GPGPU-SIM Code Study (version: 3.1.2)

-by Yong Li

This note is not a tutorial of how to use the GPGPU-Sim simulator. For the official manual and introduction of GPGPU-Sim please refer to http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual. This note aims to put together major components from the simulator and help understanding the entire simulation process from the source code level. Emphasis is put on the data structure and software flow of the simulator, which I think is critical for modifying and re-architecting the simulator for various research purposes.

Part 1. Using gdb to Get Start

start gdb and run the BFS benchmark.

When the benchmark starts running and triggers a cuda library call for the first time (with GPGPU-SIM the benchmark actually calls a library call provided by the GPGPU-SIM), the simulator will print information such as *** GPGPU-Sim Simulator Version …. ***. Using *grep* to search this string and it can be found in the function *print_splash()* defined in cuda-sim.cc.

Now break at print_splash() function and print stack:

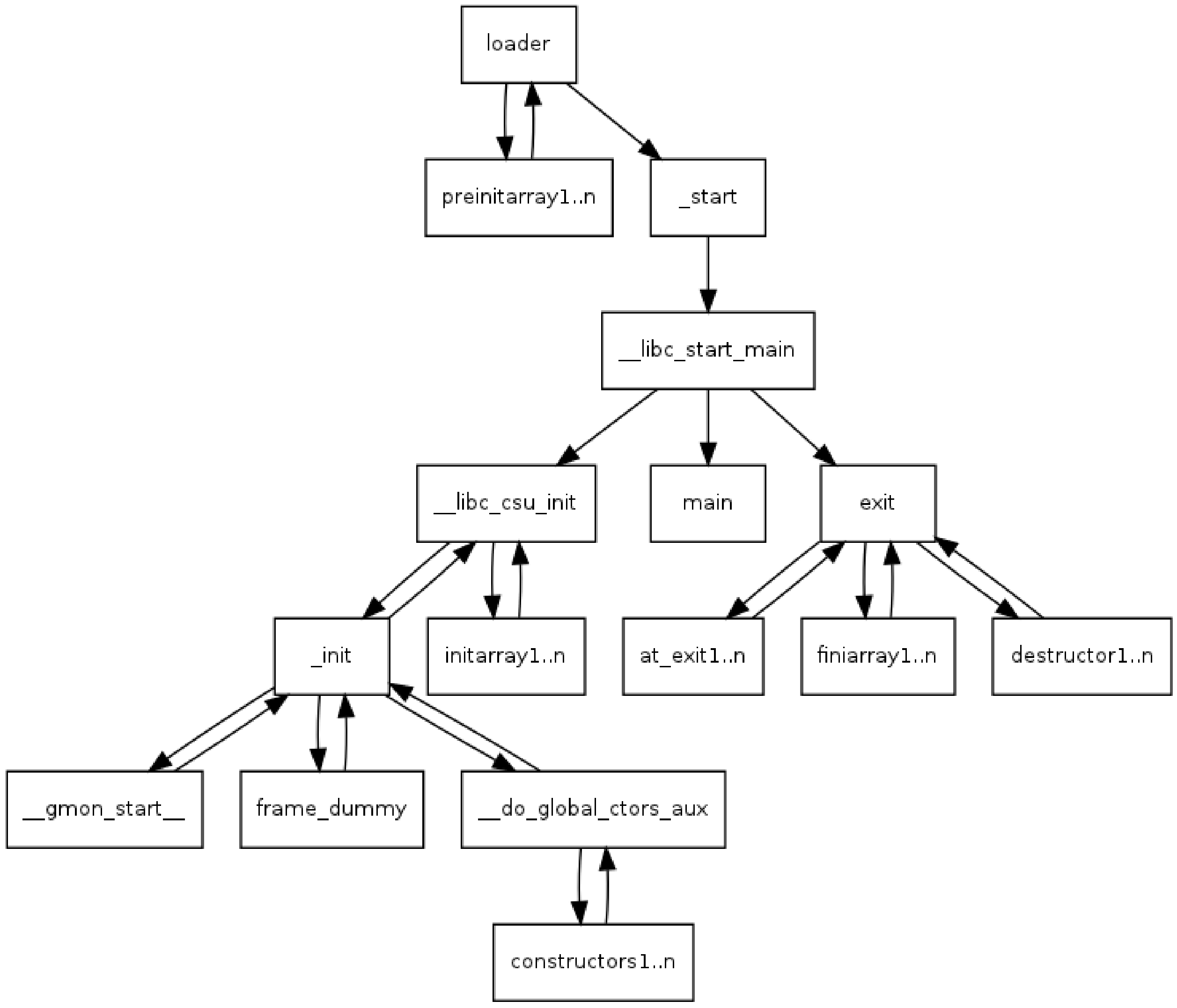
```
Breakpoint 4, Reading in symbols for gpgpusim_entrypoint.cc...done.
print_splash () at cuda-sim.cc:1425
1425          if ( !splash_printed ) {
(gdb) info stack
Reading in symbols for cuda_runtime_api.cc...done.
#0  print_splash () at cuda-sim.cc:1425
#1  0x00007ffff7af91ef in gpgpu_ptx_sim_init_perf () at gpgpusim_entrypoint.cc:173
#2  0x00007ffff79b6728 in GPGPUSim_Init () at cuda_runtime_api.cc:302
#3  0x00007ffff79b690f in GPGPUSim_Context () at cuda_runtime_api.cc:337
#4  0x00007ffff79bb31a in __cudaRegisterFatBinary (fatCubin=0x61d398) at
cuda_runtime_api.cc:1556
#5  0x000000000040400e in
__sti____cudaRegisterAll_49_tmpxft_000026e2_00000000_6_bfs_compute_10_cpp1_ii_29cadddc()
()
#6  0x0000000000414dad in __libc_csu_init ()
#7  0x00007ffff6bf8700 in __libc_start_main () from /lib/x86_64-linux-gnu/libc.so.6
#8  0x0000000000403f39 in _start ()
```

To understand the two calls on bottom of the stack (_start() and __licc_start_main()), please refer to this: http://linuxgazette.net/issue84/hawk.html
Here is what happens.

1. GCC build your program with crtbegin.o/crtend.o/gcrt1.o And the other default libraries are dynamically linked by default. Starting address of the executable is set to that of _start.
2. Kernel loads the executable and setup text/data/bss/stack, especially, kernel allocate page(s) for arguments and environment variables and pushes all necessary information on stack.
3. Control is pased to _start. _start gets all information from stack setup by kernel, sets up argument stack for __libc_start_main, and calls it.
4. __libc_start_main initializes necessary stuffs, especially C library(such as malloc) and thread environment and calls our main.
5. our main is called with main(argv, argv) Actually, here one interesting point is the signature of main. __libc_start_main thinks main's signature as main(int, char **, char **) If you are curious, try the following prgram.

There is a more detailed description here and it talks about *__libc_csu_init()* as well:

http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html
A nice figure from the above link:



Yet to understand *__libc_csu_init()* better and what's its functionality during the cuda registration phase before the main function is called I found this material: http://www.acsu.buffalo.edu/~charngda/elf.html. It says the following:

What user functions will be executed before main and at program exit?

As above call strack trace shows, _init is NOT the only function to be called before main. It is __libc_csu_init function (in Glibc's source file csu/elf-init.c) that determines what functions to be run before main and the order of running them. Its code is like this

```
 void __libc_csu_init (int argc, char **argv, char **envp)
 {
 #ifndef LIBC_NONSHARED
   {
     const size_t size = __preinit_array_end - __preinit_array_start;
     size_t i;
     for (i = 0; i < size; i++)
       (*__preinit_array_start [i]) (argc, argv, envp);
   }
 #endif

   _init ();

   const size_t size = __init_array_end - __init_array_start;
   for (size_t i = 0; i < size; i++)
       (*__init_array_start [i]) (argc, argv, envp);
 }
```

(Symbols such as __preinit_array_start, __preinit_array_end, __init_array_start, __init_array_end are defined by the default ld script; look for PROVIDE and PROVIDE_HIDDEN keywords in the output of ld -verbose command.)

The __libc_csu_fini function has similar code, but what functions to be executed at program exit are actually determined by exit:

```
 void __libc_csu_fini (void)
 {
 #ifndef LIBC_NONSHARED
   size_t i = __fini_array_end - __fini_array_start;
   while (i-- > 0)
     (*__fini_array_start [i]) ();

   _fini ();
 #endif
 }
```

To see what's going on, consider the following C code example:

```
#include <stdio.h>
#include <stdlib.h>

void preinit(int argc, char **argv, char **envp) {
  printf("%s\n", __FUNCTION__);
}

void init(int argc, char **argv, char **envp) {
  printf("%s\n", __FUNCTION__);
}

void fini() {
  printf("%s\n", __FUNCTION__);
}

__attribute__((section(".init_array"))) typeof(init) *__init = init;
__attribute__((section(".preinit_array"))) typeof(preinit) *__preinit = preinit;
__attribute__((section(".fini_array"))) typeof(fini) *__fini = fini;

void  __attribute__ ((constructor)) constructor() {
  printf("%s\n", __FUNCTION__);
}

void __attribute__ ((destructor)) destructor() {
  printf("%s\n", __FUNCTION__);
}

void my_atexit() {
  printf("%s\n", __FUNCTION__);
}

void my_atexit2() {
  printf("%s\n", __FUNCTION__);
}

int main() {
  atexit(my_atexit);
  atexit(my_atexit2);
}
```

The output will be

```
preinit
constructor
init
my_atexit2
my_atexit
fini
destructor
```

The .preinit_array and .init_array sections must contain function pointers (NOT code!) The prototype of these functions must be

void func(int argc,char** argv,char** envp)

__libc_csu_init executes them in the following order:

1. Function pointers in .preinit_array section
2. Functions marked as __attribute__ ((constructor)), via _init
3. Function pointers in .init_array section

The .fini_array section must also contain function pointers and the prototype is like the destructor, i.e. taking no arguments and returning void. If the program exits normally, then the exit function (Glibc source file stdlib/exit.c) is called and it will do the following:

1. In reverse order, functions registered via atexit or on_exit
2. Function pointers in .fini_array section, via __libc_csu_fini
3. Functions marked as __attribute__ ((destructor)), via __libc_csu_fini (which calls _fini after Step 2)
4. stdio cleanup functions

For the **cudaRegisterAll() function I googled it and found the following explanation:
The simplest way to look at how nvcc compiles the ECS (Execution Configuration Syntax) and manages kernel code is to use nvcc's --cuda switch. This generates a .cu.c file that can be compiled and linked without any support from NVIDIA proprietary tools. It can be thought of as CUDA source files in open source C. Inspection of this file verified how the ECS is managed, and showed how kernel code was managed.
1. Device code is embedded as a fat binary object in the executable's .rodata section. It has variable length depending on the kernel code.
2. For each kernel, a host function with the same name as the kernel is added to the source code.
3. Before main(..) is called, a function called cudaRegisterAll(..) performs the following work:
• Calls a registration function, cudaRegisterFatBinary(..), with a void pointer to the fat binary data. This is where we can access the kernel code directly.
• For each kernel in the source file, a device function registration function, cudaRegisterFunction(..), is called. With the list of parameters is a pointer to the function mentioned in step 2.
4. As aforementioned, each ECS is replaced with the following function calls from the execution management category of the CUDA runtime API.
• cudaConfigureCall(..) is called once to set up the launch configuration.
• The function from the second step is called. This calls another function, in which, cudaSetupArgument(..) is called once for each kernel parameter. Then, cudaLaunch(..) launches the kernel with a pointer to the function from the second step.
5. An unregister function, cudaUnregisterBinaryUtil(..), is called with a handle to the fatbin data on program exit.

It seems that nvcc keeps track of kernels and kernel launches by registering a list of function pointers (the second step in the list above). All the functions above are undocumented from NVIDIA's part. The virtual CUDA library solves the problem of registering kernel code and tracking kernel launches by reimplementing each one of them, which we will get to in a minute.

From the *__cudaRegisterFatBinary* above *(GPGPUSim_Context () , GPGPUSim_Init (), etc.)* are functions that GPGPU-SIM defines to mimic the behavior of the CUDA runtime API.

At this point we should now understand why the stack looks like that and the calling sequence at the entry point in GPGPU-SIM.
Here are additional stack information I printed after stepping into the function *cudaRegisterAll:*

```
(gdb)
Single stepping until exit from function
_ZL86__sti____cudaRegisterAll_49_tmpxft_000026e2_00000000_6_bfs_compute_10_cpp1_ii_29cadddcv,
which has no line number information.
GPGPU-Sim PTX: __cudaRegisterFunction _Z6KernelP4NodePiPbS2_S1_S2_i : hostFun
0x0x404170, fat_cubin_handle = 1
GPGPU-Sim PTX: instruction assembly for function '_Z6KernelP4NodePiPbS2_S1_S2_i'...   done.
GPGPU-Sim PTX: finding reconvergence points for '_Z6KernelP4NodePiPbS2_S1_S2_i'...
GPGPU-Sim PTX: Finding dominators for '_Z6KernelP4NodePiPbS2_S1_S2_i'...
GPGPU-Sim PTX: Finding immediate dominators for '_Z6KernelP4NodePiPbS2_S1_S2_i'...
GPGPU-Sim PTX: Finding postdominators for '_Z6KernelP4NodePiPbS2_S1_S2_i'...
GPGPU-Sim PTX: Finding immediate postdominators for '_Z6KernelP4NodePiPbS2_S1_S2_i'...
GPGPU-Sim PTX: pre-decoding instructions for '_Z6KernelP4NodePiPbS2_S1_S2_i'...
GPGPU-Sim PTX: reconvergence points for _Z6KernelP4NodePiPbS2_S1_S2_i...
GPGPU-Sim PTX:  1 (potential) branch divergence @  PC=0x030 (_1.ptx:72)     @%p1 bra
$Lt_0_5122
GPGPU-Sim PTX:       immediate post dominator         @  PC=0x1f8 (_1.ptx:147)     exit
GPGPU-Sim PTX:  2 (potential) branch divergence @  PC=0x068 (_1.ptx:79)     @%p2 bra
$Lt_0_5122
GPGPU-Sim PTX:       immediate post dominator         @  PC=0x1f8 (_1.ptx:147)     exit
GPGPU-Sim PTX:  3 (potential) branch divergence @  PC=0x0e0 (_1.ptx:97)     @%p3 bra
$Lt_0_5122
GPGPU-Sim PTX:       immediate post dominator         @  PC=0x1f8 (_1.ptx:147)     exit
GPGPU-Sim PTX:  4 (potential) branch divergence @  PC=0x140 (_1.ptx:114)     @%p4 bra
$Lt_0_4354
GPGPU-Sim PTX:       immediate post dominator         @  PC=0x1d8 (_1.ptx:140)     add.s32
%r8, %r8, 1
GPGPU-Sim PTX:  5 (potential) branch divergence @  PC=0x1f0 (_1.ptx:143)     @%p5 bra
$Lt_0_4098
GPGPU-Sim PTX:       immediate post dominator         @  PC=0x1f8 (_1.ptx:147)     exit
GPGPU-Sim PTX: ... end of reconvergence points for _Z6KernelP4NodePiPbS2_S1_S2_i
GPGPU-Sim PTX: ... done pre-decoding instructions for '_Z6KernelP4NodePiPbS2_S1_S2_i'.
GPGPU-Sim PTX: finished parsing EMBEDDED .ptx file _1.ptx
Adding _cuobjdump_1.ptx with cubin handle 1
GPGPU-Sim PTX: extracting embedded .ptx to temporary file "_ptx_DIAeer"
Running: cat _ptx_DIAeer | sed 's/.version 1.5/.version 1.4/' | sed 's/, texmode_independent//' | sed
's/\(\.extern \.const\[1\] .b8 \w\+\)\[\]/\1\[1\]/' | sed 's/const\[.\]/const\[0\]/g' > _ptx2_Jxpnzr
Detaching after fork from child process 5632.
```

```
GPGPU-Sim PTX: generating ptxinfo using "$CUDA_INSTALL_PATH/bin/ptxas --gpu-name=sm_20 -
v _ptx2_Jxpnzr --output-file  /dev/null 2> _ptx_DIAeerinfo"
Detaching after fork from child process 5638.
GPGPU-Sim PTX: Kernel '_Z6KernelP4NodePiPbS2_S1_S2_i' : regs=18, lmem=0, smem=0,
cmem=84
GPGPU-Sim PTX: removing ptxinfo using "rm -f _ptx_DIAeer _ptx2_Jxpnzr _ptx_DIAeerinfo"
Detaching after fork from child process 5640.
GPGPU-Sim PTX: loading globals with explicit initializers...
GPGPU-Sim PTX: finished loading globals (0 bytes total).
GPGPU-Sim PTX: loading constants with explicit initializers...  done.
0x0000000000414dad in __libc_csu_init ()
(gdb) info stack
#0  0x0000000000414dad in __libc_csu_init ()
#1  0x00007ffff6bf8700 in __libc_start_main () from /lib/x86_64-linux-gnu/libc.so.6
#2  0x0000000000403f39 in _start ()
```

From the above trace we can see that GPGPU-SIM does some PTX parsing and analysis (e.g., branch divergence) at the very beginning of the simulation (before the main() function in the running benchmark.) By tracing the message "*instruction assembly for function*" I found this is actually generated by the *ptx_assemble()* function (defined in cuda-sim.cc), which is called by *gpgpu_ptx_assemble()* defined in ptx_ir.cc, which is further called by *end_function()* in ptx_parser.cc.

When the benchmark launches a kernel (at the cudaLaunch breakpoint), I print the stack as follows:

```
Breakpoint 5, cudaLaunch (hostFun=0x404170 "\351\v\377\377\377ff.\017\037\204") at
cuda_runtime_api.cc:906
906          CUctx_st* context = GPGPUSim_Context();
(gdb) info stack
#0  cudaLaunch (hostFun=0x404170 "\351\v\377\377\377ff.\017\037\204") at
cuda_runtime_api.cc:906
#1  0x0000000000404167 in __device_stub__Z6KernelP4NodePiPbS2_S1_S2_i(Node*, int*, bool*,
bool*, int*, bool*, int) ()
#2  0x0000000000404705 in BFSGraph(int, char**) ()
#3  0x0000000000404b8d in main ()
```

While the benchmark is running and stop at the step, I use "info stack" to print the stack information as follows:

```
(gdb) info stack
Reading in symbols for stream_manager.cc...done.
Reading in symbols for cuda_runtime_api.cc...done.
#0  0x00007ffff6c9683d in nanosleep () from /lib/x86_64-linux-gnu/libc.so.6
#1  0x00007ffff6cc4774 in usleep () from /lib/x86_64-linux-gnu/libc.so.6
#2  0x00007ffff7b03490 in stream_manager::push (this=0xe413f0, op=...) at stream_manager.cc:383
#3  0x00007ffff79b8b29 in cudaLaunch (hostFun=0x404170 "\351\v\377\377\377ff.\017\037\204")
    at cuda_runtime_api.cc:922
#4  0x0000000000404167 in __device_stub__Z6KernelP4NodePiPbS2_S1_S2_i(Node*, int*, bool*,
bool*, int*, bool*, int) ()
#5  0x0000000000404705 in BFSGraph(int, char**) ()
#6  0x0000000000404b8d in main ()
```

Switch to another thread I got:

```
(gdb) info stack
#0  shader_core_ctx::writeback (this=0x983830) at shader.cc:909
#1  0x00007ffff7a73d10 in shader_core_ctx::cycle (this=0x983830) at shader.cc:1713
#2  0x00007ffff7a7656f in simt_core_cluster::core_cycle (this=0x942440) at shader.cc:2307
#3  0x00007ffff7a5d257 in gpgpu_sim::cycle (this=0x6281b0) at gpu-sim.cc:885
#4  0x00007ffff7af8f8a in gpgpu_sim_thread_concurrent () at gpgpusim_entrypoint.cc:115
#5  0x00007ffff69c1e9a in start_thread () from /lib/x86_64-linux-gnu/libpthread.so.0
#6  0x00007ffff6ccacbd in clone () from /lib/x86_64-linux-gnu/libc.so.6
#7  0x0000000000000000 in ?? ()
```

Part 2. Getting Into the Entry Function __cudaRegisterFatBinary()

From Part 1 we know the entry function in the gpgpu-sim is *__cudaRegisterFatBinary()*. Let's get start from here. This function first calls *CUctx_st *context = GPGPUSim_Context();*, which again calls *GPGPUSim_Init()*. The code of the *GPGPUSim_Context()* function is simple and listed as follows:

```
static CUctx_st* GPGPUSim_Context()
{
     static CUctx_st *the_context = NULL;
     if( the_context == NULL ) {
         _cuda_device_id *the_gpu = GPGPUSim_Init();
         the_context = new CUctx_st(the_gpu);
     }
     return the_context;
}
```

Note that the_context is static thus there is only one copy of it and the context creation (*the_context = new CUctx_st(the_gpu);*) and the *GPGPUSim_Init();* are only executed once, no matter how many kernels are launched. From the above code we can see that a new context is created and returned by *GPGPUSim_Context()* with the argument of the_gpu, which has a type of *_cuda_device_id*. *_cuda_device_id* is a struct defined in cuda_runtime_api.cc and it organizes various device IDs in a linked list (it has a struct _cuda_device_id *m_next member pointing to the next id.). It also has an unsigned m_id data member used to retrieve a particular device:

```
struct _cuda_device_id *get_device( unsigned n )
    {
        assert( n < (unsigned)num_devices() );
        struct _cuda_device_id *p=this;
        for(unsigned i=0; i<n; i++)
            p = p->m_next;
        return p;
    }
```

The most important member in the _cuda_device_id struct is the m_gpgpu, which has a type of class gpgpu_sim *. gpgpu_sim is a class that defines all importance interfaces that GPGPU-SIM provides, such as gpu configuration, statistics collection, simulation control, etc. So how and when is an object of gpgpu_sim created? By looking at the gdb stack trace information and the GPGPU-SIM source code, we can find that *GPGPUSim_Init()* (defined in cuda_runtime_api.cc) invokes *gpgpu_ptx_sim_init_perf()* (defined in gpgpusim_entrypoint.cc), which creates and returns a new gpgpu_sim object by calling option_parser_cmdline() to parse the input gpgpusim.config file. The function option_parser_cmdline() further calls several other functions such as *ParseCommandLine()* and *ParseFile()* (defined in option_parser.cc) to do the actual config file parsing. After creating a gpgpu_sim object, *GPGPUSim_Init()* creates and return a new _cuda_device_id based on the created gpgpu_sim object (the created gpgpu_sim object initializes the m_gpgpu member of the _cuda_device_id object.)

Other than creating a gpgpu_sim object and a corresponding _cuda_device_id from the configuration file, *GPGPUSim_Init()* also calls the start_sim_thread(1) function:

```
void start_sim_thread(int api)
{
   if( g_sim_done ) {
      g_sim_done = false;
      if( api == 1 ) {
         pthread_create(&g_simulation_thread,NULL,gpgpu_sim_thread_concurrent,NULL);
      } else {
         pthread_create(&g_simulation_thread,NULL,gpgpu_sim_thread_sequential,NULL);
      }
   }
}
```

Obviously the *start_sim_thread(1)* will fork a thread to execute the *gpgpu_sim_thread_concurrent()* function (defined in the gpgpusim_entrypoint.cc):

```
void *gpgpu_sim_thread_concurrent(void*)
{
   // concurrent kernel execution simulation thread
   do {
     if(g_debug_execution >= 3) {
       printf("GPGPU-Sim: *** simulation thread starting and spinning waiting for work ***\n");
       fflush(stdout);
     }
      while( g_stream_manager->empty_protected() && !g_sim_done )
          ;
      if(g_debug_execution >= 3) {
         printf("GPGPU-Sim: ** START simulation thread (detected work) **\n");
         g_stream_manager->print(stdout);
         fflush(stdout);
      }
      pthread_mutex_lock(&g_sim_lock);
      g_sim_active = true;
      pthread_mutex_unlock(&g_sim_lock);
      bool active = false;
      bool sim_cycles = false;
      g_the_gpu->init();
      do {
         // check if a kernel has completed
                  // launch operation on device if one is pending and can be run
                  g_stream_manager->operation(&sim_cycles);
               if( g_the_gpu->active() ) {
                  g_the_gpu->cycle();
                  sim_cycles = true;
```

```
                g_the_gpu->deadlock_check();
            }
            active=g_the_gpu->active() || !g_stream_manager->empty_protected();
      } while( active );
      if(g_debug_execution >= 3) {
         printf("GPGPU-Sim: ** STOP simulation thread (no work) **\n");
         fflush(stdout);
      }
      if(sim_cycles) {
          g_the_gpu->update_stats();
          print_simulation_time();
      }
      pthread_mutex_lock(&g_sim_lock);
      g_sim_active = false;
      pthread_mutex_unlock(&g_sim_lock);
   } while( !g_sim_done );
   if(g_debug_execution >= 3) {
      printf("GPGPU-Sim: *** simulation thread exiting ***\n");
      fflush(stdout);
   }
   sem_post(&g_sim_signal_exit);
   return NULL;
}
```

Note that *g_the_gpu* is a global variable defined in gpgpusim_entrypoint.cc and is initialized to point to the newly created *gpgpu_sim* object in *gpgpu_ptx_sim_init_perf()* .

