공학박사학위논문

CPU/GPU 이종 병렬 플랫폼을 위한 GPU-in-the-loop 시뮬레이션 기법

GPU-in-the-loop Simulation for CPU/GPU Heterogeneous Platform

2016년 2월

서울대학교 대학원

전기컴퓨터 공학부

고 영 섭

Abstract

A mobile GPU has been widely adopted in most embedded systems to handle the complex graphics computations required in modern 3D games and highly interactive UI (User Interface). Moreover, as mobile GPUs are gaining more computation power and becoming increasingly programmable, they are also used to accelerate general-purpose computations in various fields such as physics and math, and so on. Unlike server GPUs, mobile GPUs usually have fewer cores since a limited amount of power is available in a battery. Thus, it is important to efficiently utilize both CPUs and GPUs in mobile platforms to satisfy the performance and power constraints.

For design space exploration of such a CPU-GPU heterogeneous architecture or debugging the SW in the early design stage, a full system simulator is typically used, in which simulation models of all HW components in the target system is included. Unfortunately, building a full system simulator with GPU simulator is not always possible because there is no available GPU simulator, or if any, it is prohibitively slow since they are mainly developed for architecture exploration varying the internal micro-architecture of GPUs.

To solve these problems, this thesis proposes a GPU-in-the-loop (GIL) simulation technique that integrates a real GPU with a full system simulator for CPU/GPU heterogeneous platforms.

In the first part of this thesis, we propose a system call-level simulation technique in which a full system simulator interacts with a GPU board at system call level.

Since the shared on-chip memory in the target system is modeled by two separate memories in the simulator and the board, memory synchronization is the most challenging problem in the proposed technique. To handle this problem in the system call-level technique, address translation tables are maintained for the shared memory regions and these memory regions are synchronized whenever the system calls which trigger the GPU execution are invoked in the board. To model the GPU execution in the simulator, interrupt-based modeling technique is proposed, in which the GPU interrupt is generated in consideration of the GPU execution time obtained from the real board.

In the second part of this thesis, we propose an API-level simulation technique in which a simulator and a board interact with each other at API level. Since the device driver in the original software stack makes it difficult to support various GPUs, a synthetic library is defined and it replaces the GPU library in the original software stack in order to ensure that the device driver is not used. To model timing of the API execution in the simulator, the sleep function is called in the synthetic driver so that the measured API time in the board elapses in the simulated time.

From the existing GPU APIs, we propose API-level simulation techniques for three commonly used APIs which are OpenCL, CUDA and OpenGL ES. And several challenging problems such as asynchronous behavior, multi-process support and memory synchronization for complex data structures are properly handled by several methods for correct simulation.

From the experimental results, we can confirm that the proposed technique can provide fast simulation speed with a reasonable timing accuracy. Therefore, it can be used not only for SW development but also for system level performance estimation. Moreover, the proposed technique makes the full system simulation for CPU/GPU heterogeneous platforms feasible even if a GPU simulator is not available.

Keywords : CPU/GPU heterogeneous platform, GPU Simulation, Virtual prototyping system, GPU-in-the-loop simulation, System call, API

Student Number : 2009-20750

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Motivation

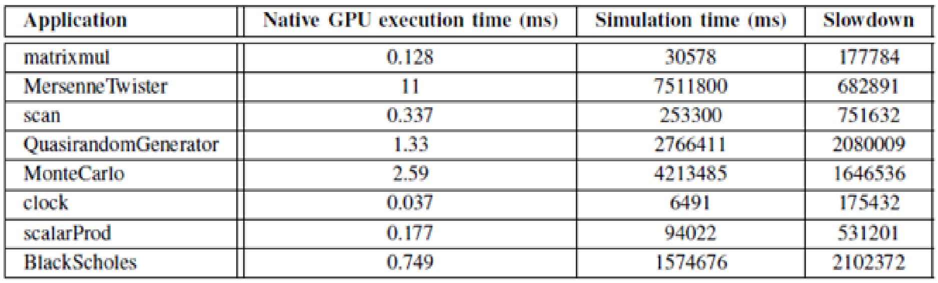
With ever increasing demand for computation in the embedded systems, a mobile GPU has become an essential component in most embedded systems. We can easily find many SoCs that integrate both a CPU and a GPU: Tegra from NVIDIA, Snapdragon from Qualcomm, and Exynos from Samsung, to name a few. These chips are widely used on many platforms ranging from automobiles to high-performance smart phones and tablet PCs. Since low power consumption is the major design constraint in most computer systems these days, the trend towards CPU/GPU heterogeneous platforms will continue, also with the increasing number of cores in CPUs and GPUs.

To design such a CPU/GPU heterogeneous platform efficiently, it is crucial to profile the target applications and utilize both a CPU and GPU better by identifying the performance bottleneck and capturing the dynamic system behaviors between

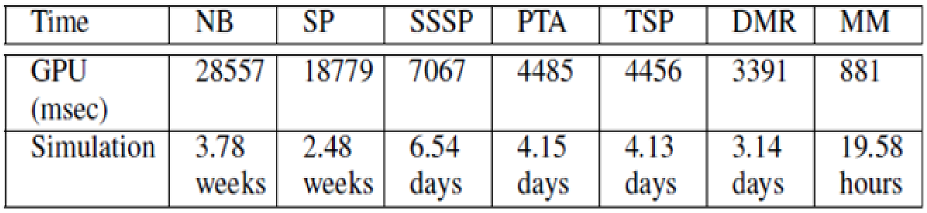
CPU and GPU. There exist many profiling tools [1][2][3], where a predefined set of hardware performance counters is collected and displayed to the designer to provide an overview of performance. However, this approach is only applicable to the exiting target platforms.

For the target platform under design, a virtual prototype is commonly used for performance estimation. Especially, full system simulation is performed in virtual prototypes since complete software stacks can run without modification by modeling all components of the target system including processors (CPUs, GPUs), memory, interconnections as well as peripherals. Generally, full system simulation is used for early software development or system-level DSE (Design Space Exploration) in early design stage. In these purposes, since SW implementation is modified frequently and lots of design candidates are verified, a large number of simulations are performed repetitively and fast simulation is really important. Moreover, as the complexity of the embedded system is increased greatly, much more HW components are integrated in a single system and the importance of the fast simulation is even more highlighted.

Upon the current move toward CPU/GPU heterogeneous platforms, many researches have been performed to simulate these platforms by integrating a CPU simulator with a GPU simulator. However, in this approach, there are some problems due to the existing GPU simulators. Since most of the existing GPU simulators [4][5][6][7][10][11] are mainly developed for architecture exploration varying the internal micro-architecture of GPUs, GPUs are modeled accurately in cycle-level, but the simulation speed is prohibitively slow. In previous researches [22][26], they present some experimental results for the simulation speed of some GPU simulators [6][11].



Application	Native GPU execution time (ms)	Simulation time (ms)	Slowdown
matrixmul	0.128	30578	177784
MersenneTwister	11	7511800	682891
scan	0.337	253300	751632
QuasirandomGenerator	1.33	2766411	2080009
MonteCarlo	2.59	4213485	1646536
clock	0.037	6491	175432
scalarProd	0.177	94022	531201
BlackScholes	0.749	1574676	2102372



Time	NB	SP	SSSP	PTA	TSP	DMR	MM
GPU (msec)	28557	18779	7067	4485	4456	3391	881
Simulation	3.78 weeks	2.48 weeks	6.54 days	4.15 days	4.13 days	3.14 days	19.58 hours

Figure 1-1. Simulation performance comparison results from [22] and [26]

From the results shown in Figure 1-1, the slowdown is around 170,000x ~ 2,000,000x for GPGPU-Sim [6] and 80,000x for MacSim [11] compared with native execution, which means that it takes more than a day to simulate a GPU for 1 second. However, this is not acceptable speed for early SW development or system-level DSE (Design Space Exploration) since a large number of simulations are repeated for these objectives. Moreover, for some mobile GPUs such as Mali and PowerVR, there is no publicly available simulator. Thus, it is impossible to build a full system simulator for the target platforms consisting of these GPUs.

To deal with these problems, we propose GPU-in-the-loop simulation technique that integrates a real GPU and a CPU simulator for fast simulation. The full system simulator and a GPU board can interact with each other at three different levels; API (Application Programming Interface), system call, and register/memory access. From them, two interactions at the system call and the API is covered in this thesis.

There are two major challenges in the proposed technique. First, since the on-chip shared memory in the target system is modeled with the two separate memories in the simulator and the board, we must synchronize the duplicated shared memory models to maintain the coherence. Second, since the detailed behavior of the GPU cannot be observed in the board, it is not easy to model the timing of the GPU in the proposed technique. To handle these problems, several methods for memory synchronization and timing modeling are proposed for each interfacing mechanism.

1.2 Contribution

The contribution of this thesis can be summarized as follows.

1) We propose a GPU-in-the-loop (GIL) simulation technique that integrates an existent GPU hardware with a full system simulator.

   A. Unlike previous works, since real GPU HW is used instead of slow

GPU simulators, the full system simulation becomes fast enough for early software development in the early stage with sacrificing some timing accuracy.

    B. Moreover, it make the full system simulation feasible for CPU/GPU heterogeneous platforms even if a GPU simulator is not available for the target platforms.

2) As well as the simulation speed is increased in the proposed technique, approximate timing of GPU can be modeled by novel modeling techniques

    A. The proposed technique can be used to estimate the performance for system level design space exploration such as task partitioning problem between CPUs and GPUs

3) The proposed interfacing mechanisms can also be applied in integrating a HW other than GPU with a full system simulator

1.3 Thesis Organization

This thesis is organized as follows. In Chapter 2, the representative previous researches on acceleration technique for GPU simulation and CPU/GPU simulation frameworks are reviewed. Chapter 3 explains the basic idea of the "GPU-in-the-loop" simulation technique and overall simulation flow will be briefly explained in this chapter. In Chapter 4 and Chapter 5, two simulation interface mechanisms, system call-level and API-level, are explained. Finally, we draw the conclusion and address future work in Chapter 6.

Chapter 2 Related Works

2.1 Acceleration techniques for GPU simulation

Since GPUs have become an important component in many platforms ranging from mobile devices to desktop PCs, the research interest in the GPU architecture is increasing and several GPU simulators are developed for the research purpose. For the architecture research, since micro-architecture of GPUs should be modeled accurately, current GPU simulators are really slow as mentioned in Chapter 1. To accelerate the slow GPU simulators, various techniques are proposed and they can be categorized into four approaches: parallel simulation, sampled simulation, statistical simulation and HW-accelerated simulation. In this chapter, we review the some exiting acceleration techniques for GPU simulations in these approaches.

2.1.1 Parallel Simulation

To accelerate the simulation for the many-core architectures, several parallel simulation frameworks have been proposed such as HSim [20] and Graphite [21], in which the simulation work for each processor is partitioned into multiple threads and performed in parallel on multi-core CPUs or multi-host machines. Since there are a large number of cores in a GPU, parallel simulation technique might be a viable solutions to accelerate the GPU simulation and several researches are proposed recently [22][23].

In [22], they proposed the work-group parallel simulation technique. Among the internal components in a GPU, the simulation for Computing Units (CU) are parallelized by multiple simulation threads; A CU corresponds to a Stream Multiprocessor (SM) in Nvidia GPU and a Data Parallel Processor (DPP) array in AMD GPU where several cores are executed in a SIMD manner. And, the other components such as a work control unit, interconnection networks and memory sub systems are simulated by two separate threads: Work distribution and control (WDC) and Interconnect-memory subsystem (IMS) threads. Since the lock-step synchronization method suffers from the synchronization overhead, they proposed the work-group based synchronization method in which synchronization is performed at the end of work-group execution on a CU to keep the same work-group distribution in the single-threaded simulation as much as possible. To improve the accuracy of the simulation for interconnection networks and memory sub systems, two additional synchronization mechanisms are applied to maintain the memory request order and model the contention in the interconnection network and the memory system more accurately.

Since the CU threads and the IMS thread are simulated independently in [22], the global memory request sequence can be different with that in the single-threaded simulation and this incurs the simulation error. To address this problem, error predictive synchronization (EPS) is proposed in [23] as an extended work for [22]. In this synchronization method, the instruction history is recorded within a specific cycle range to count the number of memory instructions executed in a CU. And, if the total memory instruction count for all CUs is larger than a given threshold, the parallel simulation is disabled and the simulation is performed sequentially until the total memory instruction count is below the threshold to reduce the memory latency error.

2.1.2 Sampled Simulation

Sampling is a well-known technique to speed up architecture simulation of long-running workloads by simulating only a small but representative portion of the application in detail while maintaining the accuracy. Several sampling techniques for single-threaded and multi-threaded CPU applications are proposed so far [24][25], they cannot be directly applied to GPU simulation, since it may lead to large sampling sizes and need to re-profile the target platform when the simulated configuration is changed.

In TBPoint [26], they proposed a new profile-based sampling technique for GPU simulation. For the hardware independency, it uses GPUOcelot [29] profile tool to collect the information about each thread block. Using the profiled information, it designs feature vectors for each kernel and thread blocks, then they are used for inter-launch sampling and intra-launch sampling technique to reduce simulation time. In inter-launch sampling, the kernels are clustered based on the kernel feature vectors and only the kernel selected as a simulation point is simulated by a detailed simulator such as MacSim [11] and other kernels in the same cluster re-use the IPC of the simulated kernel without the simulation. In intra-launch sampling, the thread blocks are clustered based on stall probability and region ID is assigned to each cluster. During the simulation, the homogeneous regions are identified when the region IDs for all concurrently running thread blocks are same. Then, the simulation is skipped just using the sampled IPC until the one of the region ID for concurrent thread blocks differs from others. Since the proposed technique leverages the regular execution behaviors in GPGPU kernels, GPGPU kernels with irregular execution pattern can incur high sampling size and the slow simulation is still a problem.

2.1.3 Statistical Simulation

In a statistical simulation, it measures a well-chosen set of program characteristics during GPGPU execution and generates a small synthetic benchmark with those characteristics. Then, the architecture simulation is performed using the small benchmarks. In [27], to keep the original characteristic, it first profiles the original GPGPU workloads through a fast functional simulator and the characteristics such as thread hierarchy, instruction mix, control flow and memory access pattern are collected. Then, loop patterns of the GPGPU kernel are analyzed based on the Divergence Flow Statistics Graph (DFSG) and the synthetic benchmark is generated by reducing the iteration counts of loops while maintaining the original characteristics. Since it requires some loops to increase the simulation speed, speed-up is limited for GPGPU kernels with small loop counts and a large number of thread blocks.

2.1.4 HW-accelerated Simulation

In software-based parallel simulation, due to the synchronization overhead, only coarse-grained parallelism can be exploited and the speed-up is limited. However, in a FPGA, since the cycle-level synchronization is much faster than SW implementation, fine-grained parallelism can be exploited. Thus, the simulation work performed sequentially in software-based parallel simulation can be done in parallel on the FPGA to further reduce the simulation speed.

In FastLanes [28], since modern GPUs are too complex to fit into even the largest single-chip FPGA, only a smaller number of multi-processors in the target GPU is implemented on the FPGA and they are re-used to simulate all multi-processors in a time-division multiplexing manner. Since only small number of threads are simulated on FPGA at a given moment, the contexts of the threads which are swapped out from the FPGA should be reserved in off-chip- memory. Since this incurs non-negligible performance overhead, the duration of a time slice is determined by an analytical performance model to balance the simulation speed and accuracy. The FPGA-based acceleration technique can provide fast simulation speed enough for full system simulation. However, it requires significant effort to develop since the simulator should written in a hardware description language (HDL) such as Verilog or VHDL.

