim. Our implementation of libcudart is not compatible with all versions of cuda because of different interfaces that NVIDIA uses between different versions.