The above *gpgpu_sim_thread_concurrent()* function is executed in a separate thread and performs the primary simulation work. It communicates with the cuda runtime api (e.g., *cudaLaunch()*) provided by GPGPU-SIM to initiate, control and update the simulation.
So how do *cudaLaunch()* and other cuda runtime APIs communicate the kernel launch and memory copying information to the concurrently executed simulation thread (i.e., *gpgpu_sim_thread_concurrent()*)? In gpgpusim_entrypoint.cc there is a global stream manager definition: *stream_manager *g_stream_manager;* This stream_manager object is declared (*extern stream_manager *g_stream_manager;*) and used by *cudaLaunch(), cudaMemcpy()*, etc., in cuda_runtime_api.cc. Thus, each time a cuda runtime API is invoked, the simulator captures the corresponding event and read some information (e.g., kernel information) and push the information onto the *g_stream_manager*, which can be queried, processed or simulated by the concurrent simulation thread *gpgpu_sim_thread_concurrent()*. The main simulation work is done in the outer do....while loop in *gpgpu_sim_thread_concurrent().*

After creating the context, initialization and forking the simulation thread, *__cudaRegisterFatBinary()* gets into a big IF statement to determine if the cuobjdump tool should be used (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.). If the cuobjdump tool is used, mainly do the following:

```
             unsigned fat_cubin_handle = next_fat_bin_handle;
            next_fat_bin_handle++;
            printf("GPGPU-Sim PTX: __cudaRegisterFatBinary, fat_cubin_handle = %u,
filename=%s\n", fat_cubin_handle, filename);
            /*!
             * This function extracts all data from all files in first call
             * then for next calls, only returns the appropriate number
             */
            assert(fat_cubin_handle >= 1);
            if (fat_cubin_handle==1) cuobjdumpInit();
            cuobjdumpRegisterFatBinary(fat_cubin_handle, filename);

            return (void**)fat_cubin_handle;
```

Otherwise, execute:

```
             if( found  ) {
                  printf("GPGPU-Sim PTX: Loading PTX for %s, capability = %s\n",
                              info->ident, info->ptx[selected_capability].gpuProfileName );
                  symbol_table *symtab;
                  const char *ptx = info->ptx[selected_capability].ptx;
                  if(context->get_device()->get_gpgpu()->get_config().convert_to_ptxplus() ) {
               printf("GPGPU-Sim PTX: ERROR ** PTXPlus is only supported through cuobjdump\n"
            "\tEither enable cuobjdump or disable PTXPlus in your configuration file\n");
                        exit(1);
                  } else {
                        symtab=gpgpu_ptx_sim_load_ptx_from_string(ptx,source_num);
                        context->add_binary(symtab,fat_cubin_handle);
                        gpgpu_ptxinfo_load_from_string( ptx, source_num );
                  }
                  source_num++;
                  load_static_globals(symtab,STATIC_ALLOC_LIMIT,0xFFFFFFFF,context-
      >get_device()->get_gpgpu());
                  load_constants(symtab,STATIC_ALLOC_LIMIT,context->get_device()->get_gpgpu());
              } else {
                  printf("GPGPU-Sim PTX: warning -- did not find an appropriate PTX in cubin\n");
              }
            return (void**)fat_cubin_handle;
```

From the above description we know there are two primary cases depending on whether cuobjdump tool is used or not. The GPGPU-SIM document presents a high-level explanation of these two cases:

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 14shows 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.

In the following two subsections let’s dive into these two cases respectively to see how exactly the PTX instructions are extracted.

Part 2.1. Extracting PTX Using cuobjdump

When cuobjdump tool is enabled, the statement *cuobjdumpInit();* does the primary “hard” work. This function is defined as follows:

```
//! Extract the code using cuobjdump and remove unnecessary sections
void cuobjdumpInit(){
    CUctx_st *context = GPGPUSim_Context();
    extract_code_using_cuobjdump(); //extract all the output of cuobjdump to _cuobjdump_*.*
    cuobjdumpSectionList = pruneSectionList(cuobjdumpSectionList, context);
}
```

First let’s take a look at what’s in *extract_code_using_cuobjdump()*

```
//global variables:
std::list<cuobjdumpSection*> cuobjdumpSectionList;
std::list<cuobjdumpSection*> libSectionList;
…..........................
//partial code for extract_code_using_cuobjdump():
…..........................
std::stringstream cmd;
cmd << "ldd /proc/" << pid.str() << "/exe | grep $CUDA_INSTALL_PATH | awk \'{print $3}\' >
_tempfile_.txt";
int result = system(cmd.str().c_str());
std::ifstream libsf;
libsf.open("_tempfile_.txt");
…....................
            std::string line;
        std::getline(libsf, line);
        std::cout << "DOING: " << line << std::endl;
        int cnt=1;
        while(libsf.good()){
            std::stringstream libcodfn;
            libcodfn << "_cuobjdump_complete_lib_" << cnt << "_";
            cmd.str(""); //resetting
            cmd << "$CUDA_INSTALL_PATH/bin/cuobjdump -ptx -elf -sass ";
            cmd << line;
            cmd << " > ";
            cmd << libcodfn.str();
            std::cout << "Running cuobjdump on " << line << std::endl;
            std::cout << "Using command: " << cmd.str() << std::endl;
            result = system(cmd.str().c_str());
            if(result) {printf("ERROR: Failed to execute: %s\n", command); exit(1);}
            std::cout << "Done" << std::endl;

            std::cout << "Trying to parse " << libcodfn << std::endl;
            cuobjdump_in = fopen(libcodfn.str().c_str(), "r");
            cuobjdump_parse();
            fclose(cuobjdump_in);
            std::getline(libsf, line);
        }
        libSectionList = cuobjdumpSectionList;
```

So from above code we should see the *extract_code_using_cuobjdump()* uses *system()* to execute cuobjdump command with necessary library support and environment variable settings. The result is stored in *libcodfn << "_cuobjdump_complete_lib_" << cnt << "_";* file by the following statements:

```
cmd << "$CUDA_INSTALL_PATH/bin/cuobjdump -ptx -elf -sass ";
cmd << line;
cmd << " > ";
cmd << libcodfn.str();
```

Note that *cuobjdump_parse();* is called to parse the output of the cuobjdump tool. This function is automatically generated by yacc and lex tool based on the .y and .l files in the gpgpu-sim\v3.x\libcuda\ directory. The code comments describe this function like this:

```
//! Call cuobjdump to extract everything (-elf -sass -ptx)/*!
 *      This Function extract the whole PTX (for all the files) using cuobjdump
 *      to _cuobjdump_complete_output_XXXXXX then runs a parser to chop it up with each binary in
 *      its own file
 *      It is also responsible for extracting the libraries linked to the binary if the option is
 *      enabled
 * */
```

After *extract_code_using_cuobjdump();* is executed in *cuobjdumpInit()* the statement *cuobjdumpSectionList = pruneSectionList(cuobjdumpSectionList, context);* removes the unnecessary sections.

After *cuobjdumpInit()* is executed the next statement is *cuobjdumpRegisterFatBinary(fat_cubin_handle, filename);*, which simply does the following

```
//! Keep track of the association between filename and cubin handle
void cuobjdumpRegisterFatBinary(unsigned int handle, char* filename){
    fatbinmap[handle] = filename;
}
```

to keep track filename and the corresponding cubin handle.

Now, *__cudaRegisterFunction(...)* function is invoked by application for each device function (nvcc injects it into the source code to get the handle to the compiled cubin and to register the program with the runtime.). 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). *__cudaRegisterFunction(...)* further calls *cuobjdumpParseBinary():*

```
//! Either submit PTX for simulation or convert SASS to PTXPlus and submit it
void cuobjdumpParseBinary(unsigned int handle){

    if(fatbin_registered[handle]) return;
    fatbin_registered[handle] = true;
    CUctx_st *context = GPGPUSim_Context();

    std::string fname = fatbinmap[handle];
    cuobjdumpPTXSection* ptx = findPTXSection(fname);

    symbol_table *symtab;
    char *ptxcode;
    const char *override_ptx_name = getenv("PTX_SIM_KERNELFILE");
  if (override_ptx_name == NULL or getenv("PTX_SIM_USE_PTX_FILE") == NULL) {
        ptxcode = readfile(ptx->getPTXfilename());
    } else {
        printf("GPGPU-Sim PTX: overriding embedded ptx with '%s' (PTX_SIM_USE_PTX_FILE is
set)\n", override_ptx_name);
        ptxcode = readfile(override_ptx_name);
    }
    if(context->get_device()->get_gpgpu()->get_config().convert_to_ptxplus() ) {
        cuobjdumpELFSection* elfsection = findELFSection(ptx->getIdentifier());
        assert (elfsection!= NULL);
        char *ptxplus_str = gpgpu_ptx_sim_convert_ptx_and_sass_to_ptxplus(
                        ptx->getPTXfilename(),
                        elfsection->getELFfilename(),
                        elfsection->getSASSfilename());
        symtab=gpgpu_ptx_sim_load_ptx_from_string(ptxplus_str, handle);
        printf("Adding %s with cubin handle %u\n", ptx->getPTXfilename().c_str(), handle);
        context->add_binary(symtab, handle);
        gpgpu_ptxinfo_load_from_string( ptxcode, handle);
        delete[] ptxplus_str;
    } else {
        symtab=gpgpu_ptx_sim_load_ptx_from_string(ptxcode, handle);
        printf("Adding %s with cubin handle %u\n", ptx->getPTXfilename().c_str(), handle);
        context->add_binary(symtab, handle);
        gpgpu_ptxinfo_load_from_string( ptxcode, handle);
    }
    load_static_globals(symtab,STATIC_ALLOC_LIMIT,0xFFFFFFFF,context->get_device()-
>get_gpgpu());
    load_constants(symtab,STATIC_ALLOC_LIMIT,context->get_device()->get_gpgpu());

    //TODO: Remove temporarily files as per configurations
}
```

As we can see *cuobjdumpParseBinary()* calls *gpgpu_ptx_sim_convert_ptx_and_sass_to_ptxplus() , gpgpu_ptx_sim_load_ptx_from_string , context->add_binary(symtab, handle);* and *gpgpu_ptxinfo_load_from_string(ptxcode, handle);* if the PTX/SASS should be converted to PTXPlus.
Otherwise *symtab=gpgpu_ptx_sim_load_ptx_from_string(ptxcode, handle);, context->add_binary(symtab, handle);* and *gpgpu_ptxinfo_load_from_string(ptxcode, handle);* are executed.
Here is a paragraph for describing these function in the GPGPU-SIM document:

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 adds 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(...) is invoked to to create PTXPlus.

Working on.... Note that *gpgpu_ptx_sim_load_ptx_from_string() (*defined in ptx_loader.cc) does the primary PTX parsing work and deserves an in-depth investigation. Also, it is important to make clear how the branch instruction, divergance and post dominator are searched. As stated in part 1, the function ptx_assemble() is highly related to the PTX parsing, branch instruction detection and divergance analysis and thus it is critical to clarify when ptx_assemble() is called and what this function does in details. To answer all the above questions, we get start from *gpgpu_ptx_sim_load_ptx_from_string()* since this function does the actual PTX parsing work and also invokes the ptx_assemble() function, as can be verified by the stack information:

```
 (gdb) info stack
#0  end_function () at ptx_parser.cc:195
#1  0x00002aaaaab30d1c in ptx_parse () at ptx.y:211
#2  0x00002aaaaab3e6b2 in gpgpu_ptx_sim_load_ptx_from_string (
    p=0xe49e00 "\t.version 1.4\n\t.target sm_10, map_f64_to_f32\n\t// compiled with
/afs/cs.pitt.edu/usr0/yongli/gpgpu-sim/cuda/open64/lib//be\n\t// nvopencc 4.0 built on 2011-05-
13\n\n\t//", '-' <repeats 37 times>..., source_num=1)
    at ptx_loader.cc:164
#3  0x00002aaaaaafc3ca in useCuobjdump () at cuda_runtime_api.cc:1377 [1]
#4  0x00002aaaaaafcc59 in __cudaRegisterFatBinary (fatCubin=0x6197f0)
    at cuda_runtime_api.cc:1421
#5  0x000000000040369e in
__sti____cudaRegisterAll_49_tmpxft_0000028b_00000000_6_bfs_compute_10_cpp1_ii_29cadddc()
()
#6  0x0000000000411ad6 in __do_global_ctors_aux ()
…..................
```

Before we get into the source code for *gpgpu_ptx_sim_load_ptx_from_string()*, we need to know what arguments are passed to *gpgpu_ptx_sim_load_ptx_from_string()* and the cuobjdump file format.

The first parameter of *gpgpu_ptx_sim_load_ptx_from_string()* is passed with *ptxcode*, which is derived as follows:

```
        std::string fname = fatbinmap[handle];
        cuobjdumpPTXSection* ptx = findPTXSection(fname);
        ptxcode = readfile(ptx->getPTXfilename());
```

In the above code, fname is originally derived from the following code

```
        typedef struct {int m; int v; const unsigned long long* d; char* f;} __fatDeviceText
__attribute__ ((aligned (8)));
        __fatDeviceText * fatDeviceText = (__fatDeviceText *) fatCubin;

        // Extract the source code file name that generate the given FatBin.
        // - Obtains the pointer to the actual fatbin structure from the FatBin handle (fatCubin).
        // - An integer inside the fatbin structure contains the relative offset to the source code file
name.
        // - This offset differs among different CUDA and GCC versions.
        char * pfatbin = (char*) fatDeviceText->d;
        int offset = *((int*)(pfatbin+48));
        char * filename = (pfatbin+16+offset);

        // The extracted file name is associated with a fat_cubin_handle passed
        // into cudaLaunch(). Inside cudaLaunch(), the associated file name is
        // used to find the PTX/SASS section from cuobjdump, which contains the
        // PTX/SASS code for the launched kernel function.
        // This allows us to work around the fact that cuobjdump only outputs the
        // file name associated with each section.
…...........................................................................................
if (fat_cubin_handle==1) cuobjdumpInit();
        cuobjdumpRegisterFatBinary(fat_cubin_handle, filename);
```

and has been put into an associated array *fatbinmap* by cuobjdumpRegisterFatBinary(). The statement *cuobjdumpPTXSection* ptx = findPTXSection(fname);* looks for the section named *fname* in a global section list *std::list<cuobjdumpSection*> cuobjdumpSectionList;* This global section list is generated by *cuobjdump_parse(),* which is a function generated by cuobjdump.l and cuobjdump.y, through the function *addCuobjdumpSection()*.By examining the following code snapshot from the .l and .y files things should be clear:

cuobjdump.l:

```
…................
"Fatbin ptx code:"{newline}     {
        yy_push_state(ptxcode);
        yy_push_state(header);
        yylval.string_value = strdup(yytext);
        return PTXHEADER;
}
"Fatbin elf code:"{newline}     {
        yy_push_state(elfcode);
        yy_push_state(header);
        yylval.string_value = strdup(yytext);
        return ELFHEADER;
}
…...............
```

cuobjdump.y:

```
…................
section :       PTXHEADER {
                        addCuobjdumpSection(0);
                        snprintf(filename, 1024, "_cuobjdump_%d.ptx", ptxserial++);
                        ptxfile = fopen(filename, "w");
                        setCuobjdumpptxfilename(filename);
                } headerinfo ptxcode {
                        fclose(ptxfile);
                }
        |       ELFHEADER {
                        addCuobjdumpSection(1);
                        snprintf(filename, 1024, "_cuobjdump_%d.elf", elfserial);
                        elffile = fopen(filename, "w");
                        setCuobjdumpelffilename(filename);
                } headerinfo elfcode {
                        fclose(elffile);
                        snprintf(filename, 1024, "_cuobjdump_%d.sass", elfserial++);
                        sassfile = fopen(filename, "w");
                        setCuobjdumpsassfilename(filename);
                } sasscode {
                        fclose(sassfile);
…................
```

(At this point it is clear that after using the cuobjdump tool to extract information from the binary and invoking *cuobjdump_parse();* to segregate ptx, elf and sass information into different files, "sections" are created and pushed into a global section list (*cuobjdumpSectionList*) and each section is attached with an appropriate ptx/elf/sass filename (not the identifier name, which is extracted from fatbin) and relevant arch information.)

Therefore, the first argument passed to *gpgpu_ptx_sim_load_ptx_from_string(), ptxcode,* is the name for the ptx file generated during *cuobjdump_parse()*.

The second parameter of *gpgpu_ptx_sim_load_ptx_from_string()* is passed with a *handle*, which associates an identifier (or the source file name?) in *fatbinmap.*

Based on the knowledge of section concept, cuobjdump , ptx/sass/elf files, the *handle* and *ptxcode* arguments, now let's explore the function *gpgpu_ptx_sim_load_ptx_from_string()*, which parses and analyzes the PTX code and creates symbol tables.

Part 2.1.1 PTX Parsing: Start From *gpgpu_ptx_sim_load_ptx_from_string*

As our convention, we first copy the code here (from gpgpu-sim\v3.x\src\cuda-sim\ptx_loader.cc):

```
symbol_table *gpgpu_ptx_sim_load_ptx_from_string( const char *p, unsigned source_num )
{
   char buf[1024];
   snprintf(buf,1024,"_%u.ptx", source_num );
   if( g_save_embedded_ptx ) {
      FILE *fp = fopen(buf,"w");
      fprintf(fp,"%s",p);
      fclose(fp);
   }
   symbol_table *symtab=init_parser(buf);
   ptx__scan_string(p);
   int errors = ptx_parse ();
   if ( errors ) {
       char fname[1024];
       snprintf(fname,1024,"_ptx_errors_XXXXXX");
       int fd=mkstemp(fname);
       close(fd);
       printf("GPGPU-Sim PTX: parser error detected, exiting... but first extracting .ptx to \"%s\"\n",
fname);
       FILE *ptxfile = fopen(fname,"w");
       fprintf(ptxfile,"%s", p );
       fclose(ptxfile);
       abort();
       exit(40);
   }

   if ( g_debug_execution >= 100 )
      print_ptx_file(p,source_num,buf);

   printf("GPGPU-Sim PTX: finished parsing EMBEDDED .ptx file %s\n",buf);
   return symtab;
}
```

*I*n the above function a huge majority of the work is done by the *ptx_parse ()* function call. This function is automatically generated from ptx.y and ptx.l files located in gpgpu-sim\v3.x\src\cuda-sim\ directory. The function name is originally *yyparse* (the default name for the parser function generated by the bison tool) and replaced by *ptx_parse* in the generated ptx.tab.c file:

```
/* Substitute the variable and function names.  */
#define yyparse ptx_parse
#define yylex   ptx_lex
#define yyerror ptx_error
#define yylval  ptx_lval
#define yychar  ptx_char
#define yydebug ptx_debug
#define yynerrs ptx_nerrs
```

The ptx.tab.c file is generated from ptx.y file, as can be verified by the following rule in the Makefile in the ...\cuda-sim\ directory:

```
ptx.tab.c: ptx.y
       bison --name-prefix=ptx_ -v -d ptx.y
```

Thus, to understand what *ptx_parse()* does it is necessary to dig into the ptx.y and ptx.l files.

At a high level, ptx.y file contains various rules specifying the construction of ptx file format. It recognizes valid tokens (.file .entry directives, types, instruction opcode, etc.) through the rules defined in ptx.l file and builds ptx grammar from ground up. The following code example from the ptx.y file shows the structure of *statement_list*:

```
statement_list: directive_statement { add_directive(); }
        | instruction_statement { add_instruction(); }
        | statement_list directive_statement { add_directive(); }
        | statement_list instruction_statement { add_instruction(); }
        | statement_list statement_block
        | statement_block
        ;

instruction_statement:  instruction SEMI_COLON
        | IDENTIFIER COLON { add_label($1); }
        | pred_spec instruction SEMI_COLON;

instruction: opcode_spec LEFT_PAREN operand RIGHT_PAREN { set_return(); } COMMA operand
COMMA LEFT_PAREN operand_list RIGHT_PAREN
        | opcode_spec operand COMMA LEFT_PAREN operand_list RIGHT_PAREN
        | opcode_spec operand COMMA LEFT_PAREN RIGHT_PAREN
        | opcode_spec operand_list
        | opcode_spec
        ;
```

As different components are detected, corresponding functions defined in gpgpu-sim\v3.x\src\cuda-sim\ptx_parser.cc are called to process the component and store the relevant information for later use. In ptx_parser.cc there are lots of different functions for processing different ptx components. For example, *add_instruction() add_variables() set_variable_type() add_identifier() add_function_arg(),* etc. Specifically, if an instruction is detected, *add_instruction()* is called to add the detected instruction into a temporary list *static std::list<ptx_instruction*> g_instructions:*

```
void add_instruction()
{
   DPRINTF("add_instruction: %s", ((g_opcode>0)?g_opcode_string[g_opcode]:"<label>") );
   assert( g_shader_core_config != 0 );
   ptx_instruction *i = new ptx_instruction( g_opcode,
                                             g_pred,
                                             g_neg_pred,
                                             g_pred_mod,
                                             g_label,
                                             g_operands,
                                             g_return_var,
                                             g_options,
                                             g_scalar_type,
                                             g_space_spec,
                                             g_filename,
                                             ptx_lineno,
                                             linebuf,
                                             g_shader_core_config );
   g_instructions.push_back(i);
   g_inst_lookup[g_filename][ptx_lineno] = i;
   init_instruction_state();
}
```

After an entire function is processed, the *end_function()* is called:

```
function_defn: function_decl { set_symtab($1); func_header(".skip"); } statement_block {
end_function(); }
        | function_decl { set_symtab($1); } block_spec_list { func_header(".skip"); } statement_block {
end_function(); }
        ;

void end_function()
{
   DPRINTF("end_function");

   init_directive_state();
   init_instruction_state();
   g_max_regs_per_thread = mymax( g_max_regs_per_thread, (g_current_symbol_table-
>next_reg_num()-1));
   g_func_info->add_inst( g_instructions );
   g_instructions.clear();
   gpgpu_ptx_assemble( g_func_info->get_name(), g_func_info );
   g_current_symbol_table = g_global_symbol_table;

   DPRINTF("function %s, PC = %d\n", g_func_info->get_name().c_str(), g_func_info-
>get_start_PC());
}
```

As we can see *end_function()* executes *g_func_info->add_inst(g_instructions)* :

```
void add_inst( const std::list<ptx_instruction*> &instructions )
   {
      m_instructions = instructions;
   }
```

to put all the parsed instructions so far into the *m_instructions* member in the object pointed by *g_func_info* pointer (*g_func_info* is a static pointer defined in ptx_parser.cc). *g_func_info* is initialized by the *g_global_symbol_table->add_function_decl(name, g_entry_point, &g_func_info,*