2.2 CPU/GPU Simulation framework

To simulate the CPU/GPU heterogeneous platforms, a common practice is to integrate a GPU simulator with a CPU simulator. In [10][11][12], application-only simulators are implemented to simulate the GPGPU applications, in which only applications are simulated and OS services such as system calls are emulated by the simulation host. In FusionSim [12], several existing simulators (PTLSim [35], GPGPU-Sim [6] and MARSSx86 [36]) are integrated and two kinds of CPU/GPU systems (fused and discrete system) are modeled. In Multi2Sim [10], the functional simulator and the architecture simulator (timing model) are decoupled and the instruction traces obtained from the functional simulator are feed to the architecture simulator to accurately model the latencies of the instructions. In MacSim [11], only the trace-driven architecture simulator is implemented and the trace generators such as (Pin [34] and GPUOcelot [29]) are used to generate the CPU and GPU traces for the architecture simulator. And the OS is modeled in the process manager considering process and thread scheduling.

Since the significant errors can be introduced if the OS effect is not modeled [37], several researches are conducted based on the full system simulator such as gem5 [8], QEMU [9] and MARSSx86 [36]. In [13][14], they integrate the gem5 simulator with GPGPU-Sim by providing a common memory interface for both simulators. In SCHP [13], each process is created for gem5 and GPGPU-Sim respectively and the overall simulation is performed in lockstep. In order to ensure that both simulators are running in lock-step, the shared memory region is defined for inter-process-communication (IPC) and gem5 triggers the simulation of GPGPU-SIM by setting a flag in the shared memory and blocks until GPGPU-Sim completes the execution of a GPU cycle and the flags is reset. Also, since the memory system is modeled in the gem5, the memory requests from the GPGPU-SIM are stored in the shared memory to be handled in the gem5 simulator.

Unlike SCHP, gem5-gpu [14] combines two simulator as one process by integrating GPGPU-Sim's CU model into gem5.

Since the exiting CPU/GPU simulation frameworks only consider GPGPU applications written with CUDA or OpenCL, several full system simulation frameworks are developed to simulate the graphics applications [15][16][17][18]. In [15], it integrates the gem5 with ATTILA simulator [4] and can support Multi-CPU and Multi-GPU heterogeneous architecture.

In other CPU/GPU simulation frameworks [16][17][18], since fast simulation is really important and their concern is not in the CPU, QEMU full system simulator is used since it can achieve fast simulation based on dynamic binary translation. In [16], to verify the software and hardware architecture for multi-view GPU in the early design stage, QEMU simulator is integrated with multi-view GPU model implemented in SystemC or RTL codes. Since the unimportant HW components are simulated quickly with the QEMU simulator and only the important part is simulated in detail with SystemC model, the co-design for hardware and software can be performed efficiently. In [18], it develops cycle-accurate GPU simulators which can models two types of micro-architectures in modern GPUs such as Tile-Based Renderer (TBDR) and Immediate-Mode Rendering (IMR). Also, it provides a power model for a GPU using McPAT [19] to analyze energy consumption in GPUs. In [17], the full system simulator for many-core heterogeneous SoCs is developed using GPU and QEMU semi-hosting [46]. Though this work targets the many-core accelerator not a typical GPU, the semi-hosting interface mechanism proposed in this work can be used to integrate the QEMU simulator with a GPU simulator such as GPGPU-Sim.

2.3 Summary

Table 2-1 shows the comparison result of the acceleration techniques for GPU simulation. From the result, they shows remarkable speed-up results in comparison with original GPU simulators with some reasonable errors. However, there some limitations for each approach and these approaches are only applicable when GPU simulators are available. But, the proposed technique can perform the full system simulation even if GPU simulators are not available.

Table 2-2 shows the comparison result of existing CPU/GPU simulation frameworks. They are classified based on three criteria: GPU functionality, simulation scope, and simulation detail. Since the accurate but slow GPU simulators are used in exiting CPU/GPU simulation frameworks, they will suffers from slow simulation speed and they are not suitable for SW development or System-level DSE purposes.

Table 2-1. Comparison of the acceleration techniques for GPU simulation

Acceleration Technique	Baseline GPU Simulator	Speed-up	Functional Correctness	Timing Accuracy (%)	Limitations
Parallel Simulation [22]	GPGPU-Sim	Up to 4.15x on quad-core	Functionally Correct	0.05 ~ 26.61 % (Total Cycle)	The speed-up is limited by the # of CPU cores in the simulation host
Profile based Sampled Simulation [26]	Mac-Sim	2x ~ 50x	No Functionality	Up to 14.0 % (Total IPC)	The sample size can be up to 50% of total instructions for kernels with irregular execution patterns
Statistical Simulation [27]	GPGPU-Sim	1x ~ 589x	No Functionality	Up to 16 % (Total IPC)	Speed-up is limited for kernels with short loop counts
FPGA-based Simulation [28]	GPGPU-Sim	about 100x (2 orders magnitude)	Functionally Correct	No Information	Hard to implement the simulator
Proposed Technique	No simulator		Functionally Correct	Up to 20% (Total Execution Time)	Only model the CPU/GPU platforms with existing GPUs

Table 2-2. Comparison of CPU/GPU simulation frameworks

Existing Researches	GPU Func.	Full System/ App. Only	Functional Correctness	Timing Accuracy	
				CPU	GPU
MacSim [11]	GPGPU	Application Only	No Functionality	Cycle-Acc. (Own)	Cycle-Acc. (Own)
Multi2Sim [10]	GPGPU	Application Only	Functionally Correct	Cycle-Acc. (Own)	Cycle-Acc. (Own)
FusionSim [12]	GPGPU	Application Only	Functionally Correct	Cycle-Acc. (PTLSim)	Cycle-Acc. (GPGPU-sim)
SCHP [14] gem5-gpu [13]	GPGPU	Full System	Functionally Correct	Cycle-Acc. (gem5)	Cycle-Acc. (GPGPU-sim)
MCMG [15]	Graphics	Full System	Functionally Correct	Cycle-Acc. (gem5)	Cycle-Acc. (ATTILA)
QEMU +SystemC [16]	Graphics	Full System	Functionally Correct	No timing (QEMU)	Cycle-Acc. (SystemC)
TEAPOT [18]	Graphics	Full System	Functionally Correct	No timing (QEMU)	Cycle-Acc. (Own)
Proposed Technique	GPGPU Graphics	Full System	Functionally Correct	Cycle-Acc. (gem5)	Cycle-Approx. (Real HW)

Chapter 3 GPU-in-the-loop Simulation

3.1 Basic Idea

The basic idea for the GIL simulation technique is to integrate a real GPU with a CPU simulator in full system simulation framework instead of a GPU simulator. Since the proposed technique is not designed to be applicable for a specific simulator, any full system simulators can be used. In this thesis, as an example, the proposed technique is implemented based on gem5 simulator [8].

Figure 3-1 illustrates an overview of GPU-in-the-loop (GIL) simulation technique. In the simulation framework, a full system simulator is configured for the target CPU/GPU heterogeneous system using the gem5 simulator, which includes the simulation model of a multi-core CPU and other HW components except for the GPU, and actually runs the Android full software stack with Full system (FS) mode.

To connect the full system simulator with the GPU board, the host interface is added in the simulation host and it interacts with the CPU model to detect GPU requests and obtain the additional information for the requests (①). In the GPU board, the board interface is implemented in Android application in which GPU requests are received from the host interface through the network interface (②) and they are processed using real GPU in the board (③). After the requests are completed, the output results are sent to the host interface (④) and they are reflected to the simulator for both functional and timing correctness.

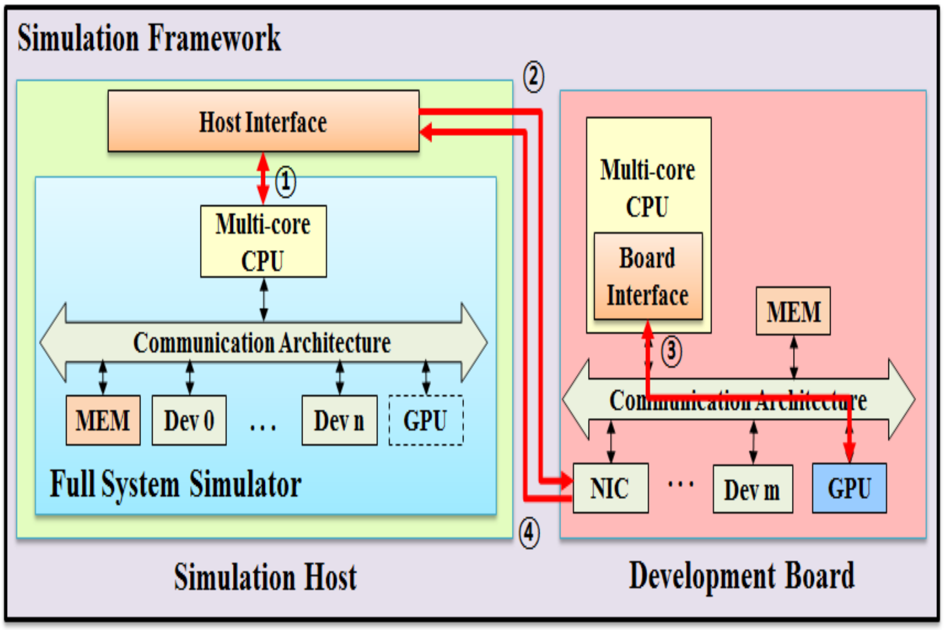


Figure 3-1. The overall GIL simulation framework

3.2 Different levels of CPU/GPU Interaction

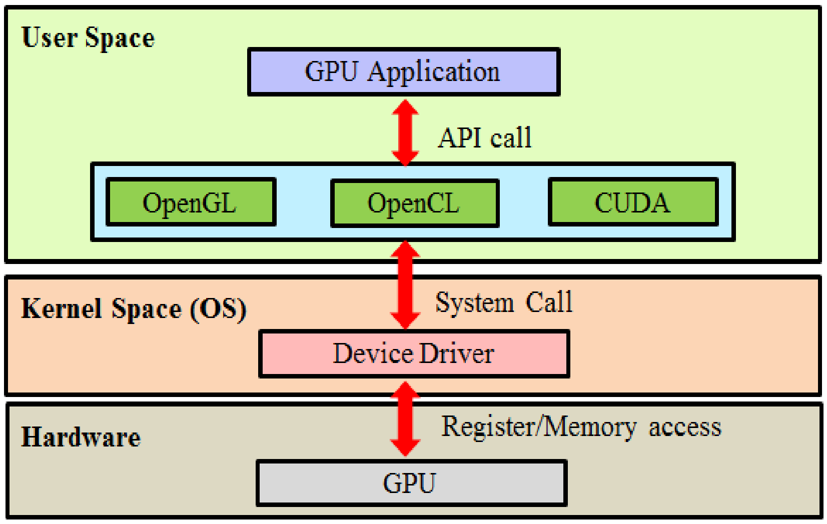


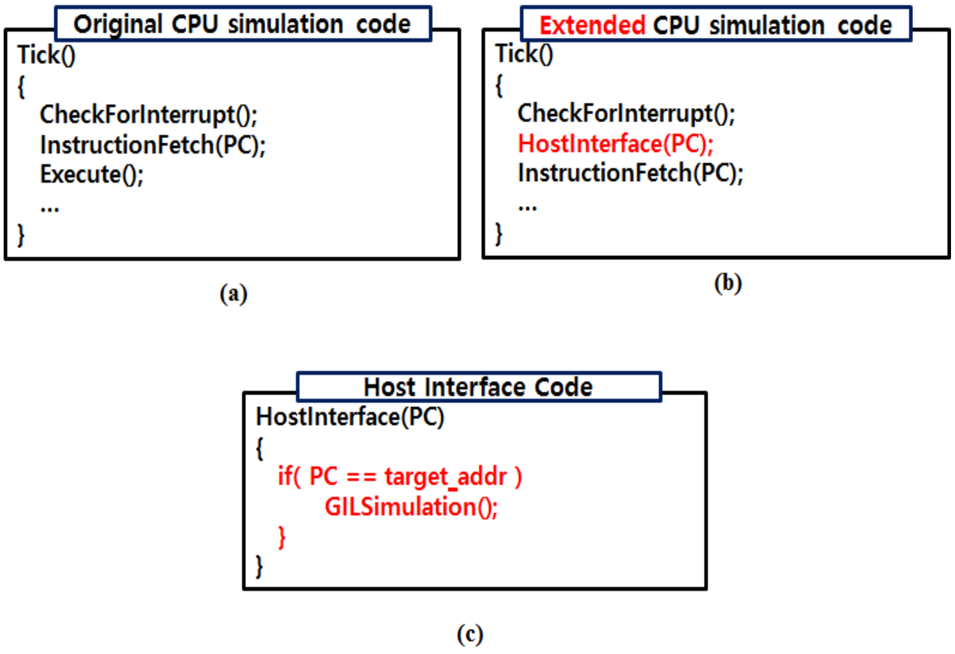
Figure 3-2. Typical GPU execution scenario

Figure 3-2 illustrates the typical GPU SW stacks in most CPU/GPU platforms in which three different levels of GPU requests are used to deliver some tasks on the GPU. In the GPU application, it invokes the API functions in the GPU libraries which are provided to enable the application developers to utilize the graphics (OpenGL ES) or GPGPU (OpenCL and CUDA) functionalities of the GPU. In the GPU libraries, each API request is translated into several low-level GPU commands and they are passed to the GPU device driver using system calls such as *ioctl* and *mmap*. In the GPU device driver, the shared memory and the GPU registers are accessed to directly pass the requested commands to the GPU.

Since the board interface is implemented as a user-level application in the proposed technique, from the three types of GPU requests, only the API and system call requests can be used in the board interface to pass the requests to the GPU. Thus, the API-level and system call-level GIL simulation techniques will be covered in this paper.

3.3 Detection Mechanism

The first process that happens in the proposed technique is to detect GPU requests from the CPU model. In the proposed technique, since a GPU request is defined as a function call (system call or API), it can be detected by comparing the current instruction address (PC) of the CPU model with the start address of the target function. For this, first we should obtain the address information for the target functions. Since the target functions are included in OS kernel, the target address can be obtained from the OS kernel image used in the simulation by disassembling the image using *objdump* utility. Using this address information, the detection mechanism is implemented in the host interface as shown in Figure 3-3 (c). And since the PC value is only available in CPU model, the original CPU model (Figure 3-3 (a)) is extended to pass the PC value to the host interface (Figure 3-3 (b)).



```
Tick()
{
   CheckForInterrupt();
   InstructionFetch(PC);
   Execute();
   ...
}
```

```
Tick()
{
   CheckForInterrupt();
   HostInterface(PC);
   InstructionFetch(PC);
   ...
}
```

```
HostInterface(PC)
{
   if( PC == target_addr )
        GILSimulation();
   }
}
```

Figure 3-3. Extension for CPU simulator and detection code in the host interface

Instead of comparing the address, it is possible to detect GPU requests by using special instructions such as pseudo instructions available in the simulator or SW interrupt instruction (svc) used in ARM semi-hosting [46]. In this approach, unlike the address-based detection mechanism mentioned above, the original source code of the GPU library or the device driver should be provided to insert the special instructions to the detection point in the code and the binaries are re-built from the modified source code. However, it is possible to detect the functions in both user space and kernel space in this approach. Whereas, only the functions in the kernel address can be detected in the address-based detection mechanism since the address of the user space is determined by the dynamic linker during the simulation. For this reason, the synthetic driver is added should be added in the API-level technique. However, if we apply the instruction-base detection mechanism, the synthetic driver is not required.

3.4 Memory Coherency Problem

In most mobile platforms, on-chip memory is shared by a CPU and a GPU. However, in the proposed technique, the shared memory is modeled by two separate memories in the simulator and the board, and the CPU model and the real GPU accesses the different memory in each side. Thus, the modification in one memory is not reflected to the other memory and in-correct simulation result can be obtained. To solve this problem, memory synchronization should be performed between two memories. Details of the memory synchronization mechanisms will be explained in later sections.

3.5 Overall GIL simulation flow

Figure 3-4 shows the simulation flow between host/board interfaces after a GPU request is detected by the detection mechanism in the host interface. First, the arguments for the GPU request are obtained using *readIntReg*() and *readMem*()

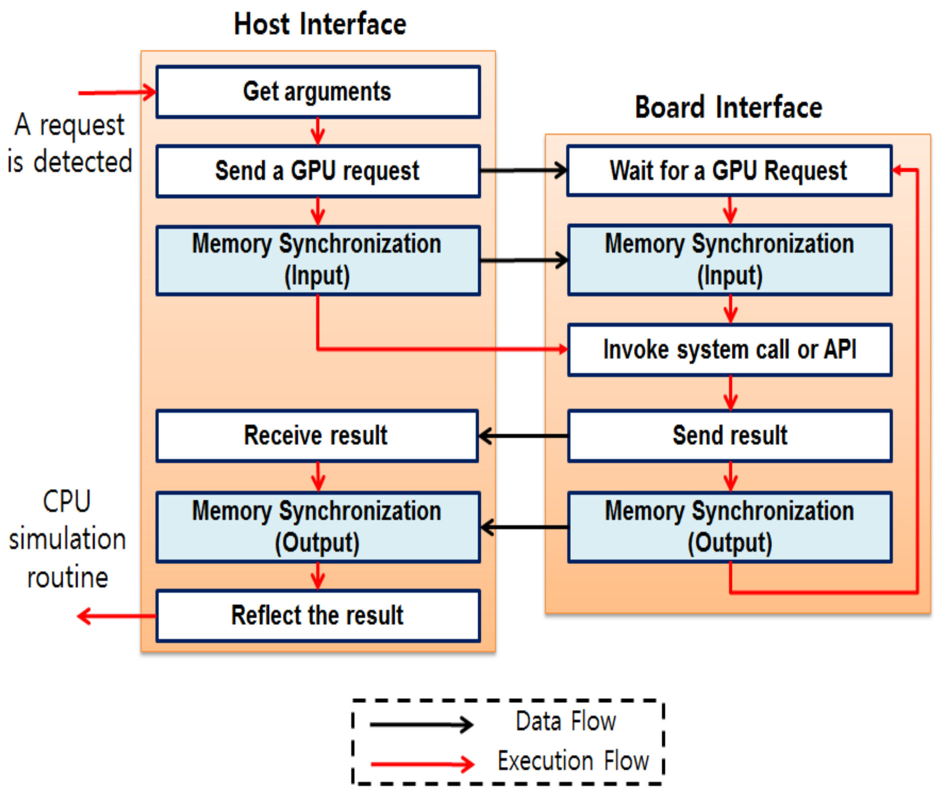


Figure 3-4. Simulation flow between host/board interfaces

functions. These gem5 functions read the value of registers (e.g. r0-r3 that are used for arguments by ARM call conventions) and memory. After that, the GPU request and arguments are sent to the board interface and memory synchronization is performed for input data in the arguments. In the board interface, the GPU request is processed by invoking system call or API with the received arguments. Once it is returned, which means that the GPU execution is completed, the result for the GPU request is sent to back to the host interface and memory synchronization is performed for output data in the arguments. After that, in the host interface, it reflects the result of the GPU request to the related simulation components and the original CPU simulation routine is executed as usual.

Chapter 4 System call- level GIL Simulation

Depending on the GPU, types of system calls and parameters for system calls are varied. Thus, the implementation issues to be considered in the system call-level GIL simulation technique may be slightly different depending on the GPU. For this reason, we assumes the target platform with Mali 400 GPU [38] as shown in Figure 4-1 and the system call level GIL simulation technique will be described with respect to this platform.

4.1 Target System

As low power and energy consumption being the crucial design constraints, GPU has become an inevitable component in the recent embedded systems. Figure 4-1 shows a typical CPU/GPU heterogeneous system: multi-core CPU and multi-core GPU are connected to the on-chip bus where the shared memory and other peripherals are connected to. The GPU usually has its own MMU so that it can directly access the memory on the bus with its own virtual address.

In this platform, GPU requests are processed using several system calls such as *gpu_ioctl()* and *gpu_mmap()*. Thus, whenever these calls are detected inside the host interface, it delivers the corresponding request to the board interface that runs on the real CPU on the board, which in turn requests to the real GPU on the board as explain in Figure 3-1.

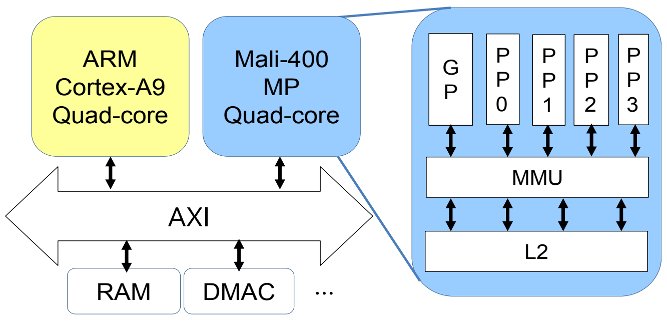


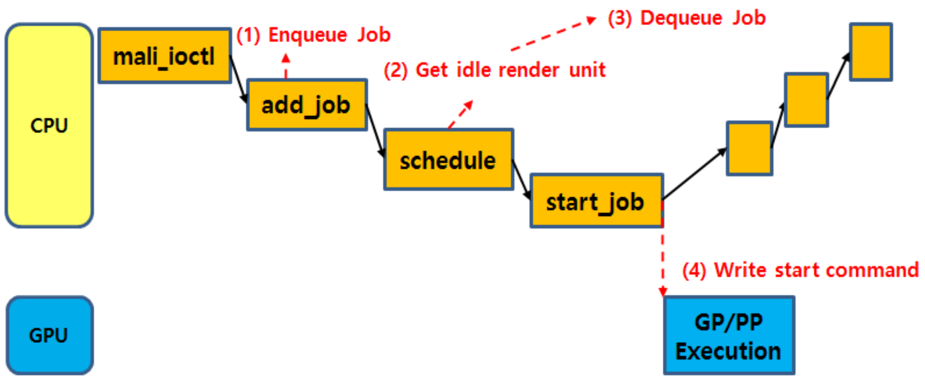
Figure 4-1. CPU/GPU heterogeneous system that integrates an ARM CPU and a Mali 400 GPU: a GPU core represented as PP stands for Pixel Processor, and as GP for Geometry Processor

4.1.1 Typical Execution Scenario of the Systems

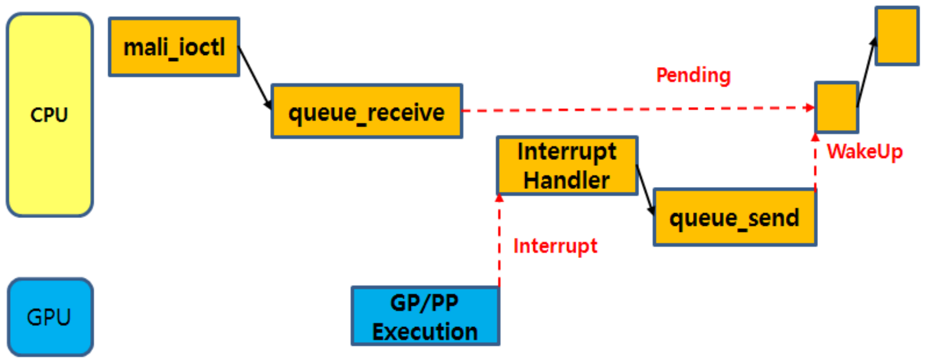
The typical software stack in the target platform includes the Linux kernel and the Android: An Android app runs on top of the Android, which in turn requests services to the Linux kernels. Suppose we have an Android app that utilizes the GPU by calling OpenGL ES APIs. In the APIs, each request for graphics computation is translated into a set of *gpu_ioctl()* calls that have different commands and arguments. There are many use cases of *gpu_ioctl()* with different commands.

Figure 4-2 shows typical scenarios for job enqueue and wait commands in Mali GPU as an example. *mali_core_session_add_job()* in the Linux driver is called first in *enqueue* command. It enqueues the target job to the GPU job queue, and calls *mali_core_subsystem_schedule()*. In this scheduling function, it checks *idle_render_unit_list* to find any idle render unit in the GPU. If there is no idle render unit, it returns without performing any operation. But if there is any, it dequeues a job from the GPU job queue and calls *susbsystem_(gp/pp)_start_job()* with the arguments (job and render unit). In *start_job()*, it writes commands and arguments (start address of input) to GPU registers to execute target job. Lastly, it writes start command to command register, and GPU starts to process the target job.

In wait command, *mali_osk_notificaion_queue_receive()* is called first. In this function, it checks if there is any notification in the notification queue. If there is no notification, it sleeps and current process stops. After some time passed, this process will be woken up by *mali_osk_notification_queue_send()*, which is called by the interrupt handler for the GPU completion.



(a) An execution scenario for *enqueue* command



(b) An execution scenario for *wait* command

Figure 4-2. Typical execution scenarios on a CPU/GPU system with the Linux kernel

4.2 Memory Synchronization

In the target platform, the memory region shared by a CPU and a GPU is allocated by *gpu_mmap()* system call which is called from a GPU library. Since a CPU virtual address is returned to the GPU library as a return value of the system call, in the GPU library, the input data for a GPU is stored to the shared memory region using the address. After that, to allocate some tasks to a GPU, *gpu_ioctl()* system call is invoked with *start* command and the addresses for the input/output data are passed as arguments. In the GPU, a given task is processed using the input data pointed by the input data address and the result is stored to the memory region pointed by the output data address.

Since the memory synchronization is required before/after GPU actually is executed on the board, the memory synchronization is performed when *gpu_ioctl()* is invoked with *start* command. While arguments of *gpu_ioctl()* are memory addresses not real data, input data exists only in the simulator and output data exists only in the board. Before *gpu_ioctl()* is invoked in the board interface, the input data must be sent to the board and the board interface must update the board memory using *memcpy()*. After *gpu_ioctl()* is finished, on the other hand, the modified memory region by the GPU execution must be sent back to the simulator so that the host simulation interface can update the modified memory region in the gem5 simulator.

4.2.1 Address Translation Table

To copy the contents of the memory mapped region for synchronization, a CPU virtual address is needed in the host and board interfaces; The host interface can access the memory only through the CPU model and the board interface itself is a CPU task that actually runs on a CPU on the board, thus it cannot access the mapped memory with the GPU virtual address.

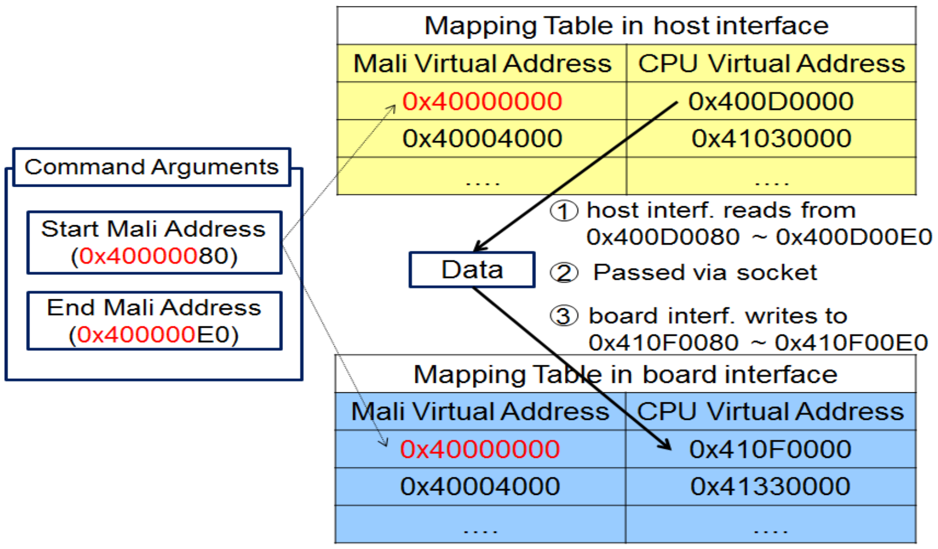

Mapping Table in host interface
Mali Virtual Address
CPU Virtual Address
0x40000000
0x400D0000
0x40004000
0x41030000
....
....
Command Arguments
Start Mali Address (0x40000080)
End Mali Address (0x400000E0)
① host interf. reads from 0x400D0080 ~ 0x400D00E0
Data
② Passed via socket
③ board interf. writes to 0x410F0080 ~ 0x410F00E0
Mapping Table in board interface
Mali Virtual Address
CPU Virtual Address
0x40000000
0x410F0000
0x40004000
0x41330000
....
....

Figure 4-3. Address translation table to match the same memory region

However, since GPU virtual addresses are provided as arguments of *gpu_ioctl(),* we maintain translation tables for GPU to CPU virtual address in each interface as depicted in Figure 4-3. Whenever *gpu_mmap()* is called in the CPU (gem5) side during the simulation, the host interface update its address translation table using the mapping information obtained from the *gpu_mmap*(). Then, when this system call is processed in the board interface, it also update its address translation table.

Figure 4-3 illustrates an example of address translation. When the address given as a command argument in the *gpu_ioctl()* is 0x40000080, the host interface searches the mapping table with 0x40000000 and find out the corresponding CPU virtual address is 0x400D0000 (①). It reads the data from the address considering the offset (0x80), and sends it to the board interface via socket (②). The board interface looks up its translation table and figures out its CPU virtual address to be 0x410F0000 (③). Finally, it writes the received data to the address considering the offset. With such a mechanism, the gem5 simulator and the GPU have an illusion that they share the same memory, although they are in fact two separate memories in different machines in the framework.

From the arguments of *gpu_ioctl()* system call, we can't know which memory regions are modified in the whole memory area. Thus, the simple solution is to synchronize all the shared memory regions. However, since this incurs significant communication overhead between the host interface and the board interface, only diffed data in the shared memory region is synchronized by performing diff operation.

4.3 Timing Modeling

Figure 4-4 shows the typical scenario where GPU execution is controlled by two commands explained in Figure 4-2, *start* command and *wait* command. In this scenario, two commands are called from the two separate threads (*Thread 0*, *Thread 1*). When *wait* command is invoked from *Thread 0*, it is blocked inside *gpu_ioctl()*

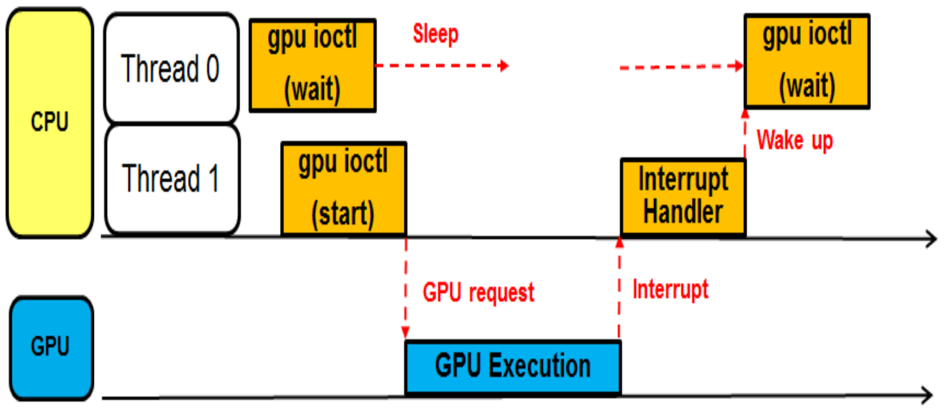


Figure 4-4. Typical execution scenario on the target platform

to wait for the notification of the GPU job completion. In the meanwhile, in *Thread 1*, *start* command is invoked and it will trigger the GPU execution. When the GPU execution is completed, an interrupt in generated by the GPU and an interrupt handler will be invoked. In last, the interrupt handler sends a notification signal which awakes the waiting thread (*Thread 0*).