&g_current_symbol_table) statement in the *add_function_name()* function. The *add_function_decl()* actually news a *function_info* object and passes it to the *g_func_info* pointer. The *g_global_symbol_table-* object, on which the *add_function_decl()* is called, in created in *symbol_table *init_parser()* function, which is called in the *gpgpu_ptx_sim_load_ptx_from_string* function. All the relevant code is listed as follows:

```
static symbol_table *g_global_symbol_table = NULL;

symbol_table *init_parser( const char *ptx_filename )
{
  g_filename = strdup(ptx_filename);
  if  (g_global_allfiles_symbol_table == NULL) {
        g_global_allfiles_symbol_table = new symbol_table("global_allfiles", 0, NULL);
        g_global_symbol_table = g_current_symbol_table = g_global_allfiles_symbol_table;
  }
  else {
        g_global_symbol_table = g_current_symbol_table = new
symbol_table("global",0,g_global_allfiles_symbol_table);
  }

symbol_table *gpgpu_ptx_sim_load_ptx_from_string( const char *p, unsigned source_num )
{
…............
   symbol_table *symtab=init_parser(buf);
}

void add_function_name( const char *name )
{
  DPRINTF("add_function_name %s %s", name,  ((g_entry_point==1)?"(entrypoint)":
((g_entry_point==2)?"(extern)":"")));
  bool prior_decl = g_global_symbol_table->add_function_decl( name, g_entry_point, &g_func_info,
&g_current_symbol_table );
  if( g_add_identifier_cached__identifier ) {
    add_identifier( g_add_identifier_cached__identifier,
                g_add_identifier_cached__array_dim,
                g_add_identifier_cached__array_ident );
    free( g_add_identifier_cached__identifier );
    g_add_identifier_cached__identifier = NULL;
    g_func_info->add_return_var( g_last_symbol );
    init_directive_state();
  }
  if( prior_decl ) {
    g_func_info->remove_args();
  }
  g_global_symbol_table->add_function( g_func_info, g_filename, ptx_lineno );
}

bool symbol_table::add_function_decl( const char *name, int entry_point, function_info **func_info,
symbol_table **sym_table )
{
  std::string key = std::string(name);
  bool prior_decl = false;
  if( m_function_info_lookup.find(key) != m_function_info_lookup.end() ) {
    *func_info = m_function_info_lookup[key];
    prior_decl = true;
  } else {
    *func_info = new function_info(entry_point);
    (*func_info)->set_name(name);
    m_function_info_lookup[key] = *func_info;
  }

  if( m_function_symtab_lookup.find(key) != m_function_symtab_lookup.end() ) {
    assert( prior_decl );
    *sym_table = m_function_symtab_lookup[key];
  } else {
    assert( !prior_decl );
    *sym_table = new symbol_table( "", entry_point, this );
    symbol *null_reg = (*sym_table)->add_variable("_",NULL,0,"",0);
    null_reg->set_regno(0, 0);
    (*sym_table)->set_name(name);
    (*func_info)->set_symtab(*sym_table);
    m_function_symtab_lookup[key] = *sym_table;
    assert( (*func_info)->get_symtab() == *sym_table );
    register_ptx_function(name,*func_info);
  }
  return prior_decl;
}
```

After adding instructions into the *g_func_info* object, *end_function()* calls *gpgpu_ptx_assemble(g_func_info->get_name(), g_func_info)*, which is defined as follows:

```
void gpgpu_ptx_assemble( std::string kname, void *kinfo )
{
   function_info *func_info = (function_info *)kinfo;
   if((function_info *)kinfo == NULL) {
      printf("GPGPU-Sim PTX: Warning - missing function definition \'%s\'\n", kname.c_str());
      return;
   }
   if( func_info->is_extern() ) {
      printf("GPGPU-Sim PTX: skipping assembly for extern declared function \'%s\'\n", func_info-
>get_name().c_str() );
      return;
   }
   func_info->ptx_assemble();
}
```

g*pgpu_ptx_assemble()* mainly executes *func_info->ptx_assemble(),* which calls *function_info::ptx_assemble(),* which does a lot of things including putting instructions into *m_instr_mem[]*,creating the pc-instruction map *s_g_pc_to_insn* , analyzing the basic block information, branch/divergence information by searching the post-dominators, computing target pc for branch instructions, etc. All these analyzed information are stored in appropriate members in the *function_info* structure. The *function_info::ptx_assemble()* function is defined in gpgpu-sim\v3.x\src\cuda-sim\cuda-sim.cc and its source code is listed here:

```
void function_info::ptx_assemble()
{
  if( m_assembled ) {
     return;
  }

  // get the instructions into instruction memory...
  unsigned num_inst = m_instructions.size();
  m_instr_mem_size = MAX_INST_SIZE*(num_inst+1);
  m_instr_mem = new ptx_instruction*[ m_instr_mem_size ];

  printf("GPGPU-Sim PTX: instruction assembly for function \'%s\'... ", m_name.c_str() );
  fflush(stdout);
  std::list<ptx_instruction*>::iterator i;

  addr_t PC = g_assemble_code_next_pc; // globally unique address (across functions)
  // start function on an aligned address
  for( unsigned i=0; i < (PC%MAX_INST_SIZE); i++ )
     s_g_pc_to_insn.push_back((ptx_instruction*)NULL);
  PC += PC%MAX_INST_SIZE;
  m_start_PC = PC;

  addr_t n=0; // offset in m_instr_mem
  s_g_pc_to_insn.reserve(s_g_pc_to_insn.size() + MAX_INST_SIZE*m_instructions.size());
  for ( i=m_instructions.begin(); i != m_instructions.end(); i++ ) {
     ptx_instruction *pI = *i;
     if ( pI->is_label() ) {
        const symbol *l = pI->get_label();
        labels[l->name()] = n;
     } else {
        g_pc_to_finfo[PC] = this;
        m_instr_mem[n] = pI;
        s_g_pc_to_insn.push_back(pI);
        assert(pI == s_g_pc_to_insn[PC]);
        pI->set_m_instr_mem_index(n);
        pI->set_PC(PC);
        assert( pI->inst_size() <= MAX_INST_SIZE );
        for( unsigned i=1; i < pI->inst_size(); i++ ) {
           s_g_pc_to_insn.push_back((ptx_instruction*)NULL);
           m_instr_mem[n+i]=NULL;
        }
        n  += pI->inst_size();
        PC += pI->inst_size();
     }
  }
  g_assemble_code_next_pc=PC;
  for ( unsigned ii=0; ii < n; ii += m_instr_mem[ii]->inst_size() ) { // handle branch instructions
```

```
    ptx_instruction *pI = m_instr_mem[ii];
    if ( pI->get_opcode() == BRA_OP || pI->get_opcode() == BREAKADDR_OP  || pI->get_opcode()
== CALLP_OP) {
       operand_info &target = pI->dst(); //get operand, e.g. target name
       if ( labels.find(target.name()) == labels.end() ) {
          printf("GPGPU-Sim PTX: Loader error (%s:%u): Branch label \"%s\" does not appear in
assembly code.",
                 pI->source_file(),pI->source_line(), target.name().c_str() );
          abort();
       }
       unsigned index = labels[ target.name() ]; //determine address from name
       unsigned PC = m_instr_mem[index]->get_PC();
       m_symtab->set_label_address( target.get_symbol(), PC );
       target.set_type(label_t);
    }
  }

  printf("  done.\n");
  fflush(stdout);
  printf("GPGPU-Sim PTX: finding reconvergence points for \'%s\'...\n", m_name.c_str() );

  create_basic_blocks();
  connect_basic_blocks();
  bool modified = false;
  do {
    find_dominators();
    find_idominators();
    modified = connect_break_targets();
  } while (modified == true);

  if ( g_debug_execution>=50 ) {
    print_basic_blocks();
    print_basic_block_links();
    print_basic_block_dot();
  }
  if ( g_debug_execution>=2 ) {
    print_dominators();
  }
  find_postdominators();
  find_ipostdominators();
  if ( g_debug_execution>=50 ) {
    print_postdominators();
    print_ipostdominators();
  }

  printf("GPGPU-Sim PTX: pre-decoding instructions for \'%s\'...\n", m_name.c_str() );
  for ( unsigned ii=0; ii < n; ii += m_instr_mem[ii]->inst_size() ) { // handle branch instructions
    ptx_instruction *pI = m_instr_mem[ii];
    pI->pre_decode();
  }
  printf("GPGPU-Sim PTX: ... done pre-decoding instructions for \'%s\'.\n", m_name.c_str() );
  fflush(stdout);

  m_assembled = true;
}
```

Part 2.2. Extracting PTX Without cuobjdump

When the configuration file indicates cuobjdump tool should not be used then the function *__cudaRegisterFatBinary()* mainly does the following(the other branch is discussed in Part 2.1):

```
symtab=gpgpu_ptx_sim_load_ptx_from_string(ptx,source_num);
                      context->add_binary(symtab,fat_cubin_handle);
                      gpgpu_ptxinfo_load_from_string( ptx, source_num );
…................................
                 source_num++;
                 load_static_globals(symtab,STATIC_ALLOC_LIMIT,0xFFFFFFFF,context-
>get_device()->get_gpgpu());
                 load_constants(symtab,STATIC_ALLOC_LIMIT,context->get_device()->get_gpgpu());
```

The functions invoked in this path looks similar to the functions called in *cuobjdumpParseBinary* discussed in part 2.1 except that this time no PTX/SASS->PTXPlus conversion is performed (e.g., *gpgpu_ptx_sim_convert_ptx_and_sass_to_ptxplus()* is not called).

Part 3. Kernel Launch

Part 2 introduces creating gpu device/context, initialization, creating simulation thread and parsing kernel information in the *__cudaRegisterFatBinary()* function. We haven't entered the simulation code yet. The simulation is triggered when the running application launches a kernel using cuda runtime api such as cudaLaunch(). So what happens in GPGPU-SIM when the benchmark launch a kernel by calling its cudaLaunch() defined in cuda_runtime_api.cc? (Note that for cuda 4.0 and above this function is replaced by cuLaunchKernel(). User code "<<< >>>" for kernel launch will be replaced by cuda runtime API cudaLaunch() or cuLaunchKernel() when cuda source file is compiled. GPGPU-SIM 3.0 currently use cudaLaunch())

To answer this question, we first put the most important lines of code in cudaLaunch() in GPGPU-SIM:

```
kernel_info_t *grid =
gpgpu_cuda_ptx_sim_init_grid(hostFun,config.get_args(),config.grid_dim(),config.block_dim(),context);
    std::string kname = grid->name();
    dim3 gridDim = config.grid_dim();
    dim3 blockDim = config.block_dim();
    printf("GPGPU-Sim PTX: pushing kernel \'%s\' to stream %u, gridDim= (%u,%u,%u) blockDim =
(%u,%u,%u) \n",
                kname.c_str(), stream?stream->get_uid():0,
gridDim.x,gridDim.y,gridDim.z,blockDim.x,blockDim.y,blockDim.z );
    stream_operation op(grid,g_ptx_sim_mode,stream);
    g_stream_manager->push(op);
```

The primary purpose of calling the gpgpu_cuda_ptx_sim_init_grid() function is to create and return a kernel_info_t object, which contains important information about a kernel such the name, the dimension, etc, as described in the GPGPU-SIM manual:

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.

There are three members in kernel_info_t that should be investigated in particular:
class function_info *m_kernel_entry;
std::list<class ptx_thread_info *> m_active_threads;
class memory_space *m_param_mem;

Here is how GPGPU-SIM document says about function_info class:
"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. "

By looking into the code we know that the function_info class provides methods to query and config function informations such as num_args(), add_param_data(), find_postdominators(),ptx_assemble(), const ptx_instruction *get_instruction(), get_start_PC(), etc.
For example find_postdominators() is defined as follows:

```
void function_info::find_postdominators( )
{
  // find postdominators using algorithm of Muchnick's Adv. Compiler Design & Implemmntation Fig
7.14
  printf("GPGPU-Sim PTX: Finding postdominators for \'%s\'...\n", m_name.c_str() );
  fflush(stdout);
  assert( m_basic_blocks.size() >= 2 ); // must have a distinquished exit block
  std::vector<basic_block_t*>::reverse_iterator bb_itr = m_basic_blocks.rbegin();
  (*bb_itr)->postdominator_ids.insert((*bb_itr)->bb_id);  // the only postdominator of the exit block is
the exit
  for (++bb_itr;bb_itr != m_basic_blocks.rend();bb_itr++) { //copy all basic blocks to all postdominator
lists EXCEPT for the exit block
    for (unsigned i=0; i<m_basic_blocks.size(); i++)
      (*bb_itr)->postdominator_ids.insert(i);
  }
  bool change = true;
  while (change) {
    change = false;
    for ( int h = m_basic_blocks.size()-2/*skip exit*/; h >= 0 ; --h ) {
      assert( m_basic_blocks[h]->bb_id == (unsigned)h );
      std::set<int> T;
      for (unsigned i=0;i< m_basic_blocks.size();i++)
        T.insert(i);
      for ( std::set<int>::iterator s = m_basic_blocks[h]->successor_ids.begin();s !=
m_basic_blocks[h]->successor_ids.end();s++)
        intersect(T, m_basic_blocks[*s]->postdominator_ids);
      T.insert(h);
      if (!is_equal(T,m_basic_blocks[h]->postdominator_ids)) {
        change = true;
        m_basic_blocks[h]->postdominator_ids = T;
      }
    }
  }
}
```

In addition to these methods, there are some private members in function_info to represent a function:

```
private:
  unsigned m_uid;
  unsigned m_local_mem_framesize;
  bool m_entry_point;
  bool m_extern;
  bool m_assembled;
  std::string m_name;
  ptx_instruction **m_instr_mem;
  unsigned m_start_PC;
  unsigned m_instr_mem_size;
  std::map<std::string,param_t> m_kernel_params;
  std::map<unsigned,param_info> m_ptx_kernel_param_info;
  const symbol *m_return_var_sym;
  std::vector<const symbol*> m_args;
  std::list<ptx_instruction*> m_instructions;
  std::vector<basic_block_t*> m_basic_blocks;
  std::list<std::pair<unsigned, unsigned> > m_back_edges;
  std::map<std::string,unsigned> labels;
  unsigned num_reconvergence_pairs;

  //Registers/shmem/etc. used (from ptxas -v), loaded from ___.ptxinfo along with ___.ptx
  struct gpgpu_ptx_sim_kernel_info m_kernel_info;

  symbol_table *m_symtab;

  static std::vector<ptx_instruction*> s_g_pc_to_insn; // a direct mapping from PC to instruction
  static unsigned sm_next_uid;
};
```

Now let's move on another important member in kernel_info_t **:** std::list<class ptx_thread_info *> m_active_threads; Obviously it's a list of ptx_thread_info *. Let's first look at the description of ptx_thread_info:

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:

The third member class memory_space *m_param_mem in the kernel_info_t class is described as follows:

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.

After our introduction to the kernel_info_t now let's get back to the gpgpu_cuda_ptx_sim_init_grid() function and look how an object of kernel_info_t is created. In the function gpgpu_cuda_ptx_sim_init_grid() an object of kernel_info_t is created by this statement: kernel_info_t *result = new kernel_info_t(gridDim,blockDim,entry);
, which invoke the constructor:

```
kernel_info_t::kernel_info_t( dim3 gridDim, dim3 blockDim, class function_info *entry )
{
    m_kernel_entry=entry;
    m_grid_dim=gridDim;
    m_block_dim=blockDim;
    m_next_cta.x=0;
    m_next_cta.y=0;
    m_next_cta.z=0;
    m_next_tid=m_next_cta;
    m_num_cores_running=0;
    m_uid = m_next_uid++;
    m_param_mem = new memory_space_impl<8192>("param",64*1024);
}
```

We can see that this constructor initialize some members but not others, most notably the std::list<class ptx_thread_info *> m_active_threads; member. For now just assume it will be initialized later.

One interesting point is in the argument entry passed to the kernel_info_t constructor. This can be made clear by looking at, in the function gpgpu_cuda_ptx_sim_init_grid(), the line before creating the kernel_info_t object:

function_info *entry = context->get_kernel(hostFun);
kernel_info_t *result = new kernel_info_t(gridDim,blockDim,entry);

From above statements it is clear that context->get_kernel() returns entry function for the kernel. The relevant code for get_kernel is listed here for your curiosity:

```
struct CUctx_st {
…....................
        function_info *get_kernel(const char *hostFun)
        {
                std::map<const void*,function_info*>::iterator i=m_kernel_lookup.find(hostFun);
                assert( i != m_kernel_lookup.end() );
                return i->second;
        }

private:
        _cuda_device_id *m_gpu; // selected gpu
        std::map<unsigned,symbol_table*> m_code; // fat binary handle => global symbol table
        unsigned m_last_fat_cubin_handle;
        std::map<const void*,function_info*> m_kernel_lookup; // unique id (CUDA app function
address) => kernel entry point
};
```

From the gdb trace in page 6 and the explanation in page 5(• *For each kernel in the source file, a device function registration function, cudaRegisterFunction() is called. …....*) we know that the *__cudaRegisterFunction* function (in cuda_runtime_api.cc) is called implicitly by the compiled benchmark at the beginning of a kernel launch. In GPGPU-SIM, *__cudaRegisterFunction* invokes context->register_function(fat_cubin_handle, hostFun, deviceFun); to register the kernel entry function based on the hostFun name. It also creates a corresponding function_info object and store the relevant pointer into the m_kernel_lookup, which is the context's hash table (map) member to store function_info pointers associated with kernel functions.

```
        void register_function( unsigned fat_cubin_handle, const char *hostFun, const char *deviceFun )
        {
                if( m_code.find(fat_cubin_handle) != m_code.end() ) {
                        symbol *s = m_code[fat_cubin_handle]->lookup(deviceFun);
                        assert( s != NULL );
                        function_info *f = s->get_pc();
                        assert( f != NULL );
                        m_kernel_lookup[hostFun] = f;
                } else {
                        m_kernel_lookup[hostFun] = NULL;
                }
        }
```

Part 4. Simulation Body: gpgpu_sim_thread_concurrent()

From the analyses in previous chapters we understand the context creation, kernel initialization, entry function information generation, etc. This all happens in *__cudaRegisterFatBinary()* and cudaLaunch(). Once these init work has been done, as the last few lines of code of cudaLaunch() indicates that a stream_operation object is created and pushed onto g_stream_manager, which will trigger the simulation in the *gpgpu_sim_thread_concurrent ()* function. This part gets into and explains this function.

In Part 2 when we explain *GPGPUSim_Init()* , we list code for *gpgpu_sim_thread_concurrent().* Here we list it again to ease our discussion.

```
void *gpgpu_sim_thread_concurrent(void*)
{
   // concurrent kernel execution simulation thread
   do {
      if(g_debug_execution >= 3) {
         printf("GPGPU-Sim: *** simulation thread starting and spinning waiting for work ***\n");
         fflush(stdout);
      }
       while( g_stream_manager->empty_protected() && !g_sim_done )
           ;
       if(g_debug_execution >= 3) {
          printf("GPGPU-Sim: ** START simulation thread (detected work) **\n");
          g_stream_manager->print(stdout);
          fflush(stdout);
       }
       pthread_mutex_lock(&g_sim_lock);
       g_sim_active = true;
       pthread_mutex_unlock(&g_sim_lock);
       bool active = false;
       bool sim_cycles = false;
       g_the_gpu->init();
       do {
          // check if a kernel has completed
                   // launch operation on device if one is pending and can be run
                   g_stream_manager->operation(&sim_cycles);
               if( g_the_gpu->active() ) {
                   g_the_gpu->cycle();
                   sim_cycles = true;
                   g_the_gpu->deadlock_check();
               }
               active=g_the_gpu->active() || !g_stream_manager->empty_protected();
       } while( active );
       if(g_debug_execution >= 3) {
          printf("GPGPU-Sim: ** STOP simulation thread (no work) **\n");
          fflush(stdout);
       }
       if(sim_cycles) {
           g_the_gpu->update_stats();
           print_simulation_time();
       }
       pthread_mutex_lock(&g_sim_lock);
       g_sim_active = false;
       pthread_mutex_unlock(&g_sim_lock);
   } while( !g_sim_done );
   if(g_debug_execution >= 3) {
      printf("GPGPU-Sim: *** simulation thread exiting ***\n");
      fflush(stdout);
   }
   sem_post(&g_sim_signal_exit);
   return NULL;
}
```

The first statement that matters in the above code is *g_the_gpu->init();* The effect of executing this is to initialize simulation cycle, control flow, memory access, statistics of various kinds (simulation results), etc. For source code details please refer to gpu-sim.cc file.
Following the init the simulation gets into a do...while loop highlighted in green. This is the main loop that simulates the running cuda application cycle by cycle. Each iteration of this loop simulates one cycle (by *g_stream_manager->operation(&sim_cycles);* and *g_the_gpu->cycle();*) if the *g_the_gpu* is active:

```
bool gpgpu_sim::active()
{
   if (m_config.gpu_max_cycle_opt && (gpu_tot_sim_cycle + gpu_sim_cycle) >=
m_config.gpu_max_cycle_opt)
      return false;
```

```
  if (m_config.gpu_max_insn_opt && (gpu_tot_sim_insn + gpu_sim_insn) >=
m_config.gpu_max_insn_opt)
     return false;
  if (m_config.gpu_max_cta_opt && (gpu_tot_issued_cta >= m_config.gpu_max_cta_opt) )
     return false;
  if (m_config.gpu_deadlock_detect && gpu_deadlock)
     return false;
  for (unsigned i=0;i<m_shader_config->n_simt_clusters;i++)
     if( m_cluster[i]->get_not_completed()>0 )
        return true;;
  for (unsigned i=0;i<m_memory_config->m_n_mem;i++)
     if( m_memory_partition_unit[i]->busy()>0 )
        return true;;
  if( icnt_busy() )
      return true;
  if( get_more_cta_left() )
      return true;
  return false;
}
```

After the loop finishes the simulation results are updated and printed out (highlighted in yellow) if any cycle(s) have been simulated. So the major work is done in the loop by *g_stream_manager->operation(&sim_cycles);* and *g_the_gpu->cycle();*. Now let's dive into these two statements for further analyses.

Part 4.1 Launch Stream Operations

In this subsection the statement *g_stream_manager->operation(&sim_cycles);* is analyzed in depth. To ease our code study, we first list the surface level code where this statement gets into:

```
void stream_manager::operation( bool * sim)
{
      pthread_mutex_lock(&m_lock);
      bool check=check_finished_kernel();
      if(check) m_gpu->print_stats();
      stream_operation op =front();
      op.do_operation( m_gpu );
      pthread_mutex_unlock(&m_lock);
}
```

To gain a good understanding of what the above code does let's first take some time to get familiar with the structure of *stream_manager.* As early in part 2 we learned that *g_stream_manager (*an object of *stream_manager)* plays an important role in communicating information between *cudaLaunch()* and the actual simulation thread *gpgpu_sim_thread_concurrent()* . It has the following interface:

```
class stream_manager {
public:
  stream_manager( gpgpu_sim *gpu, bool cuda_launch_blocking );
  bool register_finished_kernel(unsigned grid_uid  );
  bool check_finished_kernel(  );
  stream_operation front();
  void add_stream( CUstream_st *stream );
  void destroy_stream( CUstream_st *stream );
  bool concurrent_streams_empty();
  bool empty_protected();
  bool empty();
  void print( FILE *fp);
  void push( stream_operation op );
  void operation(bool * sim);
private:
  void print_impl( FILE *fp);

  bool m_cuda_launch_blocking;
  gpgpu_sim *m_gpu;
  std::list<CUstream_st *> m_streams;
  std::map<unsigned,CUstream_st *> m_grid_id_to_stream;
  CUstream_st m_stream_zero;
  bool m_service_stream_zero;
  pthread_mutex_t m_lock;
};
```

From the above definition we can see *stream_manager* primarily contains a list of CUstream_st (m_streams), a pointer (m_gpu) to the simulation instance (gpgpu_sim), and some flags such as if the cuda launch is blocking or not. The *CUstream_st* is a struct that contains a list of stream_operation objects (*std::list<stream_operation> m_operations;)* and a few methods to manipulate the stream_operation objects in the list. These methods include *bool empty()* (check if the operation list is empty), *bool busy()* (check if the there is an operating is in process), *void synchronize()* (wait in a loop until all the operations are done in the list, i.e., the list is empty)*; void push(const stream_operation &op);* and other trivial operations such as getting the next stream operation*(next()),* getting the front stream operation *(front())* and a printing method *(print())*. The class *stream_operation* has the following definition:

```
class stream_operation {
public:
…..........................
   bool is_kernel() const { return m_type == stream_kernel_launch; }
   bool is_mem() const {
      return m_type == stream_memcpy_host_to_device ||
            m_type == stream_memcpy_device_to_host ||
            m_type == stream_memcpy_host_to_device;
   }
   bool is_noop() const { return m_type == stream_no_op; }
   bool is_done() const { return m_done; }
   kernel_info_t *get_kernel() { return m_kernel; }
   void do_operation( gpgpu_sim *gpu );
   void print( FILE *fp ) const;
   struct CUstream_st *get_stream() { return m_stream; }
   void set_stream( CUstream_st *stream ) { m_stream = stream; }

private:
   struct CUstream_st *m_stream;

   bool m_done;

   stream_operation_type m_type;
   size_t      m_device_address_dst;
   size_t      m_device_address_src;
   void       *m_host_address_dst;
   const void *m_host_address_src;
   size_t      m_cnt;

   const char *m_symbol;
   size_t m_offset;

   bool m_sim_mode;
   kernel_info_t *m_kernel;
   class CUevent_st *m_event;
};
```

Thus, *stream_operation* essentially keeps track of different types of cuda operations such as kernel launch, memory copy, etc. See the following stream operation type definition:

```
enum stream_operation_type {
   stream_no_op,
   stream_memcpy_host_to_device,
   stream_memcpy_device_to_host,
   stream_memcpy_device_to_device,
   stream_memcpy_to_symbol,
   stream_memcpy_from_symbol,
   stream_kernel_launch,
   stream_event
};
```

If it is a kernel launch, *stream_operation* stores the kernel information in its *kernel_info_t *m_kernel;* member.