4.3.1 Interrupt Modeling

Since the original software stack is used in the simulation without modification, the CPU parts in Figure 4-4 simulated by the CPU model. Thus, only GPU part should be modeled in this technique for functional and timing correctness. For functional correctness, the GPU interrupt should be modeled since the waiting thread in Figure 4-4 will be blocked indefinitely if the GPU interrupt is not generated in the simulation. For timing correctness, the waiting timing in the CPU part should be accurately modeled. For this, the GPU interrupt should be generated at accurate timing considering the GPU execution.

To generate the interrupt in the simulator, the virtual GPU model is implemented in the gem5 simulator in which only interrupt related part is modeled without the details of GPU micro-architecture. And the GPU execution time (Δ) can be obtained from the result of *gpu_ioctl()* when it is invoked with *start* command in the board. Once the result is passed to the host interface, an interrupt event is inserted at timestamp $t1 + \Delta$, when Δ is the execution time of the GPU and *t1* is the current simulate time. Then, when the simulation is progressed to $t1 + \Delta$, the interrupt is generated by the virtual GPU model.

4.3.2 Regression based timing correction for GPU time

In this technique, GPU execution time (Δ) is obtained from the real board. However, this value does not include contention overhead between multiple PPs. In the real system, when multiple *gpu_ioctl()* requests for PPs can be made simultaneously and executed in parallel by multiple PPs. As PPs share the resources (cache, memory, bus, etc.), the execution time of each *gpu_ioctl()* request becomes longer than the case when only one PP is executed. In contrast, in the GIL simulation, multiple *gpu_ioctl()* requests are actually processed sequentially.

To consider the contention overhead, we measured the ratio α, which is the ratio of the average execution time with contention (Δ\`) to the one without contention (Δ) in the real board. Then, interrupt is generated at $t1 + \alpha \cdot \Delta$ (= Δ\`) in the simulation. We will explain in more detail how to measure Δ\` in the experiment section.

4.3.3 An Example of System-level GIL Simulation Scenario

The System call-level GIL simulation sequence for the scenario shown in Figure 4-2, which assumes Mali GPU as an example, is illustrated in Figure 4-5 assuming that a *wait* command is called first, followed by an *enqueue* command as shown in Figure 4-4. When the host interface detects the *wait* command, it sends the command to the board interface running on a real CPU in the board. The board interface creates a new thread (*wait* thread) waiting for the completion of the GPU execution to avoid any possible dead-lock. Simulation continues and detects *gpu_ioctl()* for *enqueue* command. Then, the host interface stores *gpu_ioctl()* arguments and calls *add_job()*, *schedule()*, *start_job()* in sequence. In *add_job()*, a job ID is assigned for *start_job()* by which the GPU execution is finally triggered. On the GPU execution, it first sends a memory synchronization message

(MALI_PUT) for input data, and the board interface updates the memory accordingly. Then, it sends *ioctl* messages (GP/PP_START) that are handled by the main thread in the board interface. The main thread sleeps until the completion of the GPU execution is notified by the *wait* thread. On the completion, the main thread sends a message to the host interface, and finally, the host interface sends a memory synchronization message (MALI_GET) to update the modified memory region by the GPU execution.

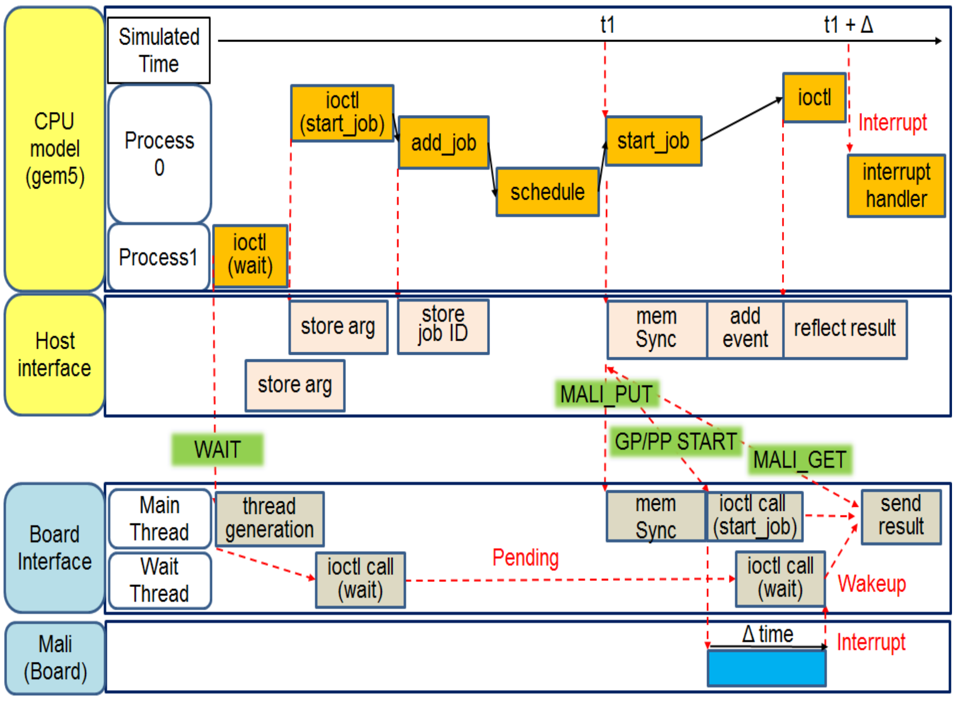


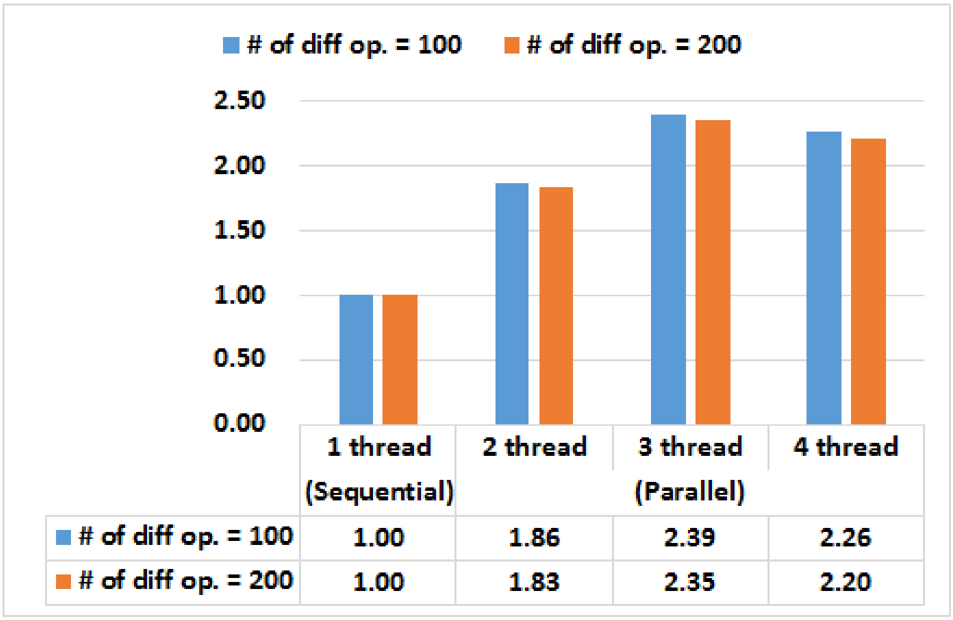
Figure 4-5. An example of the HIL simulation sequence with the scenario shown in Figure 4-2

4.4 Experiments

In our experiments, we simulated the Exynos 4412 system [39]. The system has a quad-core ARM Cortex-A9 CPU and ARM Mali-400MP GPU that has four Pixel Processors and one Geometry Processor. They are connected to an AXI bus where also 256KB on-chip memory is connected. We used ODROID-X board [40] to execute the Mali GPU hardware, and used gem5 simulator for a quad-core ARM Cortex-A9 CPU modeling. We ran Android apps, Lesson09 that moves and blends textured objects in a 3D space [30] and Cubic [31]. In Android OpenGL ES application, a rendering function called *onDrawFrame* in the application is invoked repeatedly to draw the current frame. In the Lesson09 benchmark, the rendering function only includes the API call sequence without any computation. However, in the Rubik benchmark, the rendering function includes both the computation and the API call sequence, in which the proportions of two parts are 37.4% (computation) and 63.6 % (API call sequence) respectively. We ran these apps on the proposed GIL simulation framework for 3 seconds in real time (i.e., the time in the ODROID-X board) and measured the execution time of the rendering function.

4.4.1 Parallelization for diff operation

As mentioned in 4.2.1, to reduce the communication overhead between the simulator and the board, *diff* operation is performed in the host interface and the board interface. Since the size of shared memory region is significantly large, the overhead for the *diff* operation takes large portion of the total simulation time, especially in the board interface. To reduce the *diff* overhead, we parallelize the *diff* operation in the board interface. Figure 4-6 shows the normalized speed-up for the parallel implementations (2, 3, 4 threads) compared with the sequential implementation (1 thread) when *diff* operation is performed 100 and 200 times during the simulation. From the result, we can know that the speed-up of x1.83 ~ x2.39 can be achieved in the parallel implementation and the maximum performance can be achieved when the number of thread is 3.



	1 thread (Sequential)	2 thread	3 thread (Parallel)	4 thread
🟦 # of diff op. = 100	1.00	1.86	2.39	2.26
🟧 # of diff op. = 200	1.00	1.83	2.35	2.20

Figure 4-6. The execution time for *diff* operation for sequential and parallel implementations

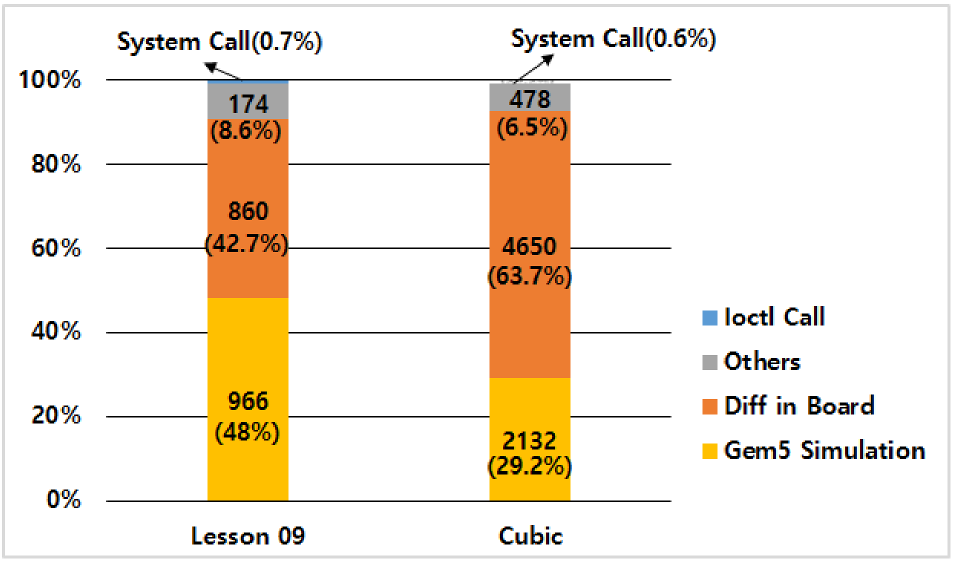


Figure 4-7. Simulation time (sec) for two benchmarks

4.4.2 Simulation Time Analysis

In the system-level GIL simulation technique, the simulation time is decomposed as shown in Figure 4-7. For Lesson09 app, the total GIL simulation takes about 2014 seconds, among which gem5 simulation time takes 48% and the interfacing time between two interfaces takes 52%. It corresponds to about 1.5M cycles per second of simulation performance. In the interfacing time, 860 seconds is spent for memory synchronization, which is 42.7% of the total time.

For Cubic app, the total simulation time is 7304 seconds and achieves about 0.8 M cycles per second of simulation performance. This is because the portion of memory synchronization increased in this application and GPU execution portion is larger than Lesson09.

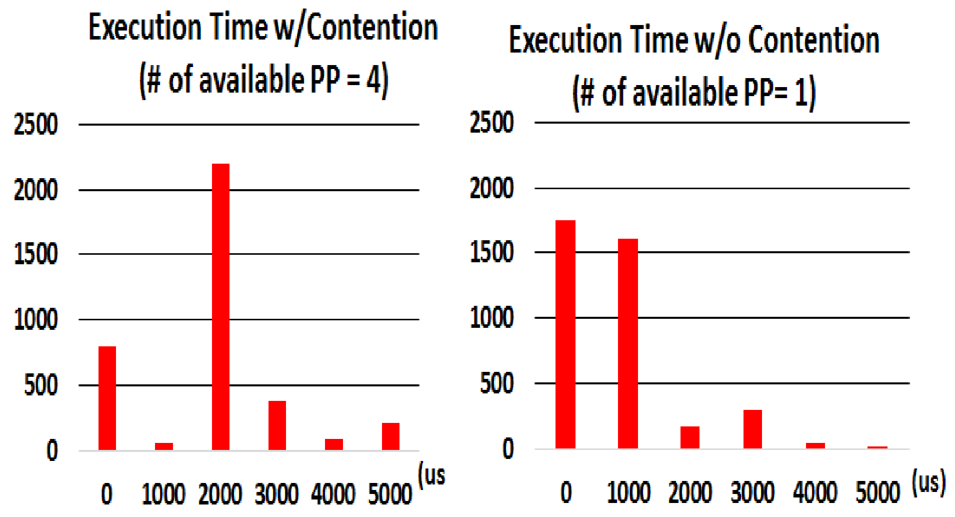


Figure 4-8. Execution time distribution for PPs

4.4.3 Contention overhead in Pixel Processors (PP)

As explained in 4.3.2, we estimate the contention overhead of PPs of the GPU by modifying the number of available PPs from Linux driver. We measured two GPU execution time, Δ\` and Δ, for 4000 *gpu_ioctl()* requests in the real target board; Δ\` is measured by setting the number of available PPs to 4 (all cores are available), and Δ is measured by setting the number to 1 (only 1 core is available so that there is no contention). Figure 4-8 shows the histogram for Δ\` and Δ, where x axis represents the execution time of a single *gpu_ioctl()* request and y axis the occurrence count. The average execution time for the Δ` and Δ are 3059 (us) and 1460 (us) respectively, which results in the ratio α to be 2.1. Thus GPU execution time was scaled by this ratio in the simulation.

4.4.4 Internal System Behavior Profiling

With the proposed GIL simulation framework, we could observe the internal system behavior during the app execution. Table 4-1 shows the execution time, the waiting time, and the response time for each processor (1 GP and 4 PPs). Also, as shown in Table 4-2, we could obtain the GPU utilization. If the app utilizes the GPU not only for the rendering or shading job, but also for the general purpose job such as OpenCL kernel, this observability would be more useful. We could not use OpenCL applications in the current implementation for Exynos system, since Mali 400 GPU does not support GPGPU.

Table 4-1. GPU response time for Cubic app

Processor Type	Avg. Exec. Time(ms)	Avg. Waiting Time(ms)	Avg. Response Time(ms)
Geometry Proc.	0.178	0.029	0.207
Pixel Processor (PP0 ~ PP3)	1.847 ~ 1.866	0.036 ~ 0.041	1.884 ~ 1.906

Table 4-2. GPU execution time and utilization for Cubic app

Processor Type	# of GPU Execution	Total Exec Time (ms)	GPU Utilization (%)
Geometry Proc.	506	90.195	1.53
Pixel Processor (PP0 ~ PP3)	505 ~ 506	932.966 ~ 943.994	15.85 ~ 15.99

Table 4-3. Accuracy evaluation for the Android apps

Accumulated Execution Time	Real Board (sec)	Simulation (sec)	Error Ratio (%)
Lesson09	1.21 ~ 1.57	0.99 ~ 1.42	-29.66 ~ - 5.22
Cubic	0.76 ~ 1.02	0.77 ~ 2.09	- 1.53 ~ + 104.91

4.4.5 Accuracy Evaluation

To evaluate the timing accuracy of the proposed framework, we measured the execution time of a rendering function that calls several OpenGL ES APIs, by inserting time stamping code to the application. We accumulated the execution time for 50 invocations of the rending function. We performed the experiment for 5 runs both on ODROID-X board and on the simulation framework. The range of execution time and the error ratio are shown in Table 4-3. It confirms that the accuracy of the proposed GIL simulation technique is about the same order of the gem5 simulator that is simulated at the instruction-level. We observed that the accuracy error gets smaller as we run an app longer. Since Cubic runs longer than Lesson09, the accuracy error of Cubic gets smaller than Lesson09. More detailed analysis on the accuracy is left for future investigation.

4.5 Summary

In this chapter, we have proposed a system call-level GIL simulation technique for CPU/GPU platforms that integrates a real GPU hardware instead of GPU simulator for full system simulation, running complete software stack without modification. We devised a novel interfacing mechanism between a CPU simulator and the GPU hardware. For correct operation, several issues had to be considered, including memory synchronization, address translation, and interrupt handing. We took Exynos 4412 system as our case study and ran two Android apps where a number of OpenGL ES APIs were called. To the best of our knowledge, it is the first example of full system simulation of a CPU/GPU heterogeneous system. We can achieve simulation performance up to 1.5 M cycles per second.

Chapter 5 API-Level GIL Simulation

System call-level GIL simulation technique has a limited extensibility. In GPU, *ioctl* system call is widely used to process the device-specific operations. It takes a parameter specifying a request code and the request code is often device-specific. Therefore, to support other GPUs, the simulation framework should be modified to consider the new request code unless other GPUs have the same request codes for *ioctl* system call. Moreover, to correctly simulate the device driver in the original software stack, some GPU specific functionalities such as GPU registers and interrupts should be modeled for functional correctness, which requires considerable effort to understand interactions between the device driver and the GPU registers.

5.1 Differences between API-level and System call-level techniques

An API (Application Programming Interface) is usually defined independent of the HW for portability. For GPU APIs such as OpenGL ES and OpenCL, the application written with APIs can run on various CPU/GPU platforms without any modification. Thus, if the GIL simulation technique is performed at API level instead of system call level, the GIL simulation can be performed with various GPU boards with minor modification for the simulation framework.

Even if the simulation is performed at API level, if the original software stack is used in the simulation, some GPU specific functionalities related with the device driver in original software stack need to be modeled. To further reduce the GPU dependency from the simulation code, the device driver in original software stack should not be used in the simulation. Since the device driver is accessed from the GPU libraries through system calls, if the original GPU libraries are not used in the simulation, the device driver is also no longer used during the simulation. For this reason, in the API-level GIL simulation, the GPU libraries in the original software stack is replaced by the synthetic library which implements stub functions for the APIs of the original GPU libraries as shown in Figure 5-1.

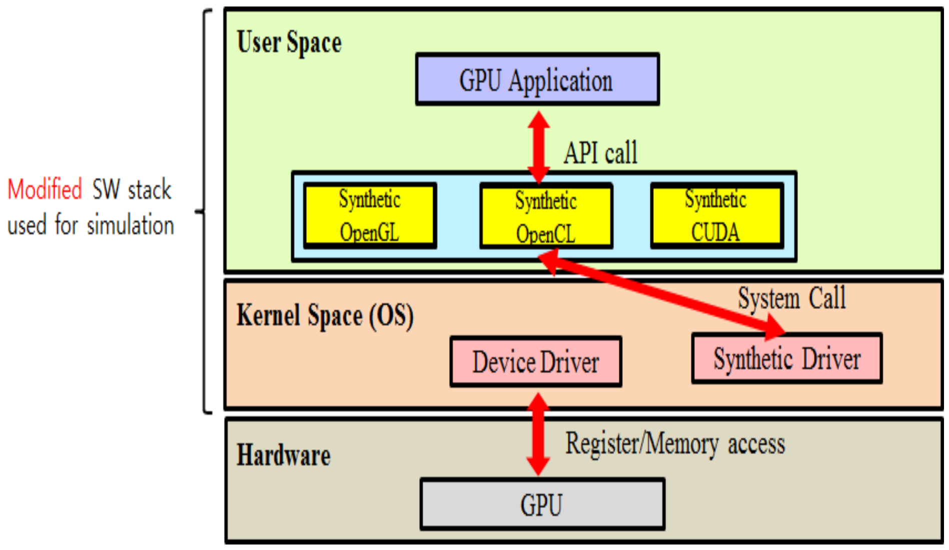


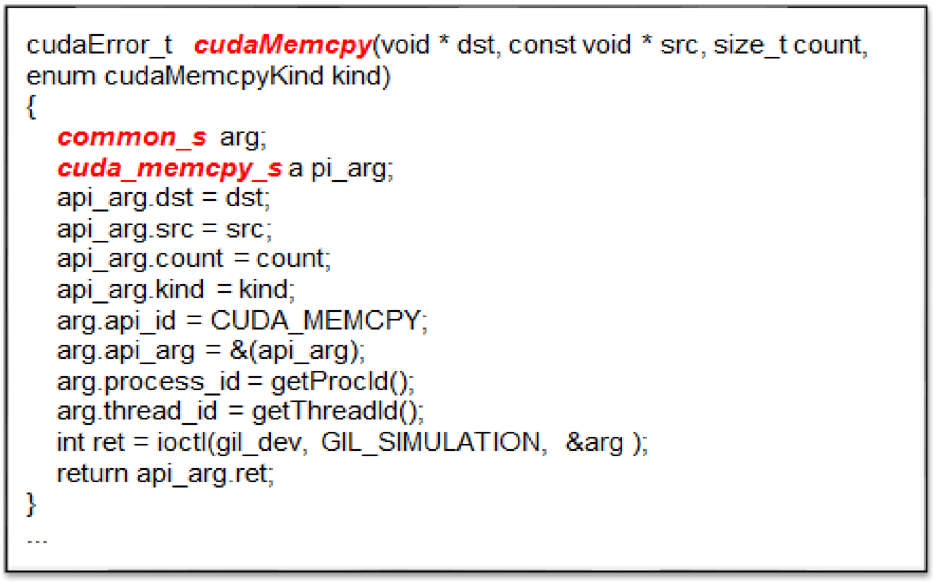
Figure 5-1. Modified SW stack in API-level GIL simulation

For the API-level simulation technique, the GPU request should be detected when an API is invoked from the GPU applications. Since the target address is required in our detection mechanism explained in section 3.3, the start address of APIs defined in the synthetic library should be known to detect the API request. However, it is not easy to know the address since the synthetic library is located in user space and the address is determined when the library is loaded by a dynamic linker. To know the address, we should track the linking process during the simulation, but this would incur considerable overhead to the simulator. Instead, since the addresses of functions in kernel space can be obtained from the kernel image before the simulation, the synthetic driver is added in OS kernel for this purpose. Thus, in the synthetic library, it just forwards the API requests to the synthetic driver without any operation and the API request is detected in the synthetic driver by the detection mechanism.

5.1.1 Synthetic Library

Figure 5-2 shows an example code of the synthetic library for *cudaMemcpy* API in the CUDA library. To share the API information between the simulator and the board, two structures represented in Table 5-1 and Table 5-2 are used.

The common structure (*common_s*) contains the data commonly used in all APIs. It has 5 variables; *api_id* variable is used to notify which API is invoked from the



```
cudaError_t  cudaMemcpy(void * dst, const void * src, size_t count,
enum cudaMemcpyKind kind)
{
   common_s  arg;
   cuda_memcpy_s a pi_arg;
   api_arg.dst = dst;
   api_arg.src = src;
   api_arg.count  = count;
   api_arg.kind  = kind;
   arg.api_id = CUDA_MEMCPY;
   arg.api_arg  = &(api_arg);
   arg.process_id = getProcId();
   arg.thread_id  = getThreadId();
   int ret = ioctl(gil_dev,  GIL_SIMULATION,  &arg );
   return api_arg.ret;
}
...
```

Figure 5-2. An example code of the synthetic library for *cudaMemcpy* API

Table 5-1. Common structure (*common_s*)

Variable name	Description
api_id	An identifier for target API
thread_id	Thread id for the thread calling the API
process_id	Process id for the process calling the API
api_arg	A pointer to an API specific structure
api_time	API time measured in the board

Table 5-2. API-specific structure for *cudaMemcpy* API (*cuda_memcpy_s*)

Variable name	Description
src	Source memory address
dst	Destination memory address
count	Copied memory size
kind	Direction for memory copy
ret	Return value

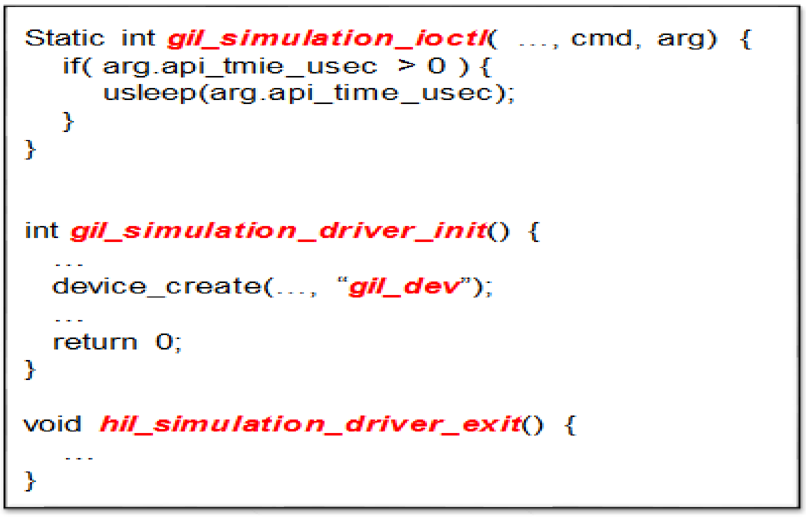
application. *thread_id* and *process_id* variables are used to distinguish the thread and process calling the API. *api_time* variable is used to store the API time measured in the board and this variable is set by the host interface when the API time is passed from the board interface. And, *api_arg* variable is a pointer to an API-specific structure.

The API-specific structure contains all arguments for the target API and it is varied depending on the target API. For example, in case of *cudaMemcpy* API, *cuda_memcpy_s* structure is used and contains four variables: destination memory address (dst), source memory address (src), copied memory size (count), and direction for memory copy (kind). After variables in the two structures are set using arguments passed from the application, *ioctl* system call is called with a pointer to the common structure to pass the API request to the synthetic driver. Once the *ioctl* is returned, since the return value has been set in *ret* variable in API-specific structure by the host interface, it is returned to the application.

5.2 Timing Modeling

In the system call-level GIL simulation technique, the GPU execution time is reflected to the simulator by the interrupt. However, since the device driver is not simulated in the API-level GIL simulation technique, the interrupt-based timing modeling technique can’t be used. Instead, since we can measure the API execution in the board by inserting timestamping code before/after the API invocation, the API execution time is reflected to the simulator by spending that amount time in the synthetic driver.

Figure 5-3 illustrates the implementation of the synthetic driver. In Linux booting phase, *gil_simulation_driver_init* is invoked and it creates the virtual device named *gil_dev*. After the initialization of the device, it can be accessed as a file using *open* and *ioctl* system calls from the synthetic library. When *ioctl* is called for *gil_dev* device from the synthetic library as shown in Figure 5-2 (the first argument of *ioctl* call), *gil_simulation_ioctl()* will be invoked. When the first instruction for this function is executed on the CPU simulator, it is detected by the detection routine in the host interface and the API is request sent to the board interface. In the board interface, target API is invoked and the API execution time is measured using time functions such as *gettimeofday()* and *clock_gettime()*. Then, the return value and the execution time for the API are sent back to the host interface and the API execution time is stored into *api_time* variable in the common structure. After the simulation for the target API has been completed, the original CPU simulation routine executes *gil_simulation_ioctl* code shown in Figure 5-3. To model the timing, it simply calls the *usleep* function so that the measured API time (*api_time*) elapses in the simulated time.



```
Static int gil_simulation_ioctl( ..., cmd, arg) {
   if( arg.api_tmie_usec > 0 ) {
      usleep(arg.api_time_usec);
   }
}


int gil_simulation_driver_init() {
   ...
   device_create(..., "gil_dev");
   ...
   return 0;
}

void hil_simulation_driver_exit() {
    ...
}
```

Figure 5-3. Synthetic driver code used in GIL simulation

5.2.1 Regression-based compensation for timing error

In the API-level simulation technique, the timing accuracy is first bounded by the processor simulator that measures the execution time only at the instruction level. For timing estimation of GPU execution, we estimate the execution time of each API by measuring the execution time directly in the board. The GPU execution time is modeled by simply summing up all the estimated API times and added to the CPU simulation time by using the *usleep* function explained in the section 5.2. This simple method itself is a source of timing inaccuracy. There are other sources of timing inaccuracy. In the board, time stamping is inserted before and after an API is called and the execution time is estimated by subtracting two time stamps. If the simulated system architecture or API implementation is not the same as the board, timing inaccuracy is inevitable. Even if we use the same architecture, time stamping affects the internal behavior. Also, *usleep* function has some overhead to set up the timer and this makes the elapsed time by the *usleep* will be larger than the measured API time.

Since most internal implementation of GPU libraries are proprietary and the source code is not available, it is impossible to model the low level details of API interaction. Hence we perform a simple linear regression analysis to compensate the unknown sources of time inaccuracy with some selected benchmarks. We compute the ratio between the measured execution time of an application in the actual board and the simulated time in the proposed simulator. We adjust the simulated time of another benchmark by this ratio and compare it with the measured execution time.