At this point we should be familiar with the necessary structures and definitions for further code study. Now let's get back to the function *stream_manager::operation(bool * sim)* at the beginning of part 4.1. The first three statements just check if there is a finished kernel and print its statistics if any. Next the first stream operation *op* is popped out from the list using (*stream_operation op =front();*) and then the statement *op.do_operation(m_gpu);* is executed. The code structure of
is like this:

```
void stream_operation::do_operation( gpgpu_sim *gpu )
{
   if( is_noop() )
      return;
   case stream_memcpy_host_to_device:
…...............................
      break;
   case stream_memcpy_device_to_host:
      if(g_debug_execution >= 3)
         printf("memcpy device-to-host\n");
      gpu->memcpy_from_gpu(m_host_address_dst,m_device_address_src,m_cnt);
```

```
        m_stream->record_next_done();
        break;
…..........................
      case stream_kernel_launch:
        if( gpu->can_start_kernel() ) {
            printf("kernel \'%s\' transfer to GPU hardware scheduler\n", m_kernel->name().c_str() );
            if( m_sim_mode )
                gpgpu_cuda_ptx_sim_main_func( *m_kernel );
            else
                gpu->launch( m_kernel );
        }
        break;
    case stream_event: {
        printf("event update\n");
        time_t wallclock = time((time_t *)NULL);
        m_event->update( gpu_tot_sim_cycle, wallclock );
        m_stream->record_next_done();
        }
        break;
    default:
        abort();
    }
    m_done=true;
    fflush(stdout);
}
```

It is obvious the above function (*stream_operation::do_operation*) contains a big switch case statement and gets to different branches based on the type of the stream operation. Since we mainly concern about the kernel launch thus we focus on analyzing the code in yellow background. Based on what mode (performance simulation mode or purely functional simulation mode.) is chosen, these are two paths. Function simulation runs faster but does not collect performance statistics.

The first path (*gpgpu_cuda_ptx_sim_main_func(*m_kernel);*) is executed when the running mode is functional simulation. *gpgpu_cuda_ptx_sim_main_func(*m_kernel)* is defined in cuda-sim.cc:

```
    //we excute the kernel one CTA (Block) at the time, as synchronization functions work block wise
    while(!kernel.no_more_ctas_to_run()){
        functionalCoreSim cta(&kernel, g_the_gpu, g_the_gpu->getShaderCoreConfig()->warp_size);
        cta.execute();
    }
```

About the functional simulation we can refer to the official document:
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.

We focus on tracing the code for performance simulation.
The second path(*gpu->launch(m_kernel)*), which is executed when performance simulation is chosen, put the *kernel_info_t *m_kernel* member of this *stream_operation* object into a “null” or “done” element in the running kernel vector (*std::vector<kernel_info_t*> m_running_kernels;*) member in the *gpgpu_sim* instance. Note that this operations is important since there are lots of further simulation behaviors depending the running kernel vector. Most importantly at this point, putting something in the running kernel vector will result in a TRUE return value upon the *bool gpgpu_sim::active()* function invocation since *bool gpgpu_sim::active()* (source code listed between part 4 and part 4.1) calls *get_more_cta_left(),* which has the following definition:

```
bool gpgpu_sim::get_more_cta_left() const
{
   if (m_config.gpu_max_cta_opt != 0) {
      if( m_total_cta_launched >= m_config.gpu_max_cta_opt )
          return false;
   }
   for(unsigned n=0; n < m_running_kernels.size(); n++ ) {
       if( m_running_kernels[n] && !m_running_kernels[n]->no_more_ctas_to_run() )
           return true;
   }
   return false;
}
```

Thus, in the loop with the green background in the *void *gpgpu_sim_thread_concurrent(void*)* code listed at the beginning of part 4, *g_the_gpu->cycle();* will be executed for performance simulation after *g_stream_manager->operation(&sim_cycles);* in which *gpu->launch(m_kernel)* is executed when performance simulated mode is selected. In the next subsection (part 4.2), we will discuss how a cycle is simulated for performance simulation.

Part 4.2 Simulating One Cycle in Performance Simulation

From the previous discussion we know *g_the_gpu->cycle()* is executed when performance simulation is selected as the running mode for GPGPU-SIM. As we have done in Part 4.1, we first list the code for the statement *g_the_gpu->cycle();*here for further discussion (a little bit long):

```
void gpgpu_sim::cycle()
{
  int clock_mask = next_clock_domain();

  if (clock_mask & CORE ) {
     // shader core loading (pop from ICNT into core) follows CORE clock
    for (unsigned i=0;i<m_shader_config->n_simt_clusters;i++)
       m_cluster[i]->icnt_cycle();
  }
   if (clock_mask & ICNT) {
      // pop from memory controller to interconnect
      for (unsigned i=0;i<m_memory_config->m_n_mem;i++) {
         mem_fetch* mf = m_memory_partition_unit[i]->top();
         if (mf) {
             unsigned response_size = mf->get_is_write()?mf->get_ctrl_size():mf->size();
             if ( ::icnt_has_buffer( m_shader_config->mem2device(i), response_size ) ) {
                 if (!mf->get_is_write())
                    mf->set_return_timestamp(gpu_sim_cycle+gpu_tot_sim_cycle);
                 mf->set_status(IN_ICNT_TO_SHADER,gpu_sim_cycle+gpu_tot_sim_cycle);
                 ::icnt_push( m_shader_config->mem2device(i), mf->get_tpc(), mf, response_size );
                 m_memory_partition_unit[i]->pop();
             } else {
                 gpu_stall_icnt2sh++;
             }
         } else {
            m_memory_partition_unit[i]->pop();
         }
      }
   }

  if (clock_mask & DRAM) {
    for (unsigned i=0;i<m_memory_config->m_n_mem;i++)
       m_memory_partition_unit[i]->dram_cycle(); // Issue the dram command (scheduler + delay
model)
  }

  // L2 operations follow L2 clock domain
  if (clock_mask & L2) {
    for (unsigned i=0;i<m_memory_config->m_n_mem;i++) {
       //move memory request from interconnect into memory partition (if not backed up)
       //Note:This needs to be called in DRAM clock domain if there is no L2 cache in the system
       if ( m_memory_partition_unit[i]->full() ) {
          gpu_stall_dramfull++;
       } else {
           mem_fetch* mf = (mem_fetch*) icnt_pop( m_shader_config->mem2device(i) );
           m_memory_partition_unit[i]->push( mf, gpu_sim_cycle + gpu_tot_sim_cycle );
       }
       m_memory_partition_unit[i]->cache_cycle(gpu_sim_cycle+gpu_tot_sim_cycle);
    }
  }

  if (clock_mask & ICNT) {
    icnt_transfer();  //function pointer to advance_interconnect() defined in interconnect_interface.cc
  }

  if (clock_mask & CORE) {
    // L1 cache + shader core pipeline stages
    for (unsigned i=0;i<m_shader_config->n_simt_clusters;i++) {
       if (m_cluster[i]->get_not_completed() || get_more_cta_left() ) {
             m_cluster[i]->core_cycle();
       }
    }
    if( g_single_step && ((gpu_sim_cycle+gpu_tot_sim_cycle) >= g_single_step) ) {
        asm("int $03");
    }
    gpu_sim_cycle++;
    if( g_interactive_debugger_enabled )
       gpgpu_debug();

    issue_block2core();
    // Flush the caches once all of threads are completed.
    if (m_config.gpgpu_flush_cache) {
       int all_threads_complete = 1 ;
       for (unsigned i=0;i<m_shader_config->n_simt_clusters;i++) {
         if (m_cluster[i]->get_not_completed() == 0)
           m_cluster[i]->cache_flush();
         else
           all_threads_complete = 0 ;
       }
       if (all_threads_complete && !m_memory_config->m_L2_config.disabled() ) {
         printf("Flushed L2 caches...\n");
         if (m_memory_config->m_L2_config.get_num_lines()) {
           int dlc = 0;
           for (unsigned i=0;i<m_memory_config->m_n_mem;i++) {
             dlc = m_memory_partition_unit[i]->flushL2();
             assert (dlc == 0); // need to model actual writes to DRAM here
             printf("Dirty lines flushed from L2 %d is %d\n", i, dlc  );
           }
         }
       }
    }

    if (!(gpu_sim_cycle % m_config.gpu_stat_sample_freq)) {
       time_t days, hrs, minutes, sec;
       time_t curr_time;
       time(&curr_time);
       unsigned long long  elapsed_time = MAX(curr_time - g_simulation_starttime, 1);
       days    = elapsed_time/(3600*24);
       hrs     = elapsed_time/3600 - 24*days;
       minutes = elapsed_time/60 - 60*(hrs + 24*days);
       sec = elapsed_time - 60*(minutes + 60*(hrs + 24*days));
       printf("GPGPU-Sim uArch: cycles simulated: %lld  inst.: %lld (ipc=%4.1f) sim_rate=%u
(inst/sec) elapsed = %u:%u:%02u:%02u / %s",
             gpu_tot_sim_cycle + gpu_sim_cycle, gpu_tot_sim_insn + gpu_sim_insn,
             (double)gpu_sim_insn/(double)gpu_sim_cycle,
             (unsigned)((gpu_tot_sim_insn+gpu_sim_insn) / elapsed_time),
             (unsigned)days,(unsigned)hrs,(unsigned)minutes,(unsigned)sec,
             ctime(&curr_time));
       fflush(stdout);
       visualizer_printstat();
       m_memory_stats->memlatstat_lat_pw();
       if (m_config.gpgpu_runtime_stat && (m_config.gpu_runtime_stat_flag != 0) ) {
         if (m_config.gpu_runtime_stat_flag & GPU_RSTAT_BW_STAT) {
           for (unsigned i=0;i<m_memory_config->m_n_mem;i++)
             m_memory_partition_unit[i]->print_stat(stdout);
           printf("maxmrqlatency = %d \n", m_memory_stats->max_mrq_latency);
           printf("maxmflatency = %d \n", m_memory_stats->max_mf_latency);
         }
         if (m_config.gpu_runtime_stat_flag & GPU_RSTAT_SHD_INFO)
           shader_print_runtime_stat( stdout );
         if (m_config.gpu_runtime_stat_flag & GPU_RSTAT_L1MISS)
           shader_print_l1_miss_stat( stdout );
       }
    }

    if (!(gpu_sim_cycle % 20000)) {
       // deadlock detection
       if (m_config.gpu_deadlock_detect && gpu_sim_insn == last_gpu_sim_insn) {
         gpu_deadlock = true;
       } else {
         last_gpu_sim_insn = gpu_sim_insn;
       }
    }
    try_snap_shot(gpu_sim_cycle);
    spill_log_to_file (stdout, 0, gpu_sim_cycle);
  }
}
```

It would be helpful to first take a look at how the GPGPU-SIM document describes the SIMT cluster, SIMT core, caches and memory class model before tracing into the source code:

4.4.1.1 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.

4.4.1.2 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.

4.4.1.2.1 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.

4.4.1.2.2 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().

4.4.1.2.3 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.

4.4.1.2.4 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.

4.4.1.2.5 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.

4.4.1.2.6 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.

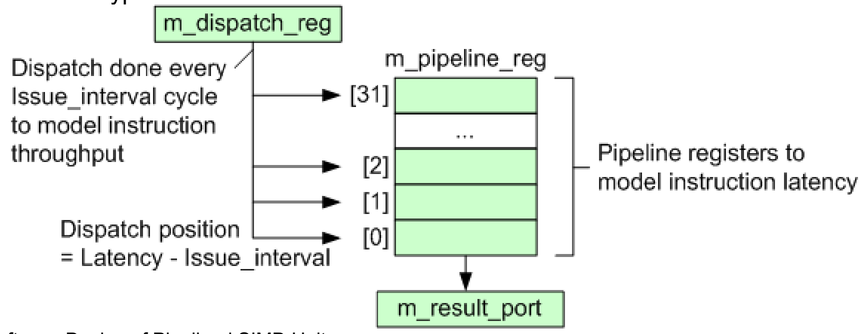

m_dispatch_reg
Dispatch done every Issue_interval cycle to model instruction throughput
Dispatch position = Latency - Issue_interval
m_pipeline_reg
[31]
...
[2]
[1]
[0]
Pipeline registers to model instruction latency
m_result_port

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 atm_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.

4.4.1.2.7 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.

4.4.1.2.8 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 theread_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.

4.4.1.2.9 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.

Now you should have a high-level overview of the software class design for the SIMT core, caches and memories. The structure of the above *gpgpu_sim::cycle()* function indicates the sequence of simulation operations in one cycle (marked in different colors) includes the shader core loading from NoC, pushing memory requests to NoC, advancing DRAM cycle, moving memory requests from NoC to memory partition /L2 operation, shader core pipeline/L1 operation, thread block scheduling and flushing caches (only be done upon kernel completion). Now we study these simulation operations in the following subsections.

Part 4.2.1 Shader Core Loading (pop from NoC into core) & Response FIFO Simulation for SIMT Cluster (code in red)

At a high level, the code in the red background iterates over SIMT core (i.e., shader core) clusters (defined as class simt_core_cluster **m_cluster; in the gpgpu_sim class) and advances a cycle by executing *m_cluster[i]->icnt_cycle().*

Since there are lots of relevant classes and structures related to this part of the simulation, it is necessary to get familiar with them before we introducing *m_cluster[i]->icnt_cycle()*. The following is a list of interfaces of important classes for understanding the simulation details analyzed in this subsection as well as the subsequent subsections. In particular, we list the interfaces for *simt_core_cluster, core_t, shader_core_ctx, ldst_unit, warp_inst_t, inst_t, mem_fetch:*

```
class simt_core_cluster {
public:
   simt_core_cluster( class gpgpu_sim *gpu,
                      unsigned cluster_id,
                      const struct shader_core_config *config,
                      const struct memory_config *mem_config,
                      shader_core_stats *stats,
                      memory_stats_t *mstats );
   void core_cycle();
   void icnt_cycle();
   void reinit();
   unsigned issue_block2core();
   void cache_flush();
   bool icnt_injection_buffer_full(unsigned size, bool write);
   void icnt_inject_request_packet(class mem_fetch *mf);
   // for perfect memory interface
   bool response_queue_full() {
      return ( m_response_fifo.size() >= m_config->n_simt_ejection_buffer_size );
   }
   void push_response_fifo(class mem_fetch *mf) {
      m_response_fifo.push_back(mf);
   }
   unsigned max_cta( const kernel_info_t &kernel );
   unsigned get_not_completed() const;
   void print_not_completed( FILE *fp ) const;
   unsigned get_n_active_cta() const;
   gpgpu_sim *get_gpu() { return m_gpu; }
   void display_pipeline( unsigned sid, FILE *fout, int print_mem, int mask );
   void print_cache_stats( FILE *fp, unsigned& dl1_accesses, unsigned& dl1_misses ) const;
```

```
private:
    unsigned m_cluster_id;
    gpgpu_sim *m_gpu;
    const shader_core_config *m_config;
    shader_core_stats *m_stats;
    memory_stats_t *m_memory_stats;
    shader_core_ctx **m_core;
    unsigned m_cta_issue_next_core;
    std::list<unsigned> m_core_sim_order;
    std::list<mem_fetch*> m_response_fifo;
};



/*
 * This abstract class used as a base for functional and performance and simulation, it has basic
functional simulation data structures and procedures.
 */
class core_t {
    public:
        virtual ~core_t() {}
        virtual void warp_exit( unsigned warp_id ) = 0;
        virtual bool warp_waiting_at_barrier( unsigned warp_id ) const = 0;
        virtual void checkExecutionStatusAndUpdate(warp_inst_t &inst, unsigned t, unsigned tid)=0;
        class gpgpu_sim * get_gpu() {return m_gpu;}
        void execute_warp_inst_t(warp_inst_t &inst, unsigned warpSize, unsigned warpId =
(unsigned)-1);
        bool  ptx_thread_done( unsigned hw_thread_id ) const ;
        void updateSIMTStack(unsigned warpId, unsigned warpSize, warp_inst_t * inst);
        void initilizeSIMTStack(unsigned warps, unsigned warpsSize);
        warp_inst_t getExecuteWarp(unsigned warpId);
        void get_pdom_stack_top_info( unsigned warpId, unsigned *pc, unsigned *rpc ) const;
        kernel_info_t * get_kernel_info(){ return m_kernel;}
    protected:
        class gpgpu_sim *m_gpu;
        kernel_info_t *m_kernel;
        simt_stack  **m_simt_stack; // pdom based reconvergence context for each warp
        class ptx_thread_info ** m_thread;
};



class shader_core_ctx : public core_t {
public:
void cycle();
    void reinit(unsigned start_thread, unsigned end_thread, bool reset_not_completed );
    void issue_block2core( class kernel_info_t &kernel );
    void cache_flush();
    void accept_fetch_response( mem_fetch *mf );
    void accept_ldst_unit_response( class mem_fetch * mf );
    void set_kernel( kernel_info_t *k )
…...........................
private:
    int test_res_bus(int latency);
    void init_warps(unsigned cta_id, unsigned start_thread, unsigned end_thread);
    virtual void checkExecutionStatusAndUpdate(warp_inst_t &inst, unsigned t, unsigned tid);
    address_type next_pc( int tid ) const;
    void fetch();
    void register_cta_thread_exit( unsigned cta_num );
    void decode();
    void issue();
    friend class scheduler_unit; //this is needed to use private issue warp.
    friend class TwoLevelScheduler;
    friend class LooseRoundRobbinScheduler;
    void issue_warp( register_set& warp, const warp_inst_t *pI, const active_mask_t &active_mask,
unsigned warp_id );
    void func_exec_inst( warp_inst_t &inst );
    void read_operands();
    void execute();
    void writeback();
…..............................
// CTA scheduling / hardware thread allocation
    unsigned m_n_active_cta; // number of Cooperative Thread Arrays (blocks) currently running on
this shader.
    unsigned m_cta_status[MAX_CTA_PER_SHADER]; // CTAs status
    unsigned m_not_completed; // number of threads to be completed (==0 when all thread on this
core completed)
    std::bitset<MAX_THREAD_PER_SM> m_active_threads;
    // thread contexts
    thread_ctx_t             *m_threadState;
    // interconnect interface
    mem_fetch_interface *m_icnt;
    shader_core_mem_fetch_allocator *m_mem_fetch_allocator;
    // fetch
    read_only_cache *m_L1I; // instruction cache
    int  m_last_warp_fetched;
    // decode/dispatch
    std::vector<shd_warp_t>   m_warp;   // per warp information array
    barrier_set_t             m_barriers;
    ifetch_buffer_t           m_inst_fetch_buffer;
    std::vector<register_set> m_pipeline_reg;
    Scoreboard                *m_scoreboard;
    opndcoll_rfu_t            m_operand_collector;
    //schedule
    std::vector<scheduler_unit*>  schedulers;
    // execute
    unsigned m_num_function_units;
    std::vector<pipeline_stage_name_t> m_dispatch_port;
    std::vector<pipeline_stage_name_t> m_issue_port;
    std::vector<simd_function_unit*> m_fu; // stallable pipelines should be last in this array
    ldst_unit *m_ldst_unit;
    static const unsigned MAX_ALU_LATENCY = 512;
    unsigned num_result_bus;
    std::vector< std::bitset<MAX_ALU_LATENCY>* > m_result_bus;
    // used for local address mapping with single kernel launch
    unsigned kernel_max_cta_per_shader;
    unsigned kernel_padded_threads_per_cta;
};



class ldst_unit: public pipelined_simd_unit {
public:
    ldst_unit( mem_fetch_interface *icnt,
               shader_core_mem_fetch_allocator *mf_allocator,
               shader_core_ctx *core,
               opndcoll_rfu_t *operand_collector,
               Scoreboard *scoreboard,
               const shader_core_config *config,
               const memory_config *mem_config,
               class shader_core_stats *stats,
               unsigned sid, unsigned tpc );
    // modifiers
    virtual void issue( register_set &inst );
    virtual void cycle();
    void fill( mem_fetch *mf );
    void flush();
    void writeback();
    // accessors
    virtual unsigned clock_multiplier() const;
    virtual bool can_issue( const warp_inst_t &inst ) const
…................//definition of can_issue()
    virtual bool stallable() const { return true; }
    bool response_buffer_full() const;
    void print(FILE *fout) const;
    void print_cache_stats( FILE *fp, unsigned& dl1_accesses, unsigned& dl1_misses );
private:
   bool shared_cycle();
   bool constant_cycle();
   bool texture_cycle();
   bool memory_cycle();
   mem_stage_stall_type process_memory_access_queue( cache_t *cache, warp_inst_t &inst );
   const memory_config *m_memory_config;
   class mem_fetch_interface *m_icnt;
   shader_core_mem_fetch_allocator *m_mf_allocator;
   class shader_core_ctx *m_core;
   unsigned m_sid;
   unsigned m_tpc;
   tex_cache *m_L1T; // texture cache
   read_only_cache *m_L1C; // constant cache
   l1_cache *m_L1D; // data cache
```

```
  std::map<unsigned/*warp_id*/, std::map<unsigned/*regnum*/,unsigned/*count*/> >
m_pending_writes;
  std::list<mem_fetch*> m_response_fifo;
  opndcoll_rfu_t *m_operand_collector;
  Scoreboard *m_scoreboard;
  mem_fetch *m_next_global;
  warp_inst_t m_next_wb;
  unsigned m_writeback_arb; // round-robin arbiter for writeback contention between L1T, L1C,
shared
  unsigned m_num_writeback_clients;
…...........................//debug, other information, etc.
};



class inst_t {
public:
   inst_t()
   {
      m_decoded=false;
      pc=(address_type)-1;
      reconvergence_pc=(address_type)-1;
      op=NO_OP;
      memset(out, 0, sizeof(unsigned));
      memset(in, 0, sizeof(unsigned));
      is_vectorin=0;
      is_vectorout=0;
      space = memory_space_t();
      cache_op = CACHE_UNDEFINED;
      latency = 1;
      initiation_interval = 1;
      for( unsigned i=0; i < MAX_REG_OPERANDS; i++ ) {
         arch_reg.src[i] = -1;
         arch_reg.dst[i] = -1;
      }
      isize=0;
   }
   bool valid() const { return m_decoded; }
   bool is_load() const { return (op == LOAD_OP || memory_op == memory_load); }
   bool is_store() const { return (op == STORE_OP || memory_op == memory_store); }
   address_type pc;        // program counter address of instruction
   unsigned isize;         // size of instruction in bytes
   op_type op;             // opcode (uarch visible)
   _memory_op_t memory_op; // memory_op used by ptxplus
   address_type reconvergence_pc; // -1 => not a branch, -2 => use function return address
   unsigned out[4];
   unsigned in[4];
   unsigned char is_vectorin;
   unsigned char is_vectorout;
   int pred; // predicate register number
   int ar1, ar2;
   // register number for bank conflict evaluation
   struct {
      int dst[MAX_REG_OPERANDS];
      int src[MAX_REG_OPERANDS];
   } arch_reg;
   //int arch_reg[MAX_REG_OPERANDS]; // register number for bank conflict evaluation
   unsigned latency; // operation latency
   unsigned initiation_interval;
   unsigned data_size; // what is the size of the word being operated on?
   memory_space_t space;
   cache_operator_type cache_op;
protected:
   bool m_decoded;
   virtual void pre_decode() {}
};



class warp_inst_t: public inst_t {
…...............................
void issue( const active_mask_t &mask, unsigned warp_id, unsigned long long cycle )
void completed( unsigned long long cycle ) const;
void generate_mem_accesses();
   void memory_coalescing_arch_13( bool is_write, mem_access_type access_type );
   void memory_coalescing_arch_13_atomic( bool is_write, mem_access_type access_type );
```

```
    void memory_coalescing_arch_13_reduce_and_send( bool is_write, mem_access_type
access_type, const transaction_info &info, new_addr_type addr, unsigned segment_size );
    void add_callback(unsigned lane_id,
                      void (*function)(const class inst_t*, class ptx_thread_info*),
                      const inst_t *inst,
                      class ptx_thread_info *thread )
…...............................
    void set_active( const active_mask_t &active );
    void clear_active( const active_mask_t &inactive );
    void set_not_active( unsigned lane_id );
…...............................
    bool active( unsigned thread ) const { return m_warp_active_mask.test(thread); }
    unsigned active_count() const { return m_warp_active_mask.count(); }
    unsigned issued_count() const { assert(m_empty == false); return m_warp_issued_mask.count(); }
 // for instruction counting
    bool empty() const { return m_empty; }
    unsigned warp_id() const
…..............................
     bool isatomic() const { return m_isatomic; }
    unsigned warp_size() const { return m_config->warp_size; }
…..............................
protected:
    unsigned m_uid;
    bool m_empty;
    bool m_cache_hit;
    unsigned long long issue_cycle;
    unsigned cycles; // used for implementing initiation interval delay
    bool m_isatomic;
    bool m_is_printf;
    unsigned m_warp_id;
    const core_config *m_config;
    active_mask_t m_warp_active_mask; // dynamic active mask for timing model (after predication)
    active_mask_t m_warp_issued_mask; // active mask at issue (prior to predication test) -- for
instruction counting
…...............................
        dram_callback_t callback;
        new_addr_type memreqaddr[MAX_ACCESSES_PER_INSN_PER_THREAD]; // effective
address, upto 8 different requests (to support 32B access in 8 chunks of 4B each)
    };
    bool m_per_scalar_thread_valid;
    std::vector<per_thread_info> m_per_scalar_thread;
    bool m_mem_accesses_created;
    std::list<mem_access_t> m_accessq;
    static unsigned sm_next_uid;
};



class mem_fetch {
public:
    mem_fetch( const mem_access_t &access,
               const warp_inst_t *inst,
               unsigned ctrl_size,
               unsigned wid,
               unsigned sid,
               unsigned tpc,
               const class memory_config *config );
   ~mem_fetch();
   void set_status( enum mem_fetch_status status, unsigned long long cycle );
   void set_reply()
   void do_atomic();
   const addrdec_t &get_tlx_addr() const { return m_raw_addr; }
   unsigned get_data_size() const { return m_data_size; }
   void     set_data_size( unsigned size ) { m_data_size=size; }
   unsigned get_ctrl_size() const { return m_ctrl_size; }
   unsigned size() const { return m_data_size+m_ctrl_size; }
   new_addr_type get_addr() const { return m_access.get_addr(); }
   new_addr_type get_partition_addr() const { return m_partition_addr; }
   bool     get_is_write() const { return m_access.is_write(); }
   unsigned get_request_uid() const { return m_request_uid; }
   unsigned get_sid() const { return m_sid; }
   unsigned get_tpc() const { return m_tpc; }
   unsigned get_wid() const { return m_wid; }
   bool istexture() const;
   bool isconst() const;
   enum mf_type get_type() const { return m_type; }
```

```
  bool isatomic() const;
  void set_return_timestamp( unsigned t ) { m_timestamp2=t; }
  void set_icnt_receive_time( unsigned t ) { m_icnt_receive_time=t; }
  unsigned get_timestamp() const { return m_timestamp; }
  unsigned get_return_timestamp() const { return m_timestamp2; }
  unsigned get_icnt_receive_time() const { return m_icnt_receive_time; }
  enum mem_access_type get_access_type() const { return m_access.get_type(); }
  const active_mask_t& get_access_warp_mask() const { return m_access.get_warp_mask(); }
  mem_access_byte_mask_t get_access_byte_mask() const { return m_access.get_byte_mask(); }
  address_type get_pc() const { return m_inst.empty()?-1:m_inst.pc; }
  const warp_inst_t &get_inst() { return m_inst; }
  enum mem_fetch_status get_status() const { return m_status; }

  const memory_config *get_mem_config(){return m_mem_config;}
private:
  // request source information
  unsigned m_request_uid;
  unsigned m_sid;
  unsigned m_tpc;
  unsigned m_wid;
  // where is this request now?
  enum mem_fetch_status m_status;
  unsigned long long m_status_change;
  // request type, address, size, mask
  mem_access_t m_access;
  unsigned m_data_size; // how much data is being written
  unsigned m_ctrl_size; // how big would all this meta data be in hardware (does not necessarily
match actual size of mem_fetch)
  new_addr_type m_partition_addr; // linear physical address *within* dram partition (partition bank
select bits squeezed out)
  addrdec_t m_raw_addr; // raw physical address (i.e., decoded DRAM chip-row-bank-column
address)
  enum mf_type m_type;
  // requesting instruction (put last so mem_fetch prints nicer in gdb)
  warp_inst_t m_inst;
…...............................
};
```

To summarize the above classes, *simt_core_cluster* models the SIMT core clusters. This class contains an array of SIMT core objects (the member *shader_core_ctx **m_core* in the *simt_core_cluster* class) modelled by the *shader_core_ctx* class. 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 (the member *std::list<mem_fetch*> m_response_fifo* in the *simt_core_cluster* class) from the interconnection network. It also pops the requests from the FIFO and sends them to the appropriate core's instruction cache (the member *read_only_cache *m_L1* in the *shader_core_ctx* class) or LDST unit (the member *ldst_unit *m_ldst_unit;* in the *shader_core_ctx* class).

The *shader_core_ctx* class models the SIMT core. Other than the *read_only_cache *m_L1* and *ldst_unit *m_ldst_unit;* members, the *shader_core_ctx* class also has other important components such as the registers for communication between two pipeline stages (e.g., the member *std::vector<register_set> m_pipeline_reg,* also other register sets in *schedulers*), the scoreboard (the *Scoreboard *m_scoreboard;* member), the scheduler (the member *std::vector<scheduler_unit*> schedulers;*) and the operand collector (the member *opndcoll_rfu_t m_operand_collector;*). The *shader_core_ctx* class also provides important methods for advancing the shader pipeline simulation *next_pc(), fetch(),decode(),issue(), read_operands(), execute()* and *writeback().*

The *core_t* class just provides an abstract base class for *shader_core_ctx* to inherit. *core_t* contains basic information such as the kernel information (the member *kernel_info_t *m_kernel*), ptx thread information (the member *class ptx_thread_info ** m_thread*), SIMT stack (the member *simt_stack **m_simt_stack*) and the current simulation instance (the member *gpgpu_sim *m_gpu*).

The *ldst_unit* class,a member in the *shader_core_ctx* class, contains various memory related components of a shader core including texture caches, constant caches, data caches.
Both the *simt_core_cluster* and the *ldst_unit* class (a shader core's load store unit) defines a member named *m_response_fifo,* which is actually a list of *mem_fetch* pointers. The *mem_fetch* class abstracts memory fetch operations with various relevant information such as the memory access type, source, destination, current status, fetched instruction (if it's an instruction access), etc. In particular, it has members *warp_inst_t m_inst* and *mem_access_t m_access.* The *mem_access_t* class just includes simple members related to a memory access such as the address, write or read, requested size, access type and active mask. The *warp_inst_t* class represents an instruction in a warp and comprises information such as the memory accesses incurred by the instruction (the member *std::list<mem_access_t> m_accessq;*), active mask(*active_mask_t m_warp_active_mask*), if atomic, cycle when it's issued, etc. In addition, The *warp_inst_t* class

inherits the class *inst_t* , which defines basic instruction information such as the opcode (the member *op_type op*), if load or store, register usage, latency (the member *unsigned latency;*), etc.

After introducing *simt_core_cluster, core_t, shader_core_ctx, ldst_unit, warp_inst_t, inst_t, mem_fetch* classes, now let's get back to the interconnect (or NoC, icnt for short) entry function of the shader core class (i.e., *icnt_cycle()*) to see how a shader core loads instruction and data access requests from the NoC.
This call (*icnt_cycle()*) is defined in gpgpu-sim\v3.x\src\gpgpu-sim\shader.cc and the full code is as follows:

```
void simt_core_cluster::icnt_cycle()
{
   if( !m_response_fifo.empty() ) {
      mem_fetch *mf = m_response_fifo.front();
      unsigned cid = m_config->sid_to_cid(mf->get_sid());
      if( mf->get_access_type() == INST_ACC_R ) {
         // instruction fetch response
         if( !m_core[cid]->fetch_unit_response_buffer_full() ) {
            m_response_fifo.pop_front();
            m_core[cid]->accept_fetch_response(mf);
         }
      } else {
         // data response
         if( !m_core[cid]->ldst_unit_response_buffer_full() ) {
            m_response_fifo.pop_front();
            m_memory_stats->memlatstat_read_done(mf);
            m_core[cid]->accept_ldst_unit_response(mf);
         }
      }
   }
   if( m_response_fifo.size() < m_config->n_simt_ejection_buffer_size ) {
      mem_fetch *mf = (mem_fetch*) ::icnt_pop(m_cluster_id);
      if (!mf)
         return;
      assert(mf->get_tpc() == m_cluster_id);
      assert(mf->get_type() == READ_REPLY || mf->get_type() == WRITE_ACK );
      mf->set_status(IN_CLUSTER_TO_SHADER_QUEUE,gpu_sim_cycle+gpu_tot_sim_cycle);
      //m_memory_stats->memlatstat_read_done(mf,m_shader_config->max_warps_per_shader);
      m_response_fifo.push_back(mf);
   }
}
```

From the above code we can see that if *simt_core_cluster.m_response_fifo* is not empty, *mem_fetch *mf* is retrieved from the front of the *m_response_fifo* and *mf*'s type is checked. If it's an instruction access execute *m_core[cid]->accept_fetch_response(mf);*, otherwise do *m_core[cid]->accept_ldst_unit_response(mf).*
The *accept_fetch_response()* and *accept_ldst_unit_response()* functions fill the shader's L1 instruction cache and ldst unit based on the *mem_fetch *mf* ,respectively:

```
void shader_core_ctx::accept_fetch_response( mem_fetch *mf )
{
   mf->set_status(IN_SHADER_FETCHED,gpu_sim_cycle+gpu_tot_sim_cycle);
   m_L1I->fill(mf,gpu_sim_cycle+gpu_tot_sim_cycle);
}
void shader_core_ctx::accept_ldst_unit_response(mem_fetch * mf)
{
  m_ldst_unit->fill(mf);
}
```

Part 4.2.2 Popping from Memory Controller to NoC (code in orange)

The second part of our analysis starts from the following code:

```
   if (clock_mask & ICNT) {
      // pop from memory controller to interconnect
      for (unsigned i=0;i<m_memory_config->m_n_mem;i++) {
         mem_fetch* mf = m_memory_partition_unit[i]->top();
         if (mf) {
            unsigned response_size = mf->get_is_write()?mf->get_ctrl_size():mf->size();
            if ( ::icnt_has_buffer( m_shader_config->mem2device(i), response_size ) ) {
               if (!mf->get_is_write())
                  mf->set_return_timestamp(gpu_sim_cycle+gpu_tot_sim_cycle);
               mf->set_status(IN_ICNT_TO_SHADER,gpu_sim_cycle+gpu_tot_sim_cycle);
               ::icnt_push( m_shader_config->mem2device(i), mf->get_tpc(), mf, response_size );
               m_memory_partition_unit[i]->pop();
            } else {
               gpu_stall_icnt2sh++;
            }
         } else {
            m_memory_partition_unit[i]->pop();
         }
      }
   }
```

The above code is almost self explainable with its comments except that the *m_memory_partition_unit* needs some extra study. *m_memory_partition_unit* is a member of the *gpgpu_sim* class:
*class memory_partition_unit **m_memory_partition_unit;*

The *memory_partition_unit* class is defined as follows:

```
class memory_partition_unit
{
public:
   memory_partition_unit( unsigned partition_id, const struct memory_config *config, class
memory_stats_t *stats );
   ~memory_partition_unit();
   bool busy() const;
   void cache_cycle( unsigned cycle );
   void dram_cycle();
   bool full() const;
   void push( class mem_fetch* mf, unsigned long long clock_cycle );
   class mem_fetch* pop();
   class mem_fetch* top();
   void set_done( mem_fetch *mf );
   unsigned flushL2();
 private:
// data
   unsigned m_id;
   const struct memory_config *m_config;
   class dram_t *m_dram;
   class l2_cache *m_L2cache;
   class L2interface *m_L2interface;
   partition_mf_allocator *m_mf_allocator;
   // model delay of ROP units with a fixed latency
   struct rop_delay_t
   {
           unsigned long long ready_cycle;
           class mem_fetch* req;
   };
   std::queue<rop_delay_t> m_rop;
   // model DRAM access scheduler latency (fixed latency between L2 and DRAM)
   struct dram_delay_t
   {
      unsigned long long ready_cycle;
      class mem_fetch* req;
   };
   std::queue<dram_delay_t> m_dram_latency_queue;
   // these are various FIFOs between units within a memory partition
   fifo_pipeline<mem_fetch> *m_icnt_L2_queue;
   fifo_pipeline<mem_fetch> *m_L2_dram_queue;
   fifo_pipeline<mem_fetch> *m_dram_L2_queue;
   fifo_pipeline<mem_fetch> *m_L2_icnt_queue; // L2 cache hit response queue
   class mem_fetch *L2dramout;
   unsigned long long int wb_addr;
   class memory_stats_t *m_stats;
   std::set<mem_fetch*> m_request_tracker;
   friend class L2interface;
};
```

To gain a high-level overview of the memory partition I refer to the document:

4.4.1.5 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.

4.4.1.5.1 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 ofmemory_partition_unit::cache_cycle().

Part 4.2.3 One Cycle in DRAM (code in yellow)

The relevant code in this part is:

```
  if (clock_mask & DRAM) {
      for (unsigned i=0;i<m_memory_config->m_n_mem;i++)
         m_memory_partition_unit[i]->dram_cycle(); // Issue the dram command (scheduler + delay
model)
   }
```

The above code simply invokes *m_memory_partition_unit[i]->dram_cycle()*, which has the following definition:

```
void memory_partition_unit::dram_cycle()
{
   // pop completed memory request from dram and push it to dram-to-L2 queue
   if ( !m_dram_L2_queue->full() ) {
      mem_fetch* mf = m_dram->pop();
      if (mf) {
         if( mf->get_access_type() == L1_WRBK_ACC ) {
            m_request_tracker.erase(mf);
            delete mf;
         } else {
            m_dram_L2_queue->push(mf);
            mf-
>set_status(IN_PARTITION_DRAM_TO_L2_QUEUE,gpu_sim_cycle+gpu_tot_sim_cycle);
         }
      }
   }
   m_dram->cycle();
   m_dram->dram_log(SAMPLELOG);
   if( !m_dram->full() && !m_L2_dram_queue->empty() ) {
      // L2->DRAM queue to DRAM latency queue
      mem_fetch *mf = m_L2_dram_queue->pop();
      dram_delay_t d;
      d.req = mf;
      d.ready_cycle = gpu_sim_cycle+gpu_tot_sim_cycle + m_config->dram_latency;
      m_dram_latency_queue.push(d);
      mf-
>set_status(IN_PARTITION_DRAM_LATENCY_QUEUE,gpu_sim_cycle+gpu_tot_sim_cycle);
   }
   // DRAM latency queue
   if( !m_dram_latency_queue.empty() && ( (gpu_sim_cycle+gpu_tot_sim_cycle) >=
m_dram_latency_queue.front().ready_cycle ) && !m_dram->full() ) {
       mem_fetch* mf = m_dram_latency_queue.front().req;
       m_dram_latency_queue.pop();
       m_dram->push(mf);
   }
}
```

In the above *memory_partition_unit::dram_cycle()*, other than moving *mf-* between two queues, *m_dram->cycle();* is executed to model the DRAM read/write operation:

```
void dram_t::cycle() {
  if( !returnq->full() ) {
      dram_req_t *cmd = rwq->pop();
      if( cmd ) {
          cmd->dqbytes += m_config->dram_atom_size;
          if (cmd->dqbytes >= cmd->nbytes) {
            mem_fetch *data = cmd->data;
            data->set_status(IN_PARTITION_MC_RETURNQ,gpu_sim_cycle+gpu_tot_sim_cycle);
           if( data->get_access_type() != L1_WRBK_ACC && data->get_access_type() !=
L2_WRBK_ACC ) {
               data->set_reply();
               returnq->push(data);
            } else {
               m_memory_partition_unit->set_done(data);
               delete data;
            }
            delete cmd;
          }
      }
  }
  /* check if the upcoming request is on an idle bank */
  /* Should we modify this so that multiple requests are checked? */
  switch (m_config->scheduler_type) {
  case DRAM_FIFO: scheduler_fifo(); break;
  case DRAM_FRFCFS: scheduler_frfcfs(); break;
      default:
            printf("Error: Unknown DRAM scheduler type\n");
            assert(0);
  }
  if ( m_config->scheduler_type == DRAM_FRFCFS ) {
     unsigned nreqs = m_frfcfs_scheduler->num_pending();
     if ( nreqs > max_mrqs) {
        max_mrqs = nreqs;
     }
     ave_mrqs += nreqs;
     ave_mrqs_partial += nreqs;
  } else {
     if (mrqq->get_length() > max_mrqs) {
        max_mrqs = mrqq->get_length();
     }
     ave_mrqs += mrqq->get_length();
     ave_mrqs_partial +=  mrqq->get_length();
  }
  unsigned k=m_config->nbk;
  bool issued = false;
  // check if any bank is ready to issue a new read
  for (unsigned i=0;i<m_config->nbk;i++) {
     unsigned j = (i + prio) % m_config->nbk;
        unsigned grp = j>>m_config->bk_tag_length;
     if (bk[j]->mrq) { //if currently servicing a memory request
        bk[j]->mrq->data->set_status(IN_PARTITION_DRAM,gpu_sim_cycle+gpu_tot_sim_cycle);
       // correct row activated for a READ
       if ( !issued && !CCDc && !bk[j]->RCDc &&
           !(bkgrp[grp]->CCDLc) &&
           (bk[j]->curr_row == bk[j]->mrq->row) &&
           (bk[j]->mrq->rw == READ) && (WTRc == 0 )  &&
           (bk[j]->state == BANK_ACTIVE) &&
           !rwq->full() ) {
         if (rw==WRITE) {
           rw=READ;
           rwq->set_min_length(m_config->CL);
         }
         rwq->push(bk[j]->mrq);
         bk[j]->mrq->txbytes += m_config->dram_atom_size;
```

```
            CCDc = m_config->tCCD;
            bkgrp[grp]->CCDLc = m_config->tCCDL;
            RTWc = m_config->tRTW;
            bk[j]->RTPc = m_config->BL/m_config->data_command_freq_ratio;
            bkgrp[grp]->RTPLc = m_config->tRTPL;
            issued = true;
            n_rd++;
            bwutil += m_config->BL/m_config->data_command_freq_ratio;
            bwutil_partial += m_config->BL/m_config->data_command_freq_ratio;
            bk[j]->n_access++;
#ifdef DRAM_VERIFY
            PRINT_CYCLE=1;
            printf("\tRD  Bk:%d Row:%03x Col:%03x \n",
                   j, bk[j]->curr_row,
                   bk[j]->mrq->col + bk[j]->mrq->txbytes - m_config->dram_atom_size);
#endif
            // transfer done
            if ( !(bk[j]->mrq->txbytes < bk[j]->mrq->nbytes) ) {
               bk[j]->mrq = NULL;
            }
         } else
            // correct row activated for a WRITE
            if ( !issued && !CCDc && !bk[j]->RCDWRc &&
                 !(bkgrp[grp]->CCDLc) &&
                 (bk[j]->curr_row == bk[j]->mrq->row)  &&
                 (bk[j]->mrq->rw == WRITE) && (RTWc == 0 )  &&
                 (bk[j]->state == BANK_ACTIVE) &&
                 !rwq->full() ) {
            if (rw==READ) {
               rw=WRITE;
               rwq->set_min_length(m_config->WL);
            }
            rwq->push(bk[j]->mrq);

            bk[j]->mrq->txbytes += m_config->dram_atom_size;
            CCDc = m_config->tCCD;
            bkgrp[grp]->CCDLc = m_config->tCCDL;
            WTRc = m_config->tWTR;
            bk[j]->WTPc = m_config->tWTP;
            issued = true;
            n_wr++;
            bwutil += m_config->BL/m_config->data_command_freq_ratio;
            bwutil_partial += m_config->BL/m_config->data_command_freq_ratio;
#ifdef DRAM_VERIFY
            PRINT_CYCLE=1;
            printf("\tWR  Bk:%d Row:%03x Col:%03x \n",
                   j, bk[j]->curr_row,
                   bk[j]->mrq->col + bk[j]->mrq->txbytes - m_config->dram_atom_size);
#endif
            // transfer done
            if ( !(bk[j]->mrq->txbytes < bk[j]->mrq->nbytes) ) {
               bk[j]->mrq = NULL;
            }
         }

         else
            // bank is idle
            if ( !issued && !RRDc &&
                 (bk[j]->state == BANK_IDLE) &&
                 !bk[j]->RPc && !bk[j]->RCc ) {
#ifdef DRAM_VERIFY
            PRINT_CYCLE=1;
            printf("\tACT BK:%d NewRow:%03x From:%03x \n",
                   j,bk[j]->mrq->row,bk[j]->curr_row);
#endif
            // activate the row with current memory request
            bk[j]->curr_row = bk[j]->mrq->row;
            bk[j]->state = BANK_ACTIVE;
            RRDc = m_config->tRRD;
            bk[j]->RCDc = m_config->tRCD;
            bk[j]->RCDWRc = m_config->tRCDWR;
            bk[j]->RASc = m_config->tRAS;
            bk[j]->RCc = m_config->tRC;
            prio = (j + 1) % m_config->nbk;
            issued = true;
            n_act_partial++;
            n_act++;
         }

         else
            // different row activated
            if ( (!issued) &&
                 (bk[j]->curr_row != bk[j]->mrq->row) &&
                 (bk[j]->state == BANK_ACTIVE) &&
                 (!bk[j]->RASc && !bk[j]->WTPc &&
                              !bk[j]->RTPc &&
                              !bkgrp[grp]->RTPLc) ) {
            // make the bank idle again
            bk[j]->state = BANK_IDLE;
            bk[j]->RPc = m_config->tRP;
            prio = (j + 1) % m_config->nbk;
            issued = true;
            n_pre++;
            n_pre_partial++;
#ifdef DRAM_VERIFY
            PRINT_CYCLE=1;
            printf("\tPRE BK:%d Row:%03x \n", j,bk[j]->curr_row);
#endif
         }
      } else {
         if (!CCDc && !RRDc && !RTWc && !WTRc && !bk[j]->RCDc && !bk[j]->RASc
             && !bk[j]->RCc && !bk[j]->RPc  && !bk[j]->RCDWRc) k--;
         bk[i]->n_idle++;
      }
   }
   if (!issued) {
      n_nop++;
      n_nop_partial++;
#ifdef DRAM_VIEWCMD
      printf("\tNOP                        ");
#endif
   }
   if (k) {
      n_activity++;
      n_activity_partial++;
   }
   n_cmd++;
   n_cmd_partial++;

   // decrements counters once for each time dram_issueCMD is called
   DEC2ZERO(RRDc);
   DEC2ZERO(CCDc);
   DEC2ZERO(RTWc);
   DEC2ZERO(WTRc);
   for (unsigned j=0;j<m_config->nbk;j++) {
      DEC2ZERO(bk[j]->RCDc);
      DEC2ZERO(bk[j]->RASc);
      DEC2ZERO(bk[j]->RCc);
      DEC2ZERO(bk[j]->RPc);
      DEC2ZERO(bk[j]->RCDWRc);
      DEC2ZERO(bk[j]->WTPc);
      DEC2ZERO(bk[j]->RTPc);
   }
   for (unsigned j=0; j<m_config->nbkgrp; j++) {
           DEC2ZERO(bkgrp[j]->CCDLc);
           DEC2ZERO(bkgrp[j]->RTPLc);
   }

#ifdef DRAM_VISUALIZE
   visualize();
#endif
}
```

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.