5.3 Memory Synchronization

Since the fact that one logical memory is modeled by two separate memories in the simulator and board is not changed, the memory synchronization is also a key

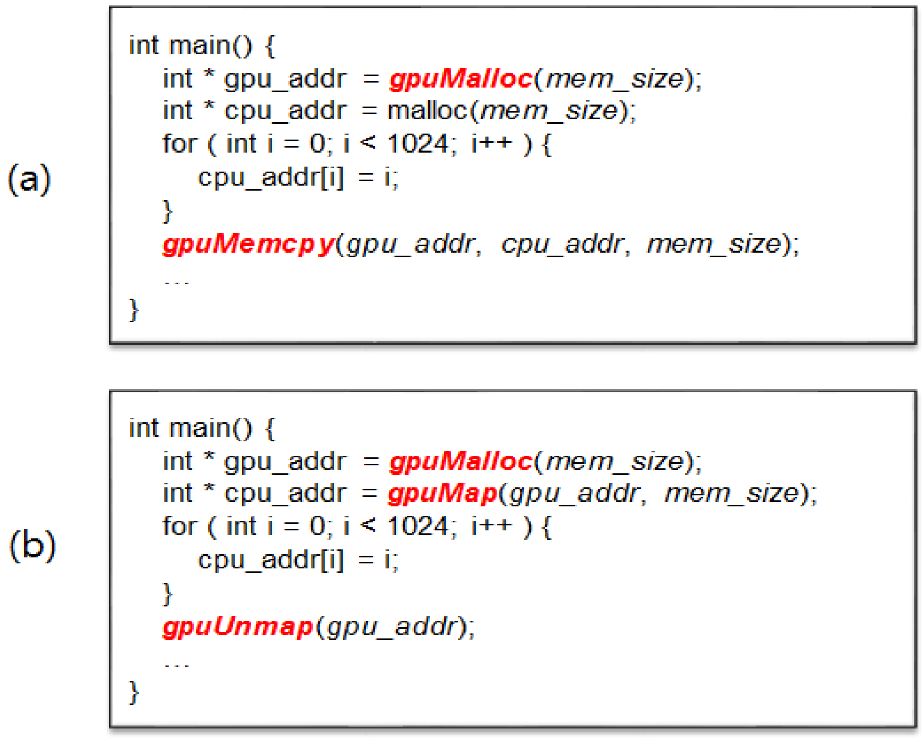
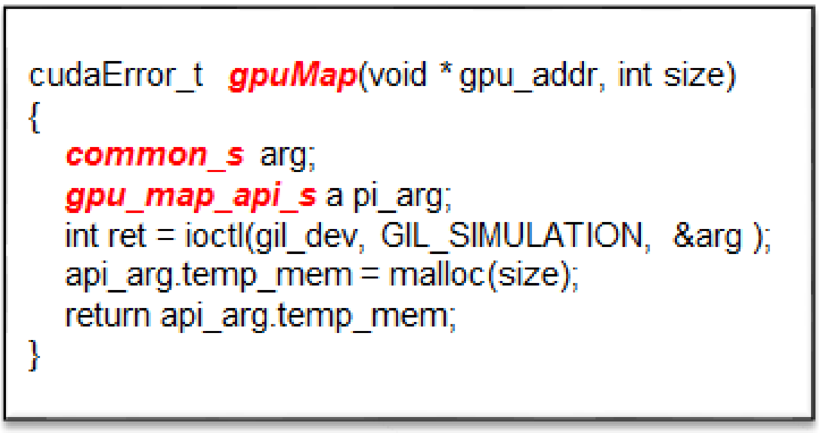

(a)
```
int main() {
    int * gpu_addr  = gpuMalloc(mem_size);
    int * cpu_addr  = malloc(mem_size);
    for ( int i = 0; i < 1024; i++ ) {
        cpu_addr[i] = i;
    }
    gpuMemcpy(gpu_addr, cpu_addr, mem_size);
    ...
}
```
(b)
```
int main() {
    int * gpu_addr  = gpuMalloc(mem_size);
    int * cpu_addr  = gpuMap(gpu_addr, mem_size);
    for ( int i = 0; i < 1024; i++ ) {
        cpu_addr[i] = i;
    }
    gpuUnmap(gpu_addr);
    ...
}
```

Figure 5-4. Two ways to share data between CPU and GPU in GPU applications

issue for correctness in API-level GIL simulation. However, since the original SW tack is not used in the API-level simulation, memory synchronization is somewhat different.

In API-level GIL simulation, there are two types of memory sharing between a CPU and a GPU. In the first way shown in Figure 5-4 (a), the application allocates a memory region (*cpu_addr*) and it is copied to a GPU memory (*gpu_addr*) using memory copy API (*gpuMemcpy*) after it is modified. Since the memory region allocated by *malloc* is not simulated in the board, it is only allocated in the simulator memory and not exist in the board. When *gpuMemcpy* is simulated in the board, it will access the board memory using the given CPU address (*cpu_addr*), but invalid pointer error would occur since this address is not defined in the board memory. To solve this problem, a new memory region is allocated temporally in the board memory and the input data in the simulated memory is copied to newly allocated board memory. Then, the address for the allocated memory region is passed as an argument of *gpuMemcpy* instead.

In the second way shown in Figure 5-4 (b), to directly access the GPU memory in this application, *gpuMap* API is invoked to obtain a CPU virtual address for the GPU memory. When this API is simulated in the board, the pointer for the board memory (*cpu_addr*) is returned and it is used to access the GPU memory in the application. However, since this address is not defined in the simulator memory, segmentation fault error will occur in the simulator when the address is accessed and the application will be aborted. To solve this problem, similar to the former case, temporal memory region is allocated in the simulator memory and it is used as a return value for *gpuMap* API by modifying the *gpuMap* API code in the synthetic library like Figure 5-5.



```
cudaError_t  gpuMap(void *gpu_addr, int size)
{
   common_s  arg;
   gpu_map_api_s a pi_arg;
   int ret = ioctl(gil_dev,  GIL_SIMULATION,  &arg );
   api_arg.temp_mem = malloc(size);
   return api_arg.temp_mem;
}
```

Figure 5-5. Synthetic Library code for *gpuMap* API

5.4 GPGPU API (CUDA & OpenCL) Implementation Case

5.4.1 Asynchronous Behavior Modeling

In the API-level GIL simulation, as explain in section 5.2, the API measured in the board is annotated to the simulated time by *usleep* function in the synthetic driver to model the API timing. This simple mechanism assumes that the APIs are synchronous, which means that code execution will wait until the actual execution of the API is completed. For the synchronous API, its execution time is not changed depending on other APIs or the call time. Thus, we can guarantee that the measured

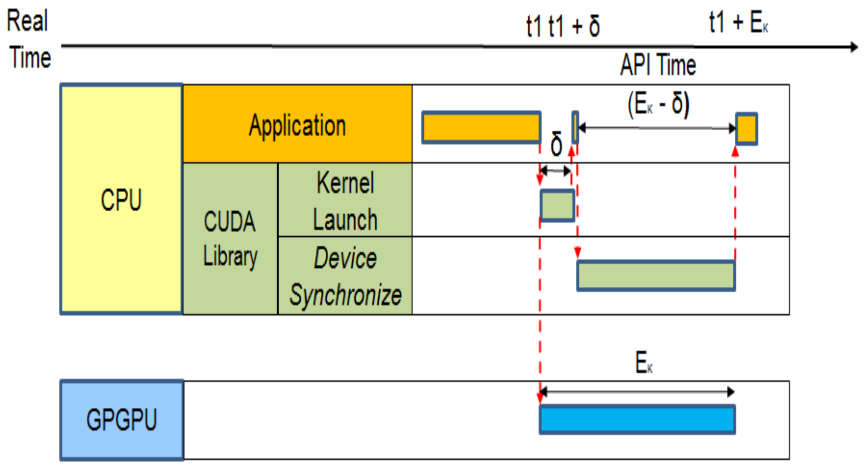


Figure 5-6. Real execution scenario for the synchronization API

API in the board during the simulation is same with the one in the real execution. However, this is not true for asynchronous APIs.

Figure 5-6 shows the scenario that a kernel launch API (*gpuKernelLaunch*) and a device synchronization API (*gpuDeviceSynchronize*) are called in series. As *gpuKernelLaunch* is asynchronous, it is returned after δ time without waiting for the kernel completion. And *gpuDeviceSynchronize* is called immediately to wait until the kernel completes. Since the execution time of the kernel is Ek, the execution time for device synchronization API becomes Ek – δ.

Let us consider the simulation scenario for the Figure 5-6, which is shown in Figure 5-7. When *gpuKernelLaunch* is invoked and captured by the host interface at time t1, it is simulated by invoking *gpuKernelLaunch* in the board and the execution time (δ) is reflected to simulated time as explained. Then, *gpuDeviceSynchronize* is invoked in the CPU model at time t1 + δ and it is also captured by the host interface. When *gpuDeviceSynchronize* is invoked in the board interface, the execution time of *gpuDeviceSynchronize* would be almost zero, not Ek since the kernel launched by *gpuKernelLaunch* has been already completed by the real GPU at that time. This is because the CPU model is considerably slower than the real GPU hardware.

Since the problem occurs for the asynchronous API followed by a synchronous API, the execution time should be calculated differently. As the synchronization API waits until all the operations launched by the previous APIs finish, the execution time of the synchronization is determined by the end time of the last completed operation and the invocation time of the synchronization API. To figure out the end

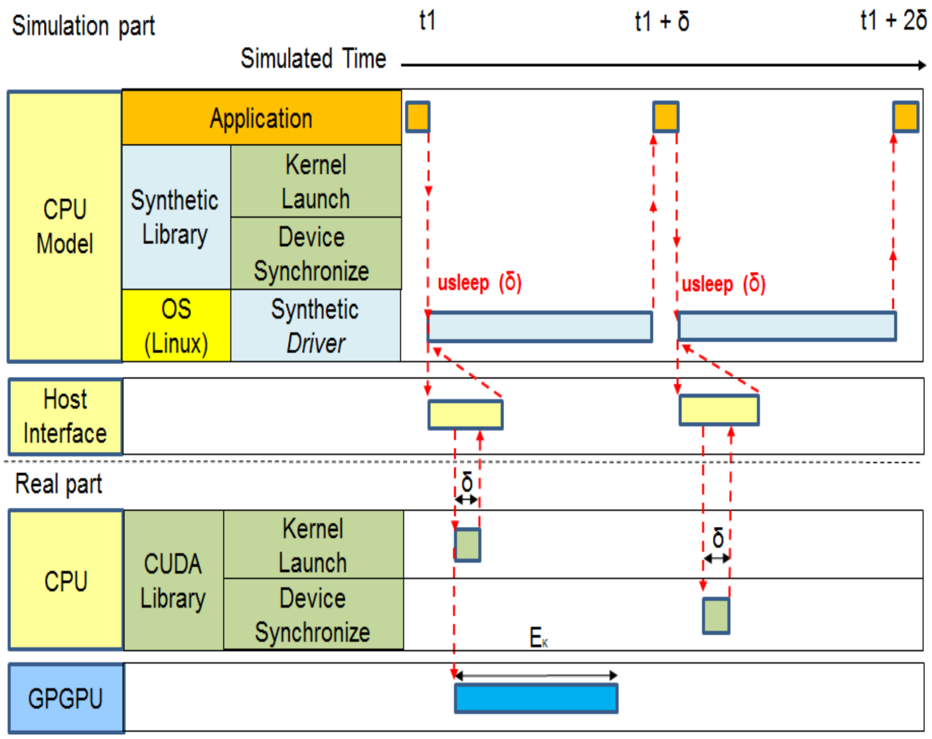


Figure 5-7. Simulation scenario for Figure 5-6

time of the last finished operation, we need to know the end times of all operations. To obtain the end time of each asynchronous operation, a dummy synchronization API is invoked to explicitly wait until the operation launched by asynchronous API completes. In such a way, the end times of all operations including asynchronous APIs are obtained and stored, and these values are used when a synchronization API is called: an API queue is managed to store the asynchronous APIs tagged with the end time. When a synchronization API is called, the API queue is searched to find out the time when the last asynchronous operation finished. Then, the execution time of a synchronization API is calculated by subtracting the invocation time from the last end time.

5.4.2 Implementation Issues

5.4.2.1 Locating Source/Binary Files

For the Android GPGPU application, it consists of a host code and a kernel code which are executed in a CPU and a GPGPU respectively. For the host code, it is statically compiled and the host executable (*apk*) is used when the application runs on Android. However, for the kernel code, both source and binary files are used to build the kernel dynamically. When the GPGPU API is invoked from the host executable to pass the source or binary files, there are two ways. One is that the host executable passes the pointer to a string which contains the whole contents of the source file. In this case, the file system call should be invoked to get a code string from the source file. The other is to pass the file name, and the file is loaded in the API internally.

For the former case, since the OpenCL host code will access the disk in the target system, source or binary files should be included in the image file for the Android file system, which is used to model the disk in the target system. However, for the latter case, since an API is executed in the simulation host not in the target system and it access the disk in the simulation host with the given file name, source or binary files should be located in the same path with the simulator.

5.4.2.2 CUDA Implementation

The proposed simulation technique supports CUDA on Android as well as OpenCL. Even though the basic mechanism of defining APIs for the synthetic library is the same, there are some CUDA specific implementation issues.

In CUDA, kernel arguments are passed by a pointer from the application using *cuLaunchkernel*. But the simulator cannot directly figure out the information such as the number of arguments and the size of each argument from the pointer. To solve this problem, we add a separate utility function that analyzes a *ptx* (Parallel Thread Execution) file, which is a pseudo-assembly code generated by the CUDA compiler. Since the *ptx* file is loaded by *cuModuleLoad* before *cuLaunchkernel* is invoked, we can obtain the necessary information by calling the analyzing routine before *cuLaunchkernel*. Then, *cuLaunchkernel* can be called with the correct information. Another issue is regarding the CUDA building with an ARM compiler. In the proposed framework, we assume that all the requests to the GPGPU are performed in a form of API call. For CUDA, however, there are some expressions which are not in this form. In CUDA, as shown in Figure 5-8 (a), it is possible to launch a kernel from the host using the notation below as the CUDA compiler (*nvcc*) can accept this notation.

kernel <<<grid, block >>> (arg1, arg2, ...)

However, as the ARM compiler is used to compile the host code in the proposed framework, this is not accepted. Hence, the application should be written in the form of an API call like Figure 5-8 (b) to use the proposed simulation framework. Also, since the ARM compiler cannot compile the kernel source code, the kernel and the host code should be split into separate files, which is not necessary in the

```
__global__ VecAdd_kernel(int * A, int * B, int * C)
{
   int index = threadIdx.x;
   C[index] = A[index] + B[index];
}

void vecAddGPGPU( int * h_A, int * h_B, int * h_C, int N)
{
   int * d_A, * d_B, * d_C;
   CUmodule cuModule;
   CUfunction cuFunc;
   ...
   VecAdd_kernel<<<grid, block>>>(d_A, d_B, d_C);
   ...
}
```

(a) Original CUDA Code (*.cu)

```
__global__ VecAdd_kernel(int * A, int * B, int * C)
{
   int index = threadIdx.x;
   C[index] = A[index] + B[index];
}
```

```
void vecAddGPGPU( int * h_A, int * h_B, int * h_C, int N)
{
   int * d_A, * d_B, * d_C;
   CUmodule cuModule;
   CUfunction cuFunc;
   ...
   cuModuleLoad(&cuModule, "*.ptx");
   cuModuleGetFunction(&cuFunc, cuModule, "VecAdd_kernel");
   void * args[3] = {&d_A, &d_B, &d_C};
   cuLaunchKernel(cuFunc, ..., args, 0);
   ...
}
```

(b) Modified CUDA (*.cu and *.cpp)

Figure 5-8. Original (a) and modified (b) CUDA code

original CUDA because *nvcc* can compile both the host and the kernel code in a same file. Since this form of CUDA application code is also allowed in the original CUDA environment, it can guarantee that the application code used in the simulation can run on the real target without any modification.

5.4.3 Experiments

The GPGPU API level GIL simulation framework is constructed based on the system call level framework explained in Chapter 4 and extended to support GPGPU applications on Android. Since the GPGPU is only supported in the host GPGPU and not supported in the GPU board currently, host GPGPU in the simulation host is used for the GPGPU API level GIL simulation technique. Since two GPU functionalities (Graphics and GPGPU) are simulated by different GPUs in the simulation host and the board (Mali GPU), this assumes the target platform with two GPUs, one takes charge of rendering jobs, whereas the GPGPU is only used as an accelerator.