Part 4.2.4 Moving Memory Requests from NoC to Memory Partition & L2 Operation (code in green)

```
// L2 operations follow L2 clock domain
if (clock_mask & L2) {
   for (unsigned i=0;i<m_memory_config->m_n_mem;i++) {
      //move memory request from interconnect into memory partition (if not backed up)
      //Note:This needs to be called in DRAM clock domain if there is no L2 cache in the system
      if ( m_memory_partition_unit[i]->full() ) {
         gpu_stall_dramfull++;
      } else {
         mem_fetch* mf = (mem_fetch*) icnt_pop( m_shader_config->mem2device(i) );
         m_memory_partition_unit[i]->push( mf, gpu_sim_cycle + gpu_tot_sim_cycle );
      }
      m_memory_partition_unit[i]->cache_cycle(gpu_sim_cycle+gpu_tot_sim_cycle);
   }
}
```

Other than popping out memory requests from NoC into *m_memory_partition_unit[i]*, the most important statement in the above code is *m_memory_partition_unit[i]->cache_cycle*, which advances an L2 cycle:

```
void memory_partition_unit::cache_cycle( unsigned cycle )
{
   // L2 fill responses
   if( !m_config->m_L2_config.disabled()) {
      if ( m_L2cache->access_ready() && !m_L2_icnt_queue->full() ) {
         mem_fetch *mf = m_L2cache->next_access();
         if(mf->get_access_type() != L2_WR_ALLOC_R){ // Don't pass write allocate read request back
to upper level cache
                     mf->set_reply();
                     mf-
>set_status(IN_PARTITION_L2_TO_ICNT_QUEUE,gpu_sim_cycle+gpu_tot_sim_cycle);
                     m_L2_icnt_queue->push(mf);
         }else{
                     m_request_tracker.erase(mf);
                     delete mf;
         }
      }
   }
   // DRAM to L2 (texture) and icnt (not texture)
   if ( !m_dram_L2_queue->empty() ) {
      mem_fetch *mf = m_dram_L2_queue->top();
      if ( !m_config->m_L2_config.disabled() && m_L2cache->waiting_for_fill(mf) ) {
         mf->set_status(IN_PARTITION_L2_FILL_QUEUE,gpu_sim_cycle+gpu_tot_sim_cycle);
         m_L2cache->fill(mf,gpu_sim_cycle+gpu_tot_sim_cycle);
         m_dram_L2_queue->pop();
      } else if ( !m_L2_icnt_queue->full() ) {
         mf->set_status(IN_PARTITION_L2_TO_ICNT_QUEUE,gpu_sim_cycle+gpu_tot_sim_cycle);
         m_L2_icnt_queue->push(mf);
         m_dram_L2_queue->pop();
      }
   }
   // prior L2 misses inserted into m_L2_dram_queue here
   if( !m_config->m_L2_config.disabled() )
      m_L2cache->cycle();
   // new L2 texture accesses and/or non-texture accesses
   if ( !m_L2_dram_queue->full() && !m_icnt_L2_queue->empty() ) {
      mem_fetch *mf = m_icnt_L2_queue->top();
      if ( !m_config->m_L2_config.disabled() &&
           ( (m_config->m_L2_texure_only && mf->istexture()) || (!m_config->m_L2_texure_only) )
         ) {
```

```
         // L2 is enabled and access is for L2
         if ( !m_L2_icnt_queue->full() ) {
            std::list<cache_event> events;
            enum cache_request_status status = m_L2cache->access(mf-
>get_partition_addr(),mf,gpu_sim_cycle+gpu_tot_sim_cycle,events);
            bool write_sent = was_write_sent(events);
            bool read_sent = was_read_sent(events);
            if ( status == HIT ) {
               if( !write_sent ) {
                  // L2 cache replies
                  assert(!read_sent);
                  if( mf->get_access_type() == L1_WRBK_ACC ) {
                     m_request_tracker.erase(mf);
                     delete mf;
                  } else {
                     mf->set_reply();
                  mf-
>set_status(IN_PARTITION_L2_TO_ICNT_QUEUE,gpu_sim_cycle+gpu_tot_sim_cycle);
                     m_L2_icnt_queue->push(mf);
                  }
                  m_icnt_L2_queue->pop();
               } else {
                  assert(write_sent);
                  m_icnt_L2_queue->pop();
               }
            } else if ( status != RESERVATION_FAIL ) {
               // L2 cache accepted request
               m_icnt_L2_queue->pop();
            } else {
               assert(!write_sent);
               assert(!read_sent);
               // L2 cache lock-up: will try again next cycle
            }
         }
      } else {
         // L2 is disabled or non-texture access to texture-only L2
         mf->set_status(IN_PARTITION_L2_TO_DRAM_QUEUE,gpu_sim_cycle+gpu_tot_sim_cycle);
         m_L2_dram_queue->push(mf);
         m_icnt_L2_queue->pop();
      }
   }
   // ROP delay queue
   if( !m_rop.empty() && (cycle >= m_rop.front().ready_cycle) && !m_icnt_L2_queue->full() ) {
      mem_fetch* mf = m_rop.front().req;
      m_rop.pop();
      m_icnt_L2_queue->push(mf);
      mf->set_status(IN_PARTITION_ICNT_TO_L2_QUEUE,gpu_sim_cycle+gpu_tot_sim_cycle);
   }
}
```

Several paragraphs from the GPGPU-SIM document are put here to explain the above function (i.e.,*memory_partition_unit::cache_cycle*):

"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."

Part 4.2.5 One Cycle in NoC (code in light green)

```
  if (clock_mask & ICNT) {
     icnt_transfer();  //function pointer to advance_interconnect() defined in interconnect_interface.cc
  }
```

NoC simulation is done using a modified booksim simulator. Skip this part since it's not our focus.

Part 4.2.6 Shader Core Pipeline Stages + L1 Cache (code in blue)

```
 if (clock_mask & CORE) {
   // L1 cache + shader core pipeline stages
   for (unsigned i=0;i<m_shader_config->n_simt_clusters;i++) {
     if (m_cluster[i]->get_not_completed() || get_more_cta_left() ) {
         m_cluster[i]->core_cycle();
     }
   }
   if( g_single_step && ((gpu_sim_cycle+gpu_tot_sim_cycle) >= g_single_step) ) {
      asm("int $03");
   }
   gpu_sim_cycle++;
   if( g_interactive_debugger_enabled )
     gpgpu_debug();
```

This part of simulation simulates the shader core pipeline stages for instruction execution and thus is our main focus. At a high level, the above code just iterates over each shader core cluster and check if there is any running Cooperative Thread Array (CTA), or Thread Block left in the shader core cludyrt. If there is still some cta left,then invoke *m_cluster[i]->core_cycle();* to advance a shader core cluster cycle. Inside *m_cluster[i]->core_cycle();* each shader core is iterated and *cycle()* is called:

```
void simt_core_cluster::core_cycle()
{
   for( std::list<unsigned>::iterator it = m_core_sim_order.begin(); it != m_core_sim_order.end(); ++it )
{
       m_core[*it]->cycle();
   }
   if (m_config->simt_core_sim_order == 1) {
       m_core_sim_order.splice(m_core_sim_order.end(), m_core_sim_order,
m_core_sim_order.begin());
   }
}
```

To see how a cycle is simulated in a shader core, let's get into *m_core[*it]->cycle()*, which corresponds to the following code:

```
void shader_core_ctx::cycle()
{
   writeback();
   execute();
   read_operands();
   issue();
   decode();
   fetch();
}
```

Note that the functions for different pipeline stages are reversely called in *shader_core_ctx::cycle()* to mimic the pipeline operations in a shader core (this preserves the initial condition in the pipeline).
 We discuss these functions in the original order as they appear in a real machine.

Part 4.2.6.1 Fetch

As we have done in part 4.2.1, we first list the source code for several important structures necessary for understanding this part of simulation. The interface of these structures (i.e., *std::vector<shd_warp_t>   m_warp* , *ifetch_buffer_t   m_inst_fetch_buffer* and *read_only_cache *m_L1I; // instruction cache* members in the *shader_core_ctx* class, *baseline_cache* and *cache_t*) are as follows:

```
class shd_warp_t {
public:
   shd_warp_t( class shader_core_ctx *shader, unsigned warp_size)
   void reset();
   void init( address_type start_pc, unsigned cta_id, unsigned wid, const
std::bitset<MAX_WARP_SIZE> &active );
   bool functional_done() const;
   bool waiting(); // not const due to membar
   bool hardware_done() const;
   bool done_exit() const { return m_done_exit; }
   void set_done_exit() { m_done_exit=true; }
   unsigned get_n_completed() const { return n_completed; }
   void set_completed( unsigned lane )
```

```
    void set_last_fetch( unsigned long long sim_cycle ) { m_last_fetch=sim_cycle; }
    unsigned get_n_atomic() const { return m_n_atomic; }
    void inc_n_atomic() { m_n_atomic++; }
    void dec_n_atomic(unsigned n) { m_n_atomic-=n; }
    void set_membar() { m_membar=true; }
    void clear_membar() { m_membar=false; }
    bool get_membar() const { return m_membar; }
    address_type get_pc() const { return m_next_pc; }
    void set_next_pc( address_type pc ) { m_next_pc = pc; }
    void ibuffer_fill( unsigned slot, const warp_inst_t *pI )
    {
       assert(slot < IBUFFER_SIZE );
       m_ibuffer[slot].m_inst=pI;
       m_ibuffer[slot].m_valid=true;
       m_next=0;
    }
    bool ibuffer_empty();
    void ibuffer_flush();
    const warp_inst_t *ibuffer_next_inst() { return m_ibuffer[m_next].m_inst; }
    bool ibuffer_next_valid() { return m_ibuffer[m_next].m_valid; }
    void ibuffer_free()
    {
        m_ibuffer[m_next].m_inst = NULL;
        m_ibuffer[m_next].m_valid = false;
    }
    void ibuffer_step() { m_next = (m_next+1)%IBUFFER_SIZE; }

    bool imiss_pending() const { return m_imiss_pending; }
    void set_imiss_pending() { m_imiss_pending=true; }
    void clear_imiss_pending() { m_imiss_pending=false; }
    bool stores_done() const { return m_stores_outstanding == 0; }
    void inc_store_req() { m_stores_outstanding++; }
    void dec_store_req() ;
    bool inst_in_pipeline() const { return m_inst_in_pipeline > 0; }
    void inc_inst_in_pipeline() { m_inst_in_pipeline++; }
    void dec_inst_in_pipeline() ;
    unsigned get_cta_id() const { return m_cta_id; }

private:
    static const unsigned IBUFFER_SIZE=2;
    class shader_core_ctx *m_shader;
    unsigned m_cta_id;
    unsigned m_warp_id;
    unsigned m_warp_size;
    address_type m_next_pc;
    unsigned n_completed;          // number of threads in warp completed
    std::bitset<MAX_WARP_SIZE> m_active_threads;
    bool m_imiss_pending;
    struct ibuffer_entry {
       ibuffer_entry() { m_valid = false; m_inst = NULL; }
       const warp_inst_t *m_inst;
       bool m_valid;
    };
    ibuffer_entry m_ibuffer[IBUFFER_SIZE];
    unsigned m_next;
    unsigned m_n_atomic;           // number of outstanding atomic operations
    bool     m_membar;             // if true, warp is waiting at memory barrier
    bool m_done_exit; // true once thread exit has been registered for threads in this warp
    unsigned long long m_last_fetch;
    unsigned m_stores_outstanding; // number of store requests sent but not yet acknowledged
    unsigned m_inst_in_pipeline;
};



struct ifetch_buffer_t {
    bool m_valid;
    address_type m_pc;
    unsigned m_nbytes;
    unsigned m_warp_id;
};



class cache_t {
public:
    virtual ~cache_t() {}
    virtual enum cache_request_status access( new_addr_type addr, mem_fetch *mf, unsigned time,
std::list<cache_event> &events ) =  0;
};



class baseline_cache : public cache_t {
public:
    baseline_cache( const char *name, const cache_config &config, int core_id, int type_id,
mem_fetch_interface *memport, enum mem_fetch_status status )
    : m_config(config), m_tag_array(config,core_id,type_id),
m_mshrs(config.m_mshr_entries,config.m_mshr_max_merge)
    virtual enum cache_request_status access( new_addr_type addr, mem_fetch *mf, unsigned time,
std::list<cache_event> &events ) =  0;
    /// Sends next request to lower level of memory
    void cycle();
    /// Interface for response from lower memory level (model bandwidth restictions in caller)
    void fill( mem_fetch *mf, unsigned time );
    /// Checks if mf is waiting to be filled by lower memory level
    bool waiting_for_fill( mem_fetch *mf );
    /// Are any (accepted) accesses that had to wait for memory now ready? (does not include
accesses that "HIT")
    bool access_ready() const {return m_mshrs.access_ready();}
    /// Pop next ready access (does not include accesses that "HIT")
    mem_fetch *next_access(){return m_mshrs.next_access();}
    // flash invalidate all entries in cache
    void flush(){m_tag_array.flush();}
    void print(FILE *fp, unsigned &accesses, unsigned &misses) const;
    void display_state( FILE *fp ) const;

    protected:
    std::string m_name;
    const cache_config &m_config;
    tag_array  m_tag_array;
    mshr_table m_mshrs;
    std::list<mem_fetch*> m_miss_queue;
    enum mem_fetch_status m_miss_queue_status;
    mem_fetch_interface *m_memport;

    struct extra_mf_fields {
        extra_mf_fields()  { m_valid = false;}
        extra_mf_fields( new_addr_type a, unsigned i, unsigned d )
        {
            m_valid = true;
            m_block_addr = a;
            m_cache_index = i;
            m_data_size = d;
        }
        bool m_valid;
        new_addr_type m_block_addr;
        unsigned m_cache_index;
        unsigned m_data_size;
    };

    typedef std::map<mem_fetch*,extra_mf_fields> extra_mf_fields_lookup;
    extra_mf_fields_lookup m_extra_mf_fields;
    /// Checks whether this request can be handled on this cycle. num_miss equals max # of misses to
be handled on this cycle
    bool miss_queue_full(unsigned num_miss){  }
    /// Read miss handler without writeback
    void send_read_request(new_addr_type addr, new_addr_type block_addr, unsigned cache_index,
mem_fetch *mf,
                unsigned time, bool &do_miss, std::list<cache_event> &events, bool read_only, bool
wa);
    /// Read miss handler. Check MSHR hit or MSHR available
    void send_read_request(new_addr_type addr, new_addr_type block_addr, unsigned cache_index,
mem_fetch *mf,
                unsigned time, bool &do_miss, bool &wb, cache_block_t &evicted,
std::list<cache_event> &events, bool read_only, bool wa);
};



class read_only_cache : public baseline_cache {
public:
    read_only_cache( const char *name, const cache_config &config, int core_id, int type_id,
mem_fetch_interface *memport, enum mem_fetch_status status )
    : baseline_cache(name,config,core_id,type_id,memport,status){}

    /// Access cache for read_only_cache: returns RESERVATION_FAIL if request could not be
accepted (for any reason)
    virtual enum cache_request_status access( new_addr_type addr, mem_fetch *mf, unsigned time,
std::list<cache_event> &events );
};
```

Now let's look at how instruction fetch stage works in a shader pipeline. The source code for *shader_core_ctx::fetch()* is as follows:

```
void shader_core_ctx::fetch()
{
    if( !m_inst_fetch_buffer.m_valid ) {
        // find an active warp with space in instruction buffer that is not already waiting on a cache miss
        // and get next 1-2 instructions from i-cache...
        for( unsigned i=0; i < m_config->max_warps_per_shader; i++ ) {
            unsigned warp_id = (m_last_warp_fetched+1+i) % m_config->max_warps_per_shader;
            // this code checks if this warp has finished executing and can be reclaimed
            if( m_warp[warp_id].hardware_done() && !m_scoreboard->pendingWrites(warp_id) &&
!m_warp[warp_id].done_exit() ) {
                bool did_exit=false;
                for(unsigned t=0; t<m_config->warp_size;t++) {
                    unsigned tid=warp_id*m_config->warp_size+t;
                    if( m_threadState[tid].m_active == true ) {
                        m_threadState[tid].m_active = false;
                        unsigned cta_id = m_warp[warp_id].get_cta_id();
                        register_cta_thread_exit(cta_id);
                        m_not_completed -= 1;
                        m_active_threads.reset(tid);
                        assert( m_thread[tid]!= NULL );
                        did_exit=true;
                    }
                }
                if( did_exit )
                    m_warp[warp_id].set_done_exit();
            }
            // this code fetches instructions from the i-cache or generates memory requests
            if( !m_warp[warp_id].functional_done() && !m_warp[warp_id].imiss_pending() &&
m_warp[warp_id].ibuffer_empty() ) {
                address_type pc  = m_warp[warp_id].get_pc();
                address_type ppc = pc + PROGRAM_MEM_START;
                unsigned nbytes=16;
                unsigned offset_in_block = pc & (m_config->m_L1I_config.get_line_sz()-1);
                if( (offset_in_block+nbytes) > m_config->m_L1I_config.get_line_sz() )
                    nbytes = (m_config->m_L1I_config.get_line_sz()-offset_in_block);
                mem_access_t acc(INST_ACC_R,ppc,nbytes,false);
                mem_fetch *mf = new mem_fetch(acc, NULL, READ_PACKET_SIZE, warp_id, m_sid,
                                              m_tpc, m_memory_config );
                std::list<cache_event> events;
                enum cache_request_status status = m_L1I->access( (new_addr_type)ppc, mf,
gpu_sim_cycle+gpu_tot_sim_cycle,events);
                if( status == MISS ) {
                    m_last_warp_fetched=warp_id;
                    m_warp[warp_id].set_imiss_pending();
                    m_warp[warp_id].set_last_fetch(gpu_sim_cycle);
                } else if( status == HIT ) {
                    m_last_warp_fetched=warp_id;
                    m_inst_fetch_buffer = ifetch_buffer_t(pc,nbytes,warp_id);
                    m_warp[warp_id].set_last_fetch(gpu_sim_cycle);
                    delete mf;
                } else {
                    m_last_warp_fetched=warp_id;
                    assert( status == RESERVATION_FAIL );
                    delete mf;
                }
                break;
            }
        }
    }
    m_L1I->cycle();
    if( m_L1I->access_ready() ) {
        mem_fetch *mf = m_L1I->next_access();
        m_warp[mf->get_wid()].clear_imiss_pending();
        delete mf;
    }
}
```

At the first glance I was confused with the role of *m_inst_fetch_buffer* and I found the following paragraph in the official document regarding the fetch operation helpful in understanding the fetch operation:
"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."

From the above code and the official document we know that the variable m_inst_fetch_buffer services as a register between the fetch and decode stage. The first statement *if(!m_inst_fetch_buffer.m_valid)* in *shader_core_ctx::fetch()* just checks if *m_inst_fetch_buffer* is free of valid instructions (if the decode stage does not stall, *m_inst_fetch_buffer* will be freed by setting *m_valid* to false: *m_inst_fetch_buffer.m_valid = false;*). If *m_inst_fetch_buffer* is free of valid instructions (*m_inst_fetch_buffer.m_valid == false*), then enters an outer for loop to iterate through every wrap running on the shader (*warp_id = (m_last_warp_fetched+1+i) % m_config->max_warps_per_shader;*). Inside the outer for loop the current warp being iterated is first checked to see if it has finished executing and can be reclaimed. If the current warp is finished (*m_warp[warp_id].hardware_done() && !m_scoreboard->pendingWrites(warp_id) && !m_warp[warp_id].done_exit()*), enters an inner for loop to iterate every thread in the current warp and check for active threads (*m_threadState[tid].m_active == true*). The inner for loop does some cleanup work for each active thread and set the *did_exit* flag to be true if any thread is cleaned up. Setting flag to be true also triggers the execution of *m_warp[warp_id].set_done_exit();*after the inner loop. Next, still in the outer for loop, fetch instructions or generate memory requests for each warp if the condition *!m_warp[warp_id].functional_done() && !m_warp[warp_id].imiss_pending() && m_warp[warp_id].ibuffer_empty()* is satisfied. To fetch an instruction, first get relevant information such as the PC/PPC[2] address and the block offset of the instruction to be fetched. Then create a memory fetch object, with the relevant information for this instruction fetch, by executing the following two statements:

```
mem_access_t acc(INST_ACC_R,ppc,nbytes,false);
mem_fetch *mf = new mem_fetch(acc, NULL, READ_PACKET_SIZE, warp_id, m_sid,
                              m_tpc, m_memory_config );
```

After creating the memory fetch object *mf* then access the shader's instruction cache by invoking:

```
enum cache_request_status status = m_L1I->access( (new_addr_type)ppc, mf,
gpu_sim_cycle+gpu_tot_sim_cycle,events);
```

Based on the return status of the cache access, there are several situations. First, if there is a HIT, create a new *ifetch_buffer_t* object and copy it to *m_inst_fetch_buffer* (by executing *m_inst_fetch_buffer = ifetch_buffer_t(pc,nbytes,warp_id)*), which will be accessed by decode stage in the next cycle (remember that *m_inst_fetch_buffer* services as a register between fetch and decode stages and it stores. Essentially, *m_inst_fetch_buffer* associates an instruction in *m_warp[]* by keeping track of the *address_type m_pc* and *unsigned m_warp_id;* , which can be used by the decode stage to index *m_warp[]* for the corresponding instruction). After updating *m_inst_fetch_buffer* and recording the last fetched time (by *m_warp[warp_id].set_last_fetch(gpu_sim_cycle)*), the *mem_fetch *mf* is deleted. Second, if it is a *RESERVATION_FAIL*, simply remember *m_last_warp_fetched=warp_id* and then delete the *mem_fetch *mf*. The third case is a MISS, then do the following:

```
          m_last_warp_fetched=warp_id;
          m_warp[warp_id].set_imiss_pending();
          m_warp[warp_id].set_last_fetch(gpu_sim_cycle);
```

The outer for loop terminates here. Next *m_L1I->cycle();* is called:

```
/// Sends next request to lower level of memory
void baseline_cache::cycle(){
    if ( !m_miss_queue.empty() ) {
        mem_fetch *mf = m_miss_queue.front();
        if ( !m_memport->full(mf->get_data_size(),mf->get_is_write()) ) {
            m_miss_queue.pop_front();
            m_memport->push(mf);
        }
    }
}
```

The above code simply sends next memory request to lower level of memory by pushing the cache misses (*mem_fetch *mf = m_miss_queue.front();*) to memory port *m_memport->push(mf)*.

After ,fetch() checks if L1 has ready access (*if(m_L1I->access_ready())*) and if so then get next available access, clear the pending miss request and delete mf:
*mem_fetch *mf = m_L1I->next_access(); m_warp[mf->get_wid()].clear_imiss_pending(); delete mf;*

Part 4.2.6.2 Decode

The decode stage in a shader core is significantly related to the *m_inst_fetch_buffer* structure, which services as a conduit for communication between the fetch and decode stage. See the following official description for the decode stage:
“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.*”

```
void shader_core_ctx::decode()
{
   if( m_inst_fetch_buffer.m_valid ) {
      // decode 1 or 2 instructions and place them into ibuffer
      address_type pc = m_inst_fetch_buffer.m_pc;
      const warp_inst_t* pI1 = ptx_fetch_inst(pc);
      m_warp[m_inst_fetch_buffer.m_warp_id].ibuffer_fill(0,pI1);
      m_warp[m_inst_fetch_buffer.m_warp_id].inc_inst_in_pipeline();
      if( pI1 ) {
         const warp_inst_t* pI2 = ptx_fetch_inst(pc+pI1->isize);
         if( pI2 ) {
            m_warp[m_inst_fetch_buffer.m_warp_id].ibuffer_fill(1,pI2);
            m_warp[m_inst_fetch_buffer.m_warp_id].inc_inst_in_pipeline();
         }
      }
      m_inst_fetch_buffer.m_valid = false;
   }
}
```
.
Based on the information (pc, warp id, etc.) stored in *m_inst_fetch_buffer*, the decode stage first executes *const warp_inst_t* pI1 = ptx_fetch_inst(pc);*, which has the following source code:

```
const warp_inst_t *ptx_fetch_inst( address_type pc )
{
   return function_info::pc_to_instruction(pc);
}
```

One step further, *function_info::pc_to_instruction()* again has the following definition in ptx_ir.h

```
static const ptx_instruction* pc_to_instruction(unsigned pc)
  {
     if( pc < s_g_pc_to_insn.size() )
        return s_g_pc_to_insn[pc];
     else
        return NULL;
  }
```

Thus, it is important that we understand how *s_g_pc_to_insn* is created and maintained. After examining the usage of *s_g_pc_to_insn* throughout the GPGPU-SIM source code, it can be found that *s_g_pc_to_insn* is created and maintained by the function *function_info::ptx_assemble(),* which is invoked by *gpgpu_ptx_sim_load_ptx_from_string()*, as stated in the red text in part 2.1 (Part 2.1. Extracting PTX Using cuobjdump). The function *gpgpu_ptx_sim_load_ptx_from_string()* basically uses Lex/Yacc to parse the PTX code in a PTX file and create symbol table for that PTX file. The function *ptx_assemble()* is highly related to the PTX parsing, branch instruction detection and divergance analysis. For the details of these two functions please refer to Part 2.1. Extracting PTX Using cuobjdump and its subsections(e.g., Part 2.1.1 PTX Parsing: Start From *gpgpu_ptx_sim_load_ptx_from_string*).

After getting an instruction by *const warp_inst_t* pI1 = ptx_fetch_inst(pc)* the decode stage executes *m_warp[m_inst_fetch_buffer.m_warp_id].ibuffer_fill(0,pI1);*to fill the instruction(s) into the I-Buffer (the *ibuffer_entry m_ibuffer[IBUFFER_SIZE];*member in the *std::vector<shd_warp_t>* m_warp* member in the *shader_core_ctx* class).

Part 4.2.6.3 Issue

Again, we need some context and knowledge of relevant data structures to understand the simulation details in this part. In particular, we present classes *scheduler_unit* and *TwoLevelScheduler:*

```
class scheduler_unit { //this can be copied freely, so can be used in std containers.
public:
   scheduler_unit(...)...
   virtual ~scheduler_unit(){}
   virtual void add_supervised_warp_id(int i) {
      supervised_warps.push_back(i);
   }
```

```
    virtual void cycle()=0;
protected:
    shd_warp_t& warp(int i);
    std::vector<int> supervised_warps;
    int m_last_sup_id_issued;
    shader_core_stats *m_stats;
    shader_core_ctx* m_shader;
    // these things should become accessors: but would need a bigger rearchitect of how
shader_core_ctx interacts with its parts.
    Scoreboard* m_scoreboard;
    simt_stack** m_simt_stack;
    std::vector<shd_warp_t>* m_warp;
    register_set* m_sp_out;
    register_set* m_sfu_out;
    register_set* m_mem_out;
};



class TwoLevelScheduler : public scheduler_unit {
public:
        TwoLevelScheduler (shader_core_stats* stats, shader_core_ctx* shader,
            Scoreboard* scoreboard, simt_stack** simt,
            std::vector<shd_warp_t>* warp,
            register_set* sp_out,
            register_set* sfu_out,
            register_set* mem_out,
            unsigned maw)
        : scheduler_unit (stats, shader, scoreboard, simt, warp, sp_out, sfu_out, mem_out),
          activeWarps(),
          pendingWarps(){
                maxActiveWarps = maw;
        }
        virtual ~TwoLevelScheduler () {}
        virtual void cycle ();
        virtual void add_supervised_warp_id(int i) {
                pendingWarps.push_back(i);
        }
private:
        unsigned maxActiveWarps;
        std::list<int> activeWarps;
        std::list<int> pendingWarps;
};
```

Now let's analyze what the issue stage does. In the issue stage, the following code is executed:

```
void shader_core_ctx::issue(){
    for (unsigned i = 0; i < schedulers.size(); i++) {
        schedulers[i]->cycle();
    }
}
```

The *schedulers* in the above code is a member (*std::vector<scheduler_unit*> schedulers;)* in the *shader_core_ctx* class. As can be seen, the *cycle()* method is invoked upon *schedulers* . The *schedulers* object could be a subclass, for example the *TwoLevelScheduler*, of the base class *scheduler_unit*, in which the *cycle()* method is defined as a pure virtual function. Thus, let's get into the actual implementation of the *cycle()* method in the *TwoLevelScheduler* class (source code a little bit long):

```
void TwoLevelScheduler::cycle() {
        //Move waiting warps to pendingWarps
        for (std::list<int>::iterator iter = activeWarps.begin();iter != activeWarps.end();iter ++) {
                bool waiting = warp(*iter).waiting();
                for (int i=0; i<4; i++){
                        const warp_inst_t* inst = warp(*iter).ibuffer_next_inst();
                        //Is the instruction waiting on a long operation?
                        if ( inst && inst->in[i] > 0 && this->m_scoreboard->islongop(*iter, inst->in[i])){
                                waiting = true; }   }
                if(waiting){
                        pendingWarps.push_back(*iter);
                        activeWarps.erase(iter);
                        break;}
        }
        //If there is space in activeWarps, try to find ready warps in pendingWarps
        if (this->activeWarps.size() < maxActiveWarps){
            for (       std::list<int>::iterator iter = pendingWarps.begin();iter != pendingWarps.end();iter++)
{
                    if(!warp(*iter).waiting()){
                          activeWarps.push_back(*iter);
                          pendingWarps.erase(iter);
                          break;  }  }    }
        //Do the scheduling only from activeWarps
        //If you schedule an instruction, move it to the end of the list
        bool valid_inst = false;  // there was one warp with a valid instruction to issue (didn't require flush
due to control hazard)
        bool ready_inst = false;  // of the valid instructions, there was one not waiting for pending
register writes
        bool issued_inst = false; // of these we issued one
        for (std::list<int>::iterator warp_id = activeWarps.begin();warp_id !=
activeWarps.end();warp_id++) {
              unsigned checked=0;
              unsigned issued=0;
              unsigned max_issue = m_shader->m_config->gpgpu_max_insn_issue_per_warp;
              while(!warp(*warp_id).waiting() && !warp(*warp_id).ibuffer_empty() && (checked <
max_issue) && (checked <= issued) && (issued < max_issue) ) {
                    const warp_inst_t *pI = warp(*warp_id).ibuffer_next_inst();
                  bool valid = warp(*warp_id).ibuffer_next_valid();
                  bool warp_inst_issued = false;
                  unsigned pc,rpc;
                  m_simt_stack[*warp_id]->get_pdom_stack_top_info(&pc,&rpc);
                  if( pI ) {
                    assert(valid);
                    if( pc != pI->pc ) {
                      // control hazard
                      warp(*warp_id).set_next_pc(pc);
                       warp(*warp_id).ibuffer_flush();}
                              else {
                     valid_inst = true;
                     if ( !m_scoreboard->checkCollision(*warp_id, pI) ) {
                    ready_inst = true;
              const active_mask_t &active_mask = m_simt_stack[*warp_id]->get_active_mask();
                    assert( warp(*warp_id).inst_in_pipeline() );
                    if ( (pI->op == LOAD_OP) || (pI->op == STORE_OP) || (pI->op ==
MEMORY_BARRIER_OP) ) {
                    if( m_mem_out->has_free() ) {
                                                m_shader-
>issue_warp(*m_mem_out,pI,active_mask,*warp_id);
                                                issued++;
                                                issued_inst=true;
                                                warp_inst_issued = true;
                                                // Move it to pendingWarps
                                                unsigned currwarp = *warp_id;
                                                activeWarps.erase(warp_id);
                                                activeWarps.push_back(currwarp);
                                          }
                                    } else {
                                          bool sp_pipe_avail = m_sp_out->has_free();
                                          bool sfu_pipe_avail = m_sfu_out->has_free();
                                          if( sp_pipe_avail && (pI->op != SFU_OP) ) {
                                                // always prefer SP pipe for operations that can use both SP
and SFU pipelines
                                                m_shader-
>issue_warp(*m_sp_out,pI,active_mask,*warp_id);
                                                issued++;
                                                issued_inst=true;
                                                warp_inst_issued = true;
                                                //Move it to end of the activeWarps
                                                unsigned currwarp = *warp_id;
                                                activeWarps.erase(warp_id);
                                                activeWarps.push_back(currwarp);
                                          } else if ( (pI->op == SFU_OP) || (pI->op == ALU_SFU_OP) ) {
                                                if( sfu_pipe_avail ) {
                                                      m_shader-
>issue_warp(*m_sfu_out,pI,active_mask,*warp_id);
                                                      issued++;
                                                      issued_inst=true;
                                                      warp_inst_issued = true;
                                                      //Move it to end of the activeWarps
                                                      unsigned currwarp = *warp_id;
                                                      activeWarps.erase(warp_id);
                                                      activeWarps.push_back(currwarp);
```

```
                        }
                    }
                }
            }
        }
    } else if( valid ) {
        // this case can happen after a return instruction in diverged warp
        warp(*warp_id).set_next_pc(pc);
        warp(*warp_id).ibuffer_flush();
    }
    if(warp_inst_issued)
        warp(*warp_id).ibuffer_step();
    checked++;
}
if ( issued ) {
    break;
}
}
}
```

Note that the *std::vector<shd_warp_t>* warp* member in the *TwoLevelScheduler* class is actually passed with the value of *m_warp* in the *shader_core_ctx* class upon creation of a scheduler. Thus, the *ibuffer_entry m_ibuffer[IBUFFER_SIZE];* member in the *warp* member in *TwoLevelScheduler* is actually the instruction buffer (i.e.,I-Buffer) filled by the decode stage, as explained in the previous subsection (i.e.,Part 4.2.6.2 Decode).

From the source code and comments (it's a bit messy after being copied here, please refer to the original code structure) for the issue stage we know that it first moves waiting warps (warps that contain instructions that are waiting on a long operation) to *pendingWarps.*

```
if(waiting){
  pendingWarps.push_back(*iter);
  activeWarps.erase(iter);
}
```

Then check if there is space in *activeWarps,* try to find ready warps in *pendingWarps* and move them to *activeWarps.*

Next get into a big for loop to iterate the *activeWarps* for scheduling. This loop goes until the end of the entire *TwoLevelScheduler::cycle()* function. This big for loop first check condition while*(!warp(*warp_id).waiting() && !warp(*warp_id).ibuffer_empty() && (checked < max_issue) && (checked <= issued) && (issued < max_issue))* {.. If the condition is met then get into the while loop. Inside the while loop a warp instruction is first retrieved from the I-Buffer: *warp_inst_t *pI = warp(*warp_id).ibuffer_next_inst()*.
Then execute:

```
unsigned pc,rpc;
m_simt_stack[*warp_id]->get_pdom_stack_top_info(&pc,&rpc);
if( pI ) {
    assert(valid);
    if( pc != pI->pc ) {
        // control hazard
        warp(*warp_id).set_next_pc(pc);
        warp(*warp_id).ibuffer_flush();
    } else { ...........
```

The above few lines of code is explained by the following paragraph:
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.
If there is no control hazard, the flow goes to the else branch, where data hazard is checked by executing
*if(!m_scoreboard->checkCollision(*warp_id, pI)...*
If there is no data hazard, then the active mask is computed by *const active_mask_t &active_mask = m_simt_stack[*warp_id]->get_active_mask()*.
After getting the active mask, the flow splits into two branches to handle memory related instructions (*if ((pI->op == LOAD_OP) || (pI->op == STORE_OP) || (pI->op == MEMORY_BARRIER_OP))*) and non-memory related instructions (*else{...}*) . The memory related branch executes the following code

```
if( m_mem_out->has_free() ) {
      m_shader->issue_warp(*m_mem_out,pI,active_mask,*warp_id);
      issued++;
      issued_inst=true;
      warp_inst_issued = true;
      // Move it to pendingWarps
         unsigned currwarp = *warp_id;
      activeWarps.erase(warp_id);
      activeWarps.push_back(currwarp);
}
```

to issue the instruction by calling *m_shader->issue_warp()* with the *m_mem_out* argument.
On the other hand, the non-memory branch executes the following code

```
bool sp_pipe_avail = m_sp_out->has_free();
        bool sfu_pipe_avail = m_sfu_out->has_free();
        if( sp_pipe_avail && (pI->op != SFU_OP) ) {
            // always prefer SP pipe for operations that can use both SP and SFU pipelines
            m_shader->issue_warp(*m_sp_out,pI,active_mask,*warp_id);
            issued++;
            issued_inst=true;
            warp_inst_issued = true;
            //Move it to end of the activeWarps
            unsigned currwarp = *warp_id;
            activeWarps.erase(warp_id);
            activeWarps.push_back(currwarp);
            }
           else if ( (pI->op == SFU_OP) || (pI->op == ALU_SFU_OP) ) {
           if( sfu_pipe_avail ) {
                      m_shader->issue_warp(*m_sfu_out,pI,active_mask,*warp_id);
              issued++;
              issued_inst=true;
              warp_inst_issued = true;
              //Move it to end of the activeWarps
              unsigned currwarp = *warp_id;
                  activeWarps.erase(warp_id);
              activeWarps.push_back(currwarp);
              }
        }
```

to issue the instruction to the SP or SFU function unit based on the type/opcode of the issued instruction.
From the above code we can see that the issue stage calls *issue_warp()* with different arguments, namely *scheduler_unit::m_sp_out, scheduler_unit::m_sfu_out* and *scheduler_unit::m_mem_out*, for different types of instructions. The three objects *scheduler_unit::m_sp_out, scheduler_unit::m_sfu_out* and *scheduler_unit::m_mem_out* service as registers between the issue and execute stages of SP,SFU and Memory pipeline, respectively.

This paragraph summarizes the issue operation at a high level:
"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 pipeline receptively. That is why they are checked before issuing any instruction to its corresponding pipeline using *shader_core_ctx::issue_warp()*."

The source code of *issue_warp()* is as follows:

```
void shader_core_ctx::issue_warp( register_set& pipe_reg_set, const warp_inst_t* next_inst, const
active_mask_t &active_mask, unsigned warp_id )
{
    warp_inst_t** pipe_reg = pipe_reg_set.get_free();
    assert(pipe_reg);
    m_warp[warp_id].ibuffer_free();
    assert(next_inst->valid());
    **pipe_reg = *next_inst; // static instruction information
    (*pipe_reg)->issue( active_mask, warp_id, gpu_tot_sim_cycle + gpu_sim_cycle ); // dynamic
instruction information
    m_stats->shader_cycle_distro[2+(*pipe_reg)->active_count()]++;
    func_exec_inst( **pipe_reg );
    if( next_inst->op == BARRIER_OP )
        m_barriers.warp_reaches_barrier(m_warp[warp_id].get_cta_id(),warp_id);
    else if( next_inst->op == MEMORY_BARRIER_OP )
        m_warp[warp_id].set_membar();
    updateSIMTStack(warp_id,m_config->warp_size,*pipe_reg);
    m_scoreboard->reserveRegisters(*pipe_reg);
    m_warp[warp_id].set_next_pc(next_inst->pc + next_inst->isize);
}
```

Inside *shader_core_ctx::issue_warp*, the statement *warp_inst_t** pipe_reg = pipe_reg_set.get_free();* is first executed to get free slots in the register set. From the earlier code we know the *pipe_reg_set* could actually be *scheduler_unit::m_sp_out, scheduler_unit::m_sfu_out* or *scheduler_unit::m_mem_out* ,depending what type of instruction is going to be issued. These three register sets (i.e. *register_set* class) service as conduit for communication between the issue and the execute stages. The *register_set* class is simply a wrapper of a vector of instructions of a particular type. It has the following interface:

```
//register that can hold multiple instructions.
class register_set {
        warp_inst_t ** get_free(){
                for( unsigned i = 0; i < regs.size(); i++ ) {
                        if( regs[i]->empty() ) {
                                return &regs[i];
                        }
                }
…......................
std::vector<warp_inst_t*> regs;
```

After obtaining a free register slot in a right register set, *shader_core_ctx::issue_warp* executes ***pipe_reg = *next_inst;* to retrieve the next instruction to be issued for execution.

Next, *(*pipe_reg)->issue()* is executed to store the relevant information for the issued instruction into the register set, which will be later process by the execution stage. The *(*pipe_reg)->issue()* statement actually invokes the following function defined in the abstract_hardware_model.h:

```
    void issue( const active_mask_t &mask, unsigned warp_id, unsigned long long cycle )
    {
        m_warp_active_mask = mask;
        m_warp_issued_mask = mask;
        m_uid = ++sm_next_uid;
        m_warp_id = warp_id;
        issue_cycle = cycle;
        cycles = initiation_interval;
        m_cache_hit=false;
        m_empty=false;
    }
```

After putting the issued instruction into the correct register set, *shader_core_ctx::issue_warp()* further calls *shader_core_ctx::func_exec_inst(),* which is defined as follows:

```
void shader_core_ctx::func_exec_inst( warp_inst_t &inst )
{
    execute_warp_inst_t(inst, m_config->warp_size);
    if( inst.is_load() || inst.is_store() )
        inst.generate_mem_accesses();
}
```

An important function called in *func_exec_inst()* is *execute_warp_inst_t()*:

```
void core_t::execute_warp_inst_t(warp_inst_t &inst, unsigned warpSize, unsigned warpId)
{
    for ( unsigned t=0; t < warpSize; t++ ) {
        if( inst.active(t) ) {
            if(warpId==(unsigned (-1)))
                warpId = inst.warp_id();
            unsigned tid=warpSize*warpId+t;
            m_thread[tid]->ptx_exec_inst(inst,t);
            //virtual function
            checkExecutionStatusAndUpdate(inst,t,tid);
        }   }  }
```

As can be seen, in *execute_warp_inst_t()* a for loop iterates each thread (*m_thread[tid]*) and calls *ptx_exec_inst()* to execute the warp instruction. The function *ptx_exec_inst()* is pretty long. Regarding the function *ptx_exec_inst()* and the instruction execution the official document says the following:

“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().
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).”

Also, instruction execution is achieved by cooperating CUDA-Sim and GPGPU-Sim, as stated in the document:

“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_tclass 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_instfunction.
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 insidecore_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).”

Now let’s study *void ptx_thread_info::ptx_exec_inst(warp_inst_t &inst, unsigned lane_id)* step by step:

First,*const ptx_instruction *pI = m_func_info->get_instruction(pc);* is executed to get a *ptx_instruction* object *pI:*

```
  const ptx_instruction *get_instruction( unsigned PC ) const
  {
     unsigned index = PC - m_start_PC;
     if( index < m_instr_mem_size )
        return m_instr_mem[index];
     return NULL;
  }
```

ptx_instruction inherits the *warp_inst_t* class and has lots of data members as well as function members to for retrieving relevant information (e.g., operand, opcode, etc.) and properties (e.g.,is predicated, is atomic, has memory load/store, etc.) of the represented instruction. It is defined in ptx_ir.h file:

```
class ptx_instruction : public warp_inst_t {
unsigned inst_size() const { return m_inst_size; }
  unsigned uid() const { return m_uid;}
  int get_opcode() const { return m_opcode;}
  const char *get_opcode_cstr() const
  {
     if ( m_opcode != -1 ) {
        return g_opcode_string[m_opcode];
     } else {
        return "label";
     }
  }
  const char *source_file() const { return m_source_file.c_str();}
  unsigned source_line() const { return m_source_line;}
  unsigned get_num_operands() const { return m_operands.size();}
  bool has_pred() const { return m_pred != NULL;}
  operand_info get_pred() const { return operand_info( m_pred );}
  bool get_pred_neg() const { return m_neg_pred;}
  int get_pred_mod() const { return m_pred_mod;}
  const char *get_source() const { return m_source.c_str();}

  typedef std::vector<operand_info>::const_iterator const_iterator;
```

```
    const_iterator op_iter_begin() const
    {
      return m_operands.begin();
    }

    const_iterator op_iter_end() const
    {
      return m_operands.end();
    }

    const operand_info &dst() const
    {
      assert( !m_operands.empty() );
      return m_operands[0];
    }


    unsigned get_cmpop() const { return m_compare_op;}
    const symbol *get_label() const { return m_label;}
    bool is_label() const { if(m_label){ assert(m_opcode==-1);return true;} return false;}
    bool is_hi() const { return m_hi;}
    bool is_lo() const { return m_lo;}
    bool is_wide() const { return m_wide;}
    bool is_uni() const { return m_uni;}
    bool is_exit() const { return m_exit;}
    bool is_abs() const { return m_abs;}
    bool is_neg() const { return m_neg;}
    bool is_to() const { return m_to_option; }
    unsigned cache_option() const { return m_cache_option; }
    unsigned rounding_mode() const { return m_rounding_mode;}
    unsigned saturation_mode() const { return m_saturation_mode;}
    unsigned dimension() const { return m_geom_spec;}

  private:
    void set_opcode_and_latency();

    basic_block_t        *m_basic_block;
    unsigned          m_uid;
    addr_t            m_PC;
    std::string            m_source_file;
    unsigned                m_source_line;
    std::string          m_source;

    const symbol           *m_pred;
    bool                  m_neg_pred;
    int                  m_pred_mod;
    int                   m_opcode;
    const symbol           *m_label;
    std::vector<operand_info> m_operands;
    operand_info m_return_var;

    std::list<int>         m_options;
    bool              m_wide;
    bool              m_hi;
    bool              m_lo;
    bool              m_exit;
    bool              m_abs;
    bool              m_neg;
    bool              m_uni; //if branch instruction, this evaluates to true for uniform branches (ie jumps)
    bool              m_to_option;
    unsigned           m_cache_option;
    unsigned           m_rounding_mode;
    unsigned           m_compare_op;
    unsigned           m_saturation_mode;

    std::list<int>         m_scalar_type;
    memory_space_t m_space_spec;
    int m_geom_spec;
    int m_vector_spec;
    int m_atomic_spec;
    enum vote_mode_t m_vote_mode;
    int m_membar_level;
    int m_instr_mem_index; //index into m_instr_mem array
    unsigned m_inst_size; // bytes

    virtual void pre_decode();
    friend class function_info;
    static unsigned g_num_ptx_inst_uid;
  ……..
```

After obtaining the *ptx_instruction* object *pI* from *m_instr_mem[]* in the *m_func_info* object (a member in *ptx_thread_info* with the type of *function_info)*, check if *pI* is a predicated instruction and if so check if the predication condition is satisfied and update a flag *skip* accordingly to indicated whether this instruction should be executed or not:

```
if( pI->has_pred() ) {
    const operand_info &pred = pI->get_pred();
    ptx_reg_t pred_value = get_operand_value(pred, pred, PRED_TYPE, this, 0);
    if(pI->get_pred_mod() == -1) {
        skip = (pred_value.pred & 0x0001) ^ pI->get_pred_neg(); //ptxplus inverts the zero flag
    } else {
        skip = !pred_lookup(pI->get_pred_mod(), pred_value.pred & 0x000F);
    }
  }
```

About predication we can refer to the official document as well as the nvidia ptx isa document at http://docs.nvidia.com/cuda/pdf/ptx_isa_3.1.pdf (chapter 8.3):

“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;”
```

After the predication test then deactivate the thread lane if the *skip* flag is set otherwise execute the instruction:

```
  if( skip ) {
    inst.set_not_active(lane_id);
  } else {
    const ptx_instruction *pI_saved = pI;
    ptx_instruction *pJ = NULL;
    if( pI->get_opcode() == VOTE_OP ) {
      pJ = new ptx_instruction(*pI);
      *((warp_inst_t*)pJ) = inst; // copy active mask information
      pI = pJ;
    }
    switch ( pI->get_opcode() ) {
#define OP_DEF(OP,FUNC,STR,DST,CLASSIFICATION) case OP: FUNC(pI,this); op_classification
= CLASSIFICATION; break;
#include "opcodes.def"
#undef OP_DEF
    default: printf( "Execution error: Invalid opcode (0x%x)\n", pI->get_opcode() ); break;
    }
    delete pJ;
    pI = pI_saved;
    // Run exit instruction if exit option included
    if(pI->is_exit())
      exit_impl(pI,this);
  }
```

In the above code the actual instruction execution is emulated in the switch case block, in which a file named opcodes.def is included to specify the emulating function for each type of instruction. The opcodes.def file looks something like:

```
OP_DEF(ABS_OP,abs_impl,"abs",1,1)
OP_DEF(ADD_OP,add_impl,"add",1,1)
OP_DEF(ADDP_OP,addp_impl,"addp",1,1)
OP_DEF(ADDC_OP,addc_impl,"addc",1,1)
OP_DEF(AND_OP,and_impl,"and",1,1)
OP_DEF(ANDN_OP,andn_impl,"andn",1,1)
OP_DEF(ATOM_OP,atom_impl,"atom",1,3)
OP_DEF(BAR_OP,bar_sync_impl,"bar.sync",1,3)
OP_DEF(BFE_OP,bfe_impl,"bfe",1,1)
OP_DEF(BFI_OP,bfi_impl,"bfi",1,1)
OP_DEF(BFIND_OP,bfind_impl,"bfind",1,1)
OP_DEF(BRA_OP,bra_impl,"bra",0,3)
```

and the second argument of each *OP_DEF* macro services as a function pointer pointing to a function to process the corresponding instruction type. For example, for bar instruction the processing function is *bra_impl* defined in src/cuda-sim/instructions.cc:

```
void bra_impl( const ptx_instruction *pI, ptx_thread_info *thread )
{
  const operand_info &target  = pI->dst();
  ptx_reg_t target_pc = thread->get_operand_value(target, target, U32_TYPE, thread, 1);

  thread->m_branch_taken = true;
  thread->set_npc(target_pc);
}
```

Note that there are many instructions not implemented in the current GPGPU-Sim version.

Code afterwards in the *ptx_thread_info::ptx_exec_inst()* function are largely debugging and statistics collection code. However, there are still a few code segments inside *ptx_thread_info::ptx_exec_inst()* that need explanation:

```
if ( (pI->has_memory_read()  || pI->has_memory_write()) ) {
    insn_memaddr = last_eaddr();
    insn_space = last_space();
    unsigned to_type = pI->get_type();
    insn_data_size = datatype2size(to_type);
    insn_memory_op = pI->has_memory_read() ? memory_load : memory_store;
  }
```

The above code checks if the instruction contains memory read or write and if so set some variables for later use.