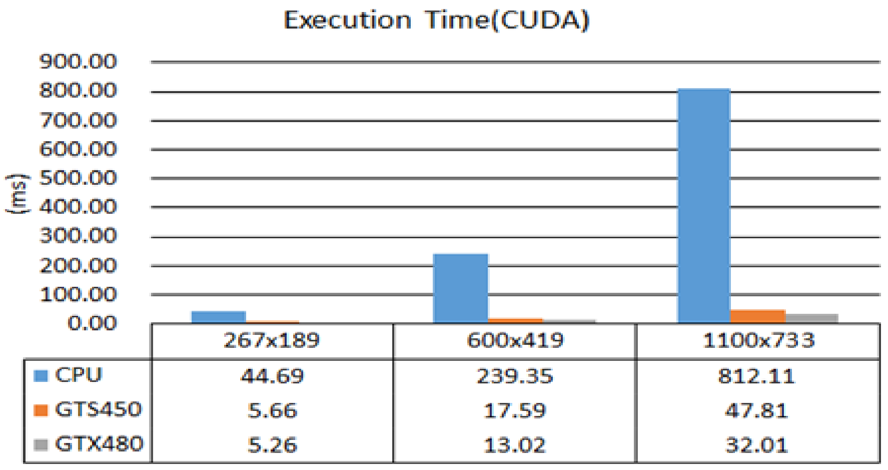
We used ODROID-X board to execute the Mali GPU hardware, gem5 simulator to simulate components other than GPU, and the GPGPU hardware in the simulation host machine for the GPGPU simulation.

5.4.3.1 GPGPU Performance

The first set of experiments is performed to show that the proposed framework can monitor the performance of GPGPU applications written with CUDA and OpenCL. For this experiment, we implemented face detection applications based on two source codes. One is the face detection sample code in OpenCV [43], which has two versions: CPU and GPGPU version written with CUDA. The second one is implemented in OpenCL. Since these two implementations parallelize the application differently, direct comparison between two implementations is not meaningful in this experiment.

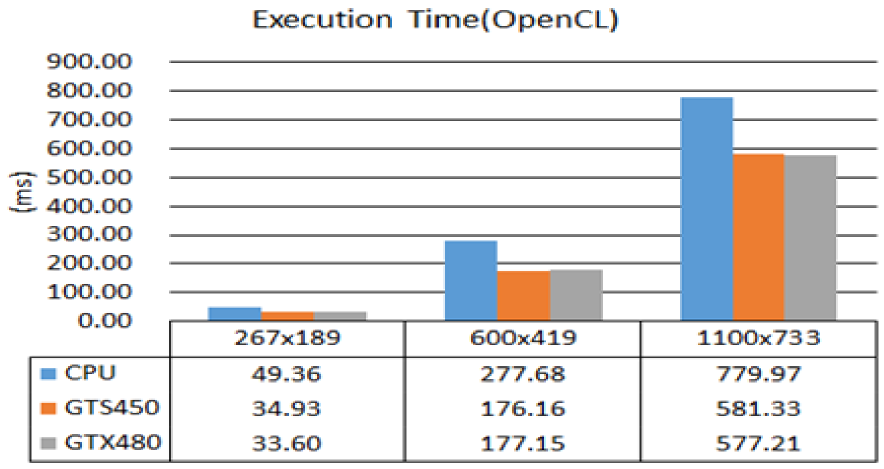
Three images in PGM (Portable Gray Map) format are used in the experiment varying the image size: 267x189, 600x419 and 1100x733. Since the CPU simulation model used in the proposed framework is not cycle-accurate (*AtomicSimpleCPU* in gem5 simulator [45]), we measured the execution time of the CPU version on ARM cortex A9 CPU board (ODROID-X). And the execution time of the GPGPU version is measured from the GIL simulation framework varying the GPGPUs used in the simulation: GTS450 (192 cores) and GTX480 (480 cores). The result is shown in Figure 5-9. As shown in Figure 5-9 (a), the CUDA version with GTS450 is 7.90 ~ 16.98 times faster than the CPU version, and the speedup is increased as the image size grows. With GTX 480, we observe the similar performance increase, 1.07 ~ 1.49 times faster than GTS450.

For the OpenCL application, GPGPU version (with GTS) is 1.34 ~ 1.58 times faster than CPU version. The speedup is not as large as the CUDA implementation, because not all algorithms were parallelized using OpenCL: Image resizing is executed on the CPU. Also, since the CPU model used in the simulation is not



	267x189	600x419	1100x733
🟦 CPU	44.69	239.35	812.11
🟧 GTS450	5.66	17.59	47.81
⬜ GTX480	5.26	13.02	32.01

(a) CUDA application



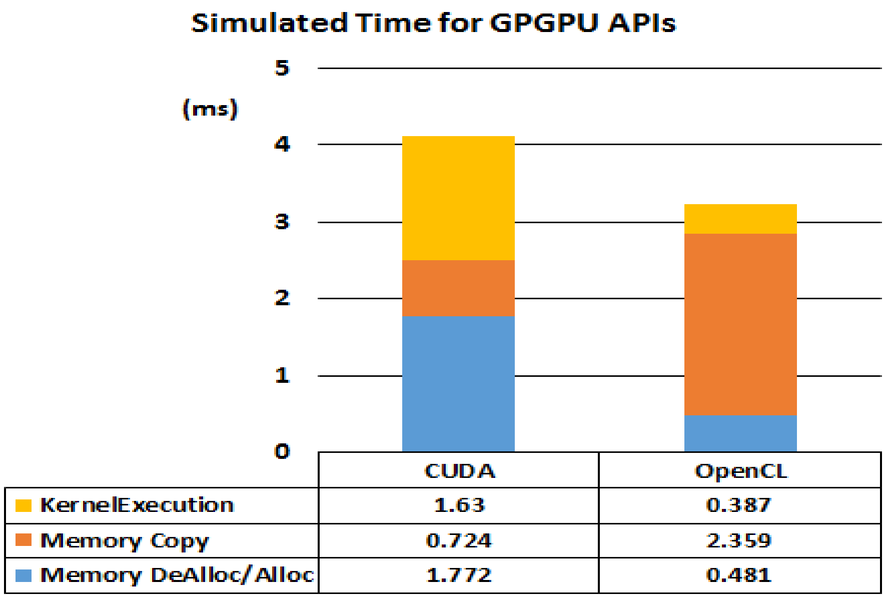
	267x189	600x419	1100x733
🟦 CPU	49.36	277.68	779.97
🟧 GTS450	34.93	176.16	581.33
⬜ GTX480	33.60	177.15	577.21

(b) OpenCL application

Figure 5-9. The execution times of the two applications (CUDA, OpenCL)

cycle-accurate and is overestimated, the execution time on the CPU would be a bit larger than it should be, increasing the total execution time. For the accurate result, cycle accurate CPU model in gem5 simulator (O3 model) should be used at the expense of slower simulation.

Figure 5-10 shows the simulated time for GPGPU APIs (267x189-sized image) that is partitioned into three parts: Kernel Execution, Memory Copy and Memory Allocation. In CUDA implementation, image resizing is performed in GPGPU and GPGPU memory space is allocated/de-allocated each time, which makes memory allocation the most time consuming part, and the kernel execution the next. In contrast, in OpenCL implementation, image resizing is performed in the CPU and the resized images are copied to GPGPU memory. Thus, memory copy is the most timing consuming part in OpenCL.



	CUDA	OpenCL
🟨 KernelExecution	1.63	0.387
🟧 Memory Copy	0.724	2.359
🟦 Memory DeAlloc/Alloc	1.772	0.481

Figure 5-10. Simulated time for GPGPU API (267x189)

Figure 5-11 shows the detailed performance profiling of the face detection applications, obtained from the proposed simulation framework. In the CUDA implementation, the classification part, *lbp_cascade* kernel, takes the most of the time but the sum of *vertical_pass* and *horizontal_pass*, which are the feature extraction parts, is larger. On the other hand, we can see that *lbp_imageGPU_1x1_aggr* which is a feature extraction part, takes a bit larger than the *processingRectLoop_WS_2D* which is a classification part. In Figure 5-11, it shows the communication overhead between CPU and GPGPU. As seen in the 5-11 more memory copy are occurred in OpenCL by *clEnqueueWriteBuffer* which transfers data from host memory to GPGPU memory.

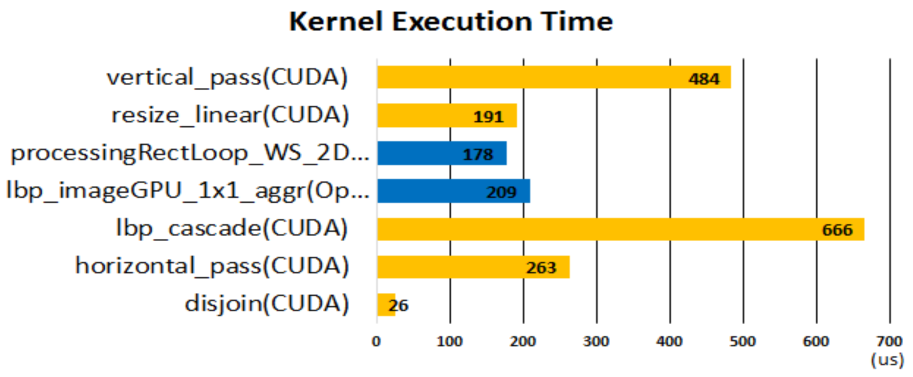


Figure 5-11. The execution time of kernel executed for the face detection application (267x189)

5.4.3.2 CPU/GPGPU Job Sharing

As we explained in Chapter 1, it is good to consider job sharing between CPU and GPU to fully utilize the system, since embedded GPUs are not powerful like server GPUs. To rapidly and easily estimate the performance impact of different CPU-GPU partitioning configurations, another set of experiments is performed with a matrix multiplication application that is written in CUDA and *Pthread*. We parallelize the computation by rows so that each row can be executed in the GPGPU or in the CPU. Although the target embedded GPU in the Android device should be used, the simulation is performed using the GPGPU in the simulation host since the OpenCL driver for the embedded GPUs is not publicly available yet. In order to invoke a target API in the embedded GPU board, information such as API arguments and which API is to be called needs to be passed to the board from the simulation host, which is straightforward to implement but remains as a future work due to the availability of the driver. In the current simulation environment, we limit the number of threads in the GPGPU from 8 to 64 to approximately model the performance of an embedded GPU. For the CPU, we fixed the number of *Pthreads*

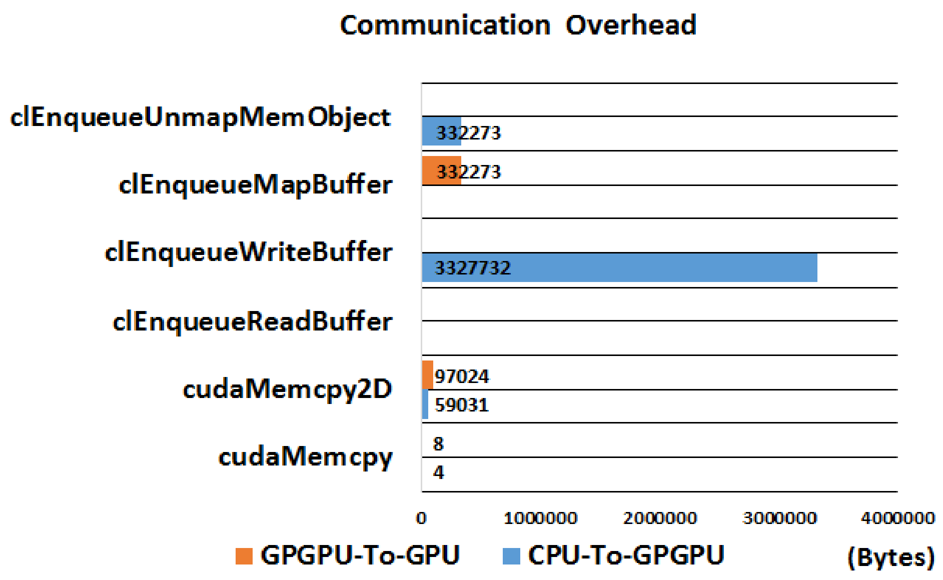


Figure 5-12. Communication overhead for memory APIs (267x189)

to four since ARM Cortex-A9 quad-core CPU is simulated in the experiment.

Figure 5-12 shows the execution time of the application when the number of rows allocated to CPU and GPGPU varies from 0 to 128. As expected, the optimal job distribution point depends on the GPGPU computing power, and each optimal point is shown in Figure 5-13. Even if the proposed simulation is not cycle-accurate, this information is useful as the approximate performance trend of the GPGPU application can be estimated.

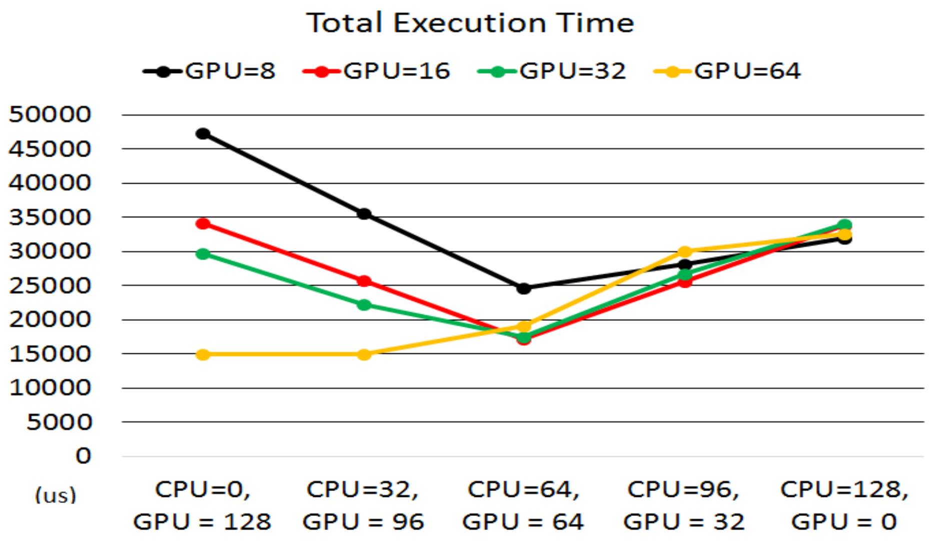


Figure 5-13. The execution time for the matrix multiplication varying the number

5.4.4 Simulation Overhead

To figure out the overhead of the proposed GIL simulation, we measured the detailed simulation time for the face detection application, which is shown in Figure 5-14. From the figure, we can see that the overhead of the GIL simulation for GPGPU is about 3.0 % of the total simulation time. This confirms that the proposed approach has very low overhead and the GPGPU simulation would not be a performance bottleneck in the full system simulation unlike the one with conventional GPGPU simulator.

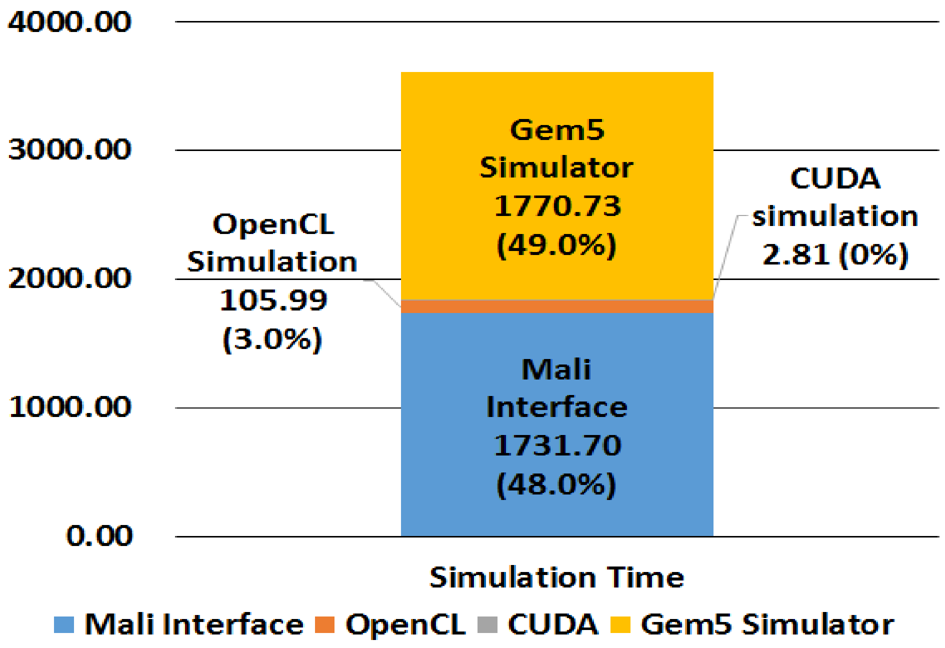


Figure 5-14. Simulation time composition in the GIL simulation

5.5 OpenGL ES Implementation Case

5.5.1 Background

5.5.1.1 Android Graphics Overview

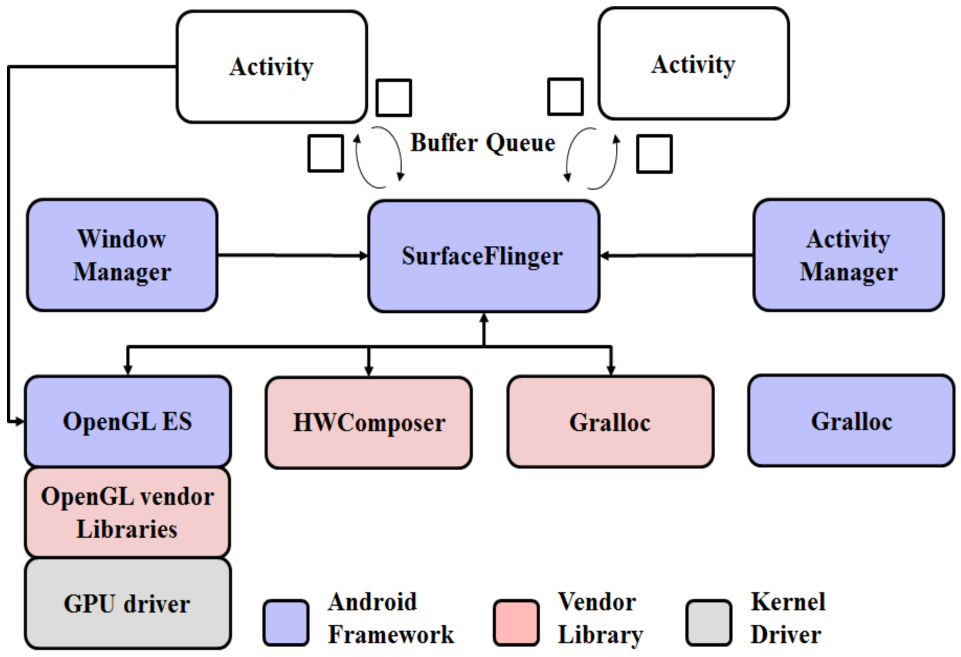


Figure 5-15. Overview for Android Graphics

Figure 5-15 shows an overview for Android Graphics. In Android applications, there are four components including activities, services, content providers and broadcast receivers [47]. Among them, the activity is responsible for the graphics operations and each activity has a window to draw its user interface. Each window has its own surface which has some buffers obtained from SurfaceFlinger process. In the SurfaceFlinger, it only provides an interface for buffer allocation, which is actually performed through a memory allocator called "*gralloc*". In Android, there are two *gralloc* modules: the one is provided by the vendor as a library (so file) and the other is a default module provided in Android and is used when the vendor-specific *gralloc* module is not provided.

After the activity obtains a buffer, the drawing is performed by 2D graphics APIs in Canvas or 3D graphics APIs in OpenGL ES, whose result is rendered onto the obtained buffer in a surface. Then, the drawn buffer is submitted to the SurfaceFlinger process and multiple surfaces are composed based on the window status (visibility, Z-order, alpha value, etc) received from the Window Manager. To compose the surfaces, OpenGL ES library is used in the SurfaceFlinger and much of the composition work can be delegated to the HW composer to offload some work to the GPU.

In Android, the OpenGL ES library consists of platform-independent and dependent layers. The platform dependent OpenGL ES library is provided in Android source code and it simply calls down to the vendor specific libraries in most cases. During the initialization process in this library, the vendor specific OpenGL ES library is loaded dynamically based on the configuration file (*egl.cfg*) which contains a tag like *mali* or *adreno*, which is used to construct the name of the vendor specific library. The vendor specific library provides an interface to the GPU and is responsible for managing the work for the GPU. To allocate rendering work to the GPU, a group of *ioctl* system calls is invoked in the library and the GPU device driver in Linux kernel accesses the registers and memories in the GPU to allocate the work.

5.5.2 Additional modification for SW stack

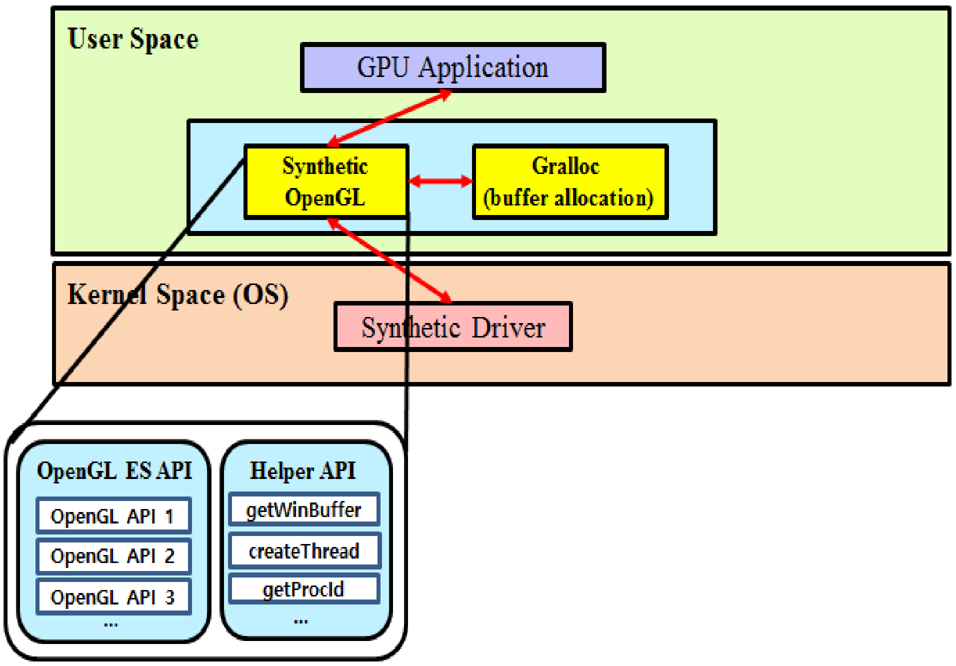


Figure 5-16. Modification for Software stack in OpenGL ES API

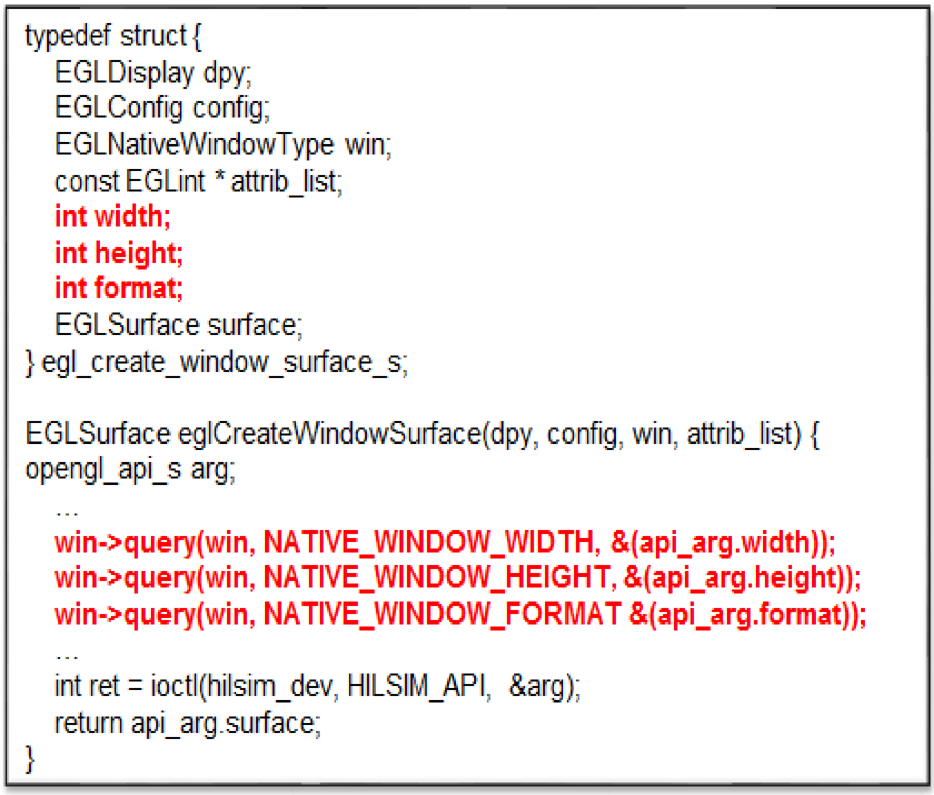
In OpenGL ES API, to consider the memory synchronization for complex data structures (*native window* and *native window buffer*) and multi-process support, additional modification for SW stack is required as shown in Figure 5-16. Helper APIs are added in the synthetic library to assist OpenGL API-level GIL simulation to obtain the process information for multi-process support and the necessary information for the memory synchronization. Since some information for memory synchronization can be obtained from the *gralloc*, original *gralloc* module is modified to invoke Helper APIs to pass the information.

5.5.3 Memory synchronization

In section 5.3, we explained how the memory synchronization is handled in the API level GIL simulation. For GPGPU APIs, since the memory region to be synchronized is fully specified with the API parameters (the start address and the size of the region), the memory synchronization is not difficult. However, in OpenGL ES API, some complex data structures such as *native window* and *native window buffer* are used in the OpenGL ES APIs and only the address for these data is provided when the APIs are called. Thus, it is far from straightforward to keep these data structures in a separate GPU board consistently with the simulator.

5.5.3.1 Native Window

The native window is a C/C++ class that corresponds to the surface in the Java application. To render image from the OpenGL ES graphics application, it requires a native window to get the buffers for rendering. So, when the application requests for the native window, it is created by the Non-OpenGL code in SurfaceFlinger and passed to the application. Then, in the application, *eglCreateWindowSurafce* OpenGL ES API is invoked with the handle for the native window to create an EGL Surface which extends the native window with auxiliary buffers. When this OpenGL ES API is called, it is executed in the board and the native window is accessed with the handle that has been given as an API parameter. However, since the handle is created by the Non-OpenGL code in SurfaceFlinger, its data is located in the simulator memory and invalid access error would occur when the API is



```
typedef struct {
   EGLDisplay dpy;
   EGLConfig config;
   EGLNativeWindowType win;
   const EGLint * attrib_list;
   int width;
   int height;
   int format;
   EGLSurface surface;
} egl_create_window_surface_s;

EGLSurface eglCreateWindowSurface(dpy, config, win, attrib_list) {
opengl_api_s arg;
   ...
   win->query(win, NATIVE_WINDOW_WIDTH, &(api_arg.width));
   win->query(win, NATIVE_WINDOW_HEIGHT, &(api_arg.height));
   win->query(win, NATIVE_WINDOW_FORMAT &(api_arg.format));
   ...
   int ret = ioctl(hilsim_dev, HILSIM_API,  &arg);
   return api_arg.surface;
}
```

Figure 5-17. Code extension for the native window in synthetic library

executed in the board.

To solve this problem, the corresponding native window should be created in the board using the same properties like width, height and format. For this, these properties are extracted from the simulator memory in the synthetic library as shown in Figure 5-17. Then, these properties will be sent to the board with the *eglCreateWindowSurface* API request and the corresponding native window will be created in the board. Even if the properties are obtained, it is not possible to create a native window in the board, since the Android does not provide a library to create a native window from an application. For this reason, the original Android software stack executed in the board should be extended to build the library that provides an interface for creating a native window.

5.5.3.2 Native Window Buffer

In the graphics applications, images can be drawn in two ways: 2D graphics APIs in Canvas and 3D graphics APIs in OpenGL ES. When the drawn buffers are submitted to the SurfaceFlinger process, they are composed using OpenGL ES APIs in the SurfaceFlinger process. Since the native window buffer is not directly usable by the OpenGL ES, it is extended to a general image object called EGL Image by calling *eglCreateImageKHR* API in the SurfaceFlinger process. The problem happens when the native buffers drawn by 2D graphics APIs are composed. Since the buffer is drawn by the Non-OpenGL graphics APIs, the drawn buffer is located in the simulator memory. When *eglCreateImageKHR* OpenGL ES API is executed in the board, the native window buffer will be accessed using the handle given as a parameter. However, since the handle points to the simulator memory, invalid access error will occur during the API execution. Thus, the corresponding native buffer should be created in the board just like the native window. To obtain the properties for a native window buffer, the *gralloc* module, which is responsible for the actual buffer allocation in Android, is modified.

Figure 5-18 illustrates the modified *gralloc_alloc* function in *gralloc* module to obtain the properties required for the native window buffer creation (width, height and format). Then, *getWindowBuffer* Helper API is called to pass these properties to the board and the corresponding native window buffer will be created.

For the case of the native window buffer, in addition to creating it with the same properties, the content in the buffer should be correctly synchronized by copying the pixel data from the simulator memory to the board memory before the OpenGL ES APIs for the composition are executed in the board. It is not sufficient to synchronize the native window buffer when *eglCreateImage* API is executed because an EGL Image is created only once when the buffer is submitted first and *eglCreateImage* API is not be called when the same buffer is re-submitted. Instead, *EGLImageTargetTexture2DOES* API is invoked to generate the texture arrays from the EGL Image whenever the buffer is submitted. Thus, we synchronize the native window buffer whenever this API is invoked.

After the OpenGL ES APIs for the composition are processed in the board, the simulated SurfaceFlinger process will pass the composited buffer to the display controller in the simulator. Since it is composed by OpenGL ES APIs, however, the result is located in the board memory and, without proper synchronization, wrong images would be displayed in the simulator display. Thus, the pixel data for the composite buffer should be synchronized after *eglSwapBuffer* API is executed in the board, which finalizes the pixel data.

```
typedef struct {
   int width;
   int height;
   int format;
   unsigned int bufferHandle;
} get_window_buffer_;s

unsigned int getWindowBuffer(int width, int height, int format) {
   api_arg.width = width;
   api_arg.height = height;
   api_arg.format = format;
   arg.api_id = GL_GET_WINDOW_BUFFER;
   arg.process_id = getProcId();
   arg.thread_id = getContext();
   arg.api_arg = &(api_arg);
   int ret = ioctl(hilsim_dev, HILSIM_API, &arg);
   return api_arg.bufferHandle;
}
```

```
gralloc_alloc(buffer_handle_t * pHandle, int w, int h, int format) {
   ...
  (pHandle->bufferHandle) = getWindowBuffer(w, h, format);
   ...
}
```

Figure 5-18. Modified code for *gralloc* module

5.5.4 Multi-Process Support