```
  if ( pI->get_opcode() == ATOM_OP ) {
    insn_memaddr = last_eaddr();
    insn_space = last_space();
    inst.add_callback( lane_id, last_callback().function, last_callback().instruction, this );
    unsigned to_type = pI->get_type();
    insn_data_size = datatype2size(to_type);
  }
```

The above code checks for atomic instruction and set callback function for handling atomic instructions.

```
  if (pI->get_opcode() == TEX_OP) {
    inst.set_addr(lane_id, last_eaddr() );
    assert( inst.space == last_space() );
    insn_data_size = get_tex_datasize(pI, this); // texture obtain its data granularity from the texture
info
  }
```

The above code checks texture operation.

```
  update_pc();
  g_ptx_sim_num_insn++;
```

After *func_exec_inst()* (calls *execute_warp_inst_t*,which further calls *ptx_exec_inst*) is finished in *issue_warp()* , the following code is simply executed to handle some special cases (*BARRIER_OP* and *MEMORY_BARRIER_OP* opcode). Finally, update the SIMT stack, reserve occupied registers by the current instruction in the scoreboard for later data hazard detection and set next pc address for the warp:

```
if( next_inst->op == BARRIER_OP )
      m_barriers.warp_reaches_barrier(m_warp[warp_id].get_cta_id(),warp_id);
   else if( next_inst->op == MEMORY_BARRIER_OP )
      m_warp[warp_id].set_membar();
   updateSIMTStack(warp_id,m_config->warp_size,*pipe_reg);
   m_scoreboard->reserveRegisters(*pipe_reg);
   m_warp[warp_id].set_next_pc(next_inst->pc + next_inst->isize);
```

Potential future working on explaining what happens when a barrier is hit and the synchronization mechanism used in GPGPU-Sim.
In particular, if a barrier operation is encountered, *barrier_set_t::warp_reaches_barrier()* is called to put the current warp being issued to the *m_barriers* member, which is a simple structure that implements a barrier:

```
class barrier_set_t {
public:
  barrier_set_t( unsigned max_warps_per_core, unsigned max_cta_per_core );
  // during cta allocation
  void allocate_barrier( unsigned cta_id, warp_set_t warps );
  // during cta deallocation
  void deallocate_barrier( unsigned cta_id );
  typedef std::map<unsigned, warp_set_t >  cta_to_warp_t;
  // individual warp hits barrier
  void warp_reaches_barrier( unsigned cta_id, unsigned warp_id );
  // fetching a warp
  bool available_for_fetch( unsigned warp_id ) const;
  // warp reaches exit
  void warp_exit( unsigned warp_id );
  // assertions
  bool warp_waiting_at_barrier( unsigned warp_id ) const;
  // debug
  void dump() const;
private:
  unsigned m_max_cta_per_core;
  unsigned m_max_warps_per_core;
  cta_to_warp_t m_cta_to_warps;
  warp_set_t m_warp_active;
  warp_set_t m_warp_at_barrier;
};
```

The source code of the function *barrier_set_t::warp_reaches_barrier()* is defined as follows:

```
void barrier_set_t::warp_reaches_barrier( unsigned cta_id, unsigned warp_id )
{
  cta_to_warp_t::iterator w=m_cta_to_warps.find(cta_id);
  if( w == m_cta_to_warps.end() ) { // cta is active
    printf("ERROR ** cta_id %u not found in barrier set on cycle %llu+%llu...\n", cta_id,
gpu_tot_sim_cycle, gpu_sim_cycle );
    dump();
    abort();
  }
  assert( w->second.test(warp_id) == true ); // warp is in cta
  m_warp_at_barrier.set(warp_id);
  warp_set_t warps_in_cta = w->second;
  warp_set_t at_barrier = warps_in_cta & m_warp_at_barrier;
  warp_set_t active = warps_in_cta & m_warp_active;
  if( at_barrier == active ) {
    // all warps have reached barrier, so release waiting warps...
    m_warp_at_barrier &= ~at_barrier;
  }
}
```

It should be pretty clear how a barrier works in GPGPU-Sim by reading the above code.
Author’s personal notes (working on branch divergence detection and synchronization elimination): This is important to research on dynamic warp formation, warp scheduling, etc.
Another data structure used a lot throughout this part is the *active_mask_t* defined in abstract_hardware_model.h: *typedef std::bitset<MAX_WARP_SIZE> active_mask_t;*

Part 4.2.6.4 Read Operand

```
void shader_core_ctx::read_operands()
{
}
```

It’s wired that the read_operands function is defined as an empty function in the shader.cc file.
Regarding the operand reading stage, just put the relevant section in the document here:
“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.”

Part 4.2.6.5 Execute

First we put the relevant source code here:

```
void shader_core_ctx::execute()
```

```
{
    for(unsigned i=0; i<num_result_bus; i++){
        *(m_result_bus[i]) >>=1;
    }
  for( unsigned n=0; n < m_num_function_units; n++ ) {
    unsigned multiplier = m_fu[n]->clock_multiplier();
    for( unsigned c=0; c < multiplier; c++ )
      m_fu[n]->cycle();
    enum pipeline_stage_name_t issue_port = m_issue_port[n];
    register_set& issue_inst = m_pipeline_reg[ issue_port ];
    warp_inst_t** ready_reg = issue_inst.get_ready();
    if( issue_inst.has_ready() && m_fu[n]->can_issue( **ready_reg ) ) {
      bool schedule_wb_now = !m_fu[n]->stallable();
      int resbus = -1;
      if( schedule_wb_now && (resbus=test_res_bus( (*ready_reg)->latency ))!=-1 ) {
        assert( (*ready_reg)->latency < MAX_ALU_LATENCY );
        m_result_bus[resbus]->set( (*ready_reg)->latency );
        m_fu[n]->issue( issue_inst );
      } else if( !schedule_wb_now ) {
        m_fu[n]->issue( issue_inst );
      } else {
        // stall issue (cannot reserve result bus)
      }
    }
  }
}
```

In *shader_core_ctx::execute()* the statement *m_fu[n]->cycle();* corresponds to the following code:

```
class pipelined_simd_unit : public simd_function_unit {
public:
  pipelined_simd_unit( register_set* result_port, const shader_core_config *config, unsigned
max_latency );
  //modifiers
  virtual void cycle()
  {
    if( !m_pipeline_reg[0]->empty() ){
      m_result_port->move_in(m_pipeline_reg[0]);
    }
    for( unsigned stage=0; (stage+1)<m_pipeline_depth; stage++ )
      move_warp(m_pipeline_reg[stage], m_pipeline_reg[stage+1]);
    if( !m_dispatch_reg->empty() ) {
      if( !m_dispatch_reg->dispatch_delay()) {
        int start_stage = m_dispatch_reg->latency - m_dispatch_reg->initiation_interval;
        move_warp(m_pipeline_reg[start_stage],m_dispatch_reg);
      }
    }
    occupied >>=1;
  }
```

In *shader_core_ctx::execute()* the statement *m_fu[n]->issue(issue_inst)* corresponds to the following code:

```
virtual void issue( register_set& source_reg ) { source_reg.move_out_to(m_dispatch_reg);
occupied.set(m_dispatch_reg->latency);}
```

To understand what *shader_core_ctx::execute()* does let’s combine a high level description about the execute stage in the GPGPU-Sim document with the corresponding source code lines:

“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():

```
………..int resbus = -1;
      if( schedule_wb_now && (resbus=test_res_bus( (*ready_reg)->latency ))!=-1 ) {
        assert( (*ready_reg)->latency < MAX_ALU_LATENCY );
        m_result_bus[resbus]->set( (*ready_reg)->latency );).
```

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

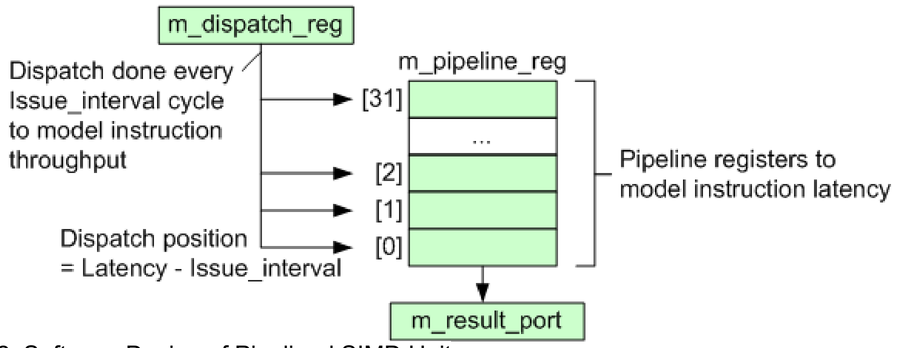

m_dispatch_reg
Dispatch done every Issue_interval cycle to model instruction throughput
m_pipeline_reg
[31]
...
[2]
[1]
[0]
Pipeline registers to model instruction latency
Dispatch position = Latency - Issue_interval
m_result_port

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 (by:
…............register_set& issue_inst = m_pipeline_reg[issue_port];
….... …....m_fu[n]->issue(issue_inst); :
virtual void issue(register_set& source_reg) { source_reg.move_out_to(m_dispatch_reg); occupied.set(m_dispatch_reg->latency);}). 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 (by:
if(!m_dispatch_reg->empty()) {
if(!m_dispatch_reg->dispatch_delay()) {
int start_stage = m_dispatch_reg->latency - m_dispatch_reg->initiation_interval;
move_warp(m_pipeline_reg[start_stage],m_dispatch_reg); in *m_fu[n]->cycle();*) 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 (by *for(unsigned stage=0; (stage+1)<m_pipeline_depth; stage++) move_warp(m_pipeline_reg[stage], m_pipeline_reg[stage+1]);* in *m_fu[n]->cycle();*) and eventually into m_result_port (by *if(!m_pipeline_reg[0]->empty()){m_result_port->move_in(m_pipeline_reg[0]);* in *m_fu[n]->cycle();*), 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."

After analyzing part 4.2.6.3 (issue stage) and part 4.2.6.5 (execute stage) we can conclude that in GPGPU-Sim the instruction execution is functionaly simulated in the issue stage, not the execution stage. The execution stage just begins by reading the instructions from the pipeline register *m_pipeline_reg[issue_port];*, fed by the issue stage. Note that in part 4.2.6.3 reads that the instructions in the issue stage are pushed into three register sets *scheduler_unit::m_sp_out, scheduler_unit::m_sfu_out,* and *scheduler_unit::m_mem_out* for processing. Actually, *scheduler_unit::m_sp_out , scheduler_unit::m_sfu_out* and *scheduler_unit::m_mem_out* are all pointers pointing to the *m_pipeline_reg register_set* vector (*std::vector<register_set> m_pipeline_reg*) member in the *shader_core_ctx* class. Thus, the issue stage issues/executes instructions and put them into *m_pipeline_reg,* which can be directly accessed by the execute stage.   This becomes clear after you look at the following code segments:

In the constructor of *shader_core_ctx* a *TwoLevelScheduler* scheduler is created this way

```
shader_core_ctx::shader_core_ctx(.........)
{
…..........................
schedulers.push_back(new TwoLevelScheduler(m_stats,this,m_scoreboard,m_simt_stack,&m_warp,
                                 &m_pipeline_reg[ID_OC_SP],
                                 &m_pipeline_reg[ID_OC_SFU],
                                 &m_pipeline_reg[ID_OC_MEM],
                                 tlmaw));
}
```

We can see that *&m_pipeline_reg[ID_OC_SP], &m_pipeline_reg[ID_OC_SFU]* and *&m_pipeline_reg[ID_OC_MEM],*are passed as the 6th, 7th and 8th arguments to create *TwoLevelScheduler:*

```
class TwoLevelScheduler : public scheduler_unit {
public:
    TwoLevelScheduler (shader_core_stats* stats, shader_core_ctx* shader,
      Scoreboard* scoreboard, simt_stack** simt,
      std::vector<shd_warp_t>* warp,
      register_set* sp_out,
      register_set* sfu_out,
      register_set* mem_out,
      unsigned maw)
    : scheduler_unit (stats, shader, scoreboard, simt, warp, sp_out, sfu_out, mem_out),
     activeWarps(),
     pendingWarps(){
        maxActiveWarps = maw;
    }
class scheduler_unit { //this can be copied freely, so can be used in std containers.
public:
   scheduler_unit(shader_core_stats* stats, shader_core_ctx* shader,
                  Scoreboard* scoreboard, simt_stack** simt,
                  std::vector<shd_warp_t>* warp,
                  register_set* sp_out,
                  register_set* sfu_out,
                  register_set* mem_out)
      : supervised_warps(), m_last_sup_id_issued(0), m_stats(stats), m_shader(shader),
      m_scoreboard(scoreboard), m_simt_stack(simt), /*m_pipeline_reg(pipe_regs),*/ m_warp(warp),
      m_sp_out(sp_out),m_sfu_out(sfu_out),m_mem_out(mem_out){}
```

while in the constructor of *scheduler_unit* , which *TwoLevelScheduler* inherits, the 6th, 7th and 8th parameters are passed to *m_sp_out(sp_out),m_sfu_out(sfu_out),m_mem_out(mem_out)* members.

Part 4.2.6.6 Writeback

The shader core writeback stage is relatively simple, compared with the previous stages:

```
void shader_core_ctx::writeback()
{
   warp_inst_t** preg = m_pipeline_reg[EX_WB].get_ready();
   warp_inst_t* pipe_reg = (preg==NULL)? NULL:*preg;
   while( preg and !pipe_reg->empty() ) {
        /*
         * Right now, the writeback stage drains all waiting instructions
         * assuming there are enough ports in the register file or the
         * conflicts are resolved at issue.
         */
        /*
         * The operand collector writeback can generally generate a stall
         * However, here, the pipelines should be un-stallable. This is
         * guaranteed because this is the first time the writeback function
         * is called after the operand collector's step function, which
         * resets the allocations. There is one case which could result in
         * the writeback function returning false (stall), which is when
         * an instruction tries to modify two registers (GPR and predicate)
         * To handle this case, we ignore the return value (thus allowing
         * no stalling).
         */
      m_operand_collector.writeback(*pipe_reg);
      unsigned warp_id = pipe_reg->warp_id();
      m_scoreboard->releaseRegisters( pipe_reg );
      m_warp[warp_id].dec_inst_in_pipeline();
      warp_inst_complete(*pipe_reg);
      m_gpu->gpu_sim_insn_last_update_sid = m_sid;
      m_gpu->gpu_sim_insn_last_update = gpu_sim_cycle;
      m_last_inst_gpu_sim_cycle = gpu_sim_cycle;
      m_last_inst_gpu_tot_sim_cycle = gpu_tot_sim_cycle;
      pipe_reg->clear();
      preg = m_pipeline_reg[EX_WB].get_ready();
      pipe_reg = (preg==NULL)? NULL:*preg;
   }
}
```

Basically, this stage releases registers in scoreboard, clears pipeline registers and updates warp information as well as simulation statistics.

Part 4.2.7 Thread Block / CTA / Work Group Scheduling (code in light blue)

```
      issue_block2core();
```

The function has the following source code:

```
unsigned simt_core_cluster::issue_block2core()
{
   unsigned num_blocks_issued=0;
   for( unsigned i=0; i < m_config->n_simt_cores_per_cluster; i++ ) {
      unsigned core = (i+m_cta_issue_next_core+1)%m_config->n_simt_cores_per_cluster;
      if( m_core[core]->get_not_completed() == 0 ) {
         if( m_core[core]->get_kernel() == NULL ) {
            kernel_info_t *k = m_gpu->select_kernel();
            if( k )
               m_core[core]->set_kernel(k);
         }
      }
```

```
        kernel_info_t *kernel = m_core[core]->get_kernel();
        if( kernel && !kernel->no_more_ctas_to_run() && (m_core[core]->get_n_active_cta() <
m_config->max_cta(*kernel)) ) {
            m_core[core]->issue_block2core(*kernel);
            num_blocks_issued++;
            m_cta_issue_next_core=core;
            break;
        }
    }
    return num_blocks_issued;
}
```

issue_block2core() iterates over each SIMT shader core in the SIMT core cluster (*simt_core_cluster*). For each core, check if the core is free (*if(m_core[core]->get_not_completed() == 0)*) and there is no kernel assigned to it (*if(m_core[core]->get_kernel() == NULL)*). If that is the case, then select a kernel to be simulated and assigns it to the core(by *kernel_info_t *k = m_gpu->select_kernel(); if(k) m_core[core]->set_kernel(k);*). Then get the *kernel_info_t* object *kernel* for the kernel running on or assigned to the current core and call *m_core[core]->issue_block2core(*kernel);*to issue the kernel to the current core. The statement *m_core[core]->issue_block2core(*kernel)* corresponds to the following code defined in gpu-sim.cc:

```
void shader_core_ctx::issue_block2core( kernel_info_t &kernel )
{
    set_max_cta(kernel);

    // find a free CTA context
    unsigned free_cta_hw_id=(unsigned)-1;
    for (unsigned i=0;i<kernel_max_cta_per_shader;i++ ) {
      if( m_cta_status[i]==0 ) {
         free_cta_hw_id=i;
         break;
      }
    }
    assert( free_cta_hw_id!=(unsigned)-1 );

    // determine hardware threads and warps that will be used for this CTA
    int cta_size = kernel.threads_per_cta();

    // hw warp id = hw thread id mod warp size, so we need to find a range
    // of hardware thread ids corresponding to an integral number of hardware
    // thread ids
    int padded_cta_size = cta_size;
    if (cta_size%m_config->warp_size)
      padded_cta_size = ((cta_size/m_config->warp_size)+1)*(m_config->warp_size);
    unsigned start_thread = free_cta_hw_id * padded_cta_size;
    unsigned end_thread  = start_thread +  cta_size;

    // reset the microarchitecture state of the selected hardware thread and warp contexts
    reinit(start_thread, end_thread,false);

    // initalize scalar threads and determine which hardware warps they are allocated to
    // bind functional simulation state of threads to hardware resources (simulation)
    warp_set_t warps;
    unsigned nthreads_in_block= 0;
    for (unsigned i = start_thread; i<end_thread; i++) {
        m_threadState[i].m_cta_id = free_cta_hw_id;
        unsigned warp_id = i/m_config->warp_size;
        nthreads_in_block += ptx_sim_init_thread(kernel,&m_thread[i],m_sid,i,cta_size-(i-
start_thread),m_config->n_thread_per_shader,this,free_cta_hw_id,warp_id,m_cluster->get_gpu());
        m_threadState[i].m_active = true;
        warps.set( warp_id );
    }
    assert( nthreads_in_block > 0 && nthreads_in_block <= m_config->n_thread_per_shader); //
should be at least one, but less than max
    m_cta_status[free_cta_hw_id]=nthreads_in_block;

    // now that we know which warps are used in this CTA, we can allocate
    // resources for use in CTA-wide barrier operations
    m_barriers.allocate_barrier(free_cta_hw_id,warps);

    // initialize the SIMT stacks and fetch hardware
    init_warps( free_cta_hw_id, start_thread, end_thread);
    m_n_active_cta++;

    shader_CTA_count_log(m_sid, 1);
    printf("GPGPU-Sim uArch: core:%3d, cta:%2u initialized @(%lld,%lld)\n", m_sid, free_cta_hw_id,
gpu_sim_cycle, gpu_tot_sim_cycle );
}
```

Part 4.2.8 Flushing Caches upon Completion (code in pink)

```
// Flush the caches once all of threads are completed.
if (m_config.gpgpu_flush_cache) {
  int all_threads_complete = 1 ;
  for (unsigned i=0;i<m_shader_config->n_simt_clusters;i++) {
    if (m_cluster[i]->get_not_completed() == 0)
      m_cluster[i]->cache_flush();
    else
      all_threads_complete = 0 ;
  }
  if (all_threads_complete && !m_memory_config->m_L2_config.disabled() ) {
    printf("Flushed L2 caches...\n");
    if (m_memory_config->m_L2_config.get_num_lines()) {
      int dlc = 0;
      for (unsigned i=0;i<m_memory_config->m_n_mem;i++) {
        dlc = m_memory_partition_unit[i]->flushL2();
        assert (dlc == 0); // need to model actual writes to DRAM here
        printf("Dirty lines flushed from L2 %d is %d\n", i, dlc  );
      }
    }
  }
}
```

This part of the simulation simply does L2 cache flushing to make sure the results are visible to the outside world.

Part 4.2.9 Printing Simulation Status (code in gray)

This part does not involve in the actual functional or performance simulation, it just prints out the simulation results.

---

[1] Note that the above trace is generated from GPGPU-SIM v3.1.1 while the version we are studying is v3.1.2. The #3 call in the stack, i.e., *useCuobjdump (),* does not exist in v3.1.2. Instead, the stack trace to *gpgpu_ptx_sim_load_ptx_from_string()* is *__cudaRegisterFunction()* -> *cuobjdumpParseBinary()->gpgpu_ptx_sim_load_ptx_from_string()* , as discussed in part 2.1.
[2] PPC address is the PC address plus an offset PROGRAM_MEM_START. Currently PROGRAM_MEM_START is defined to be 0xF0000000 in shader.cc. This should be distinct from other memory spaces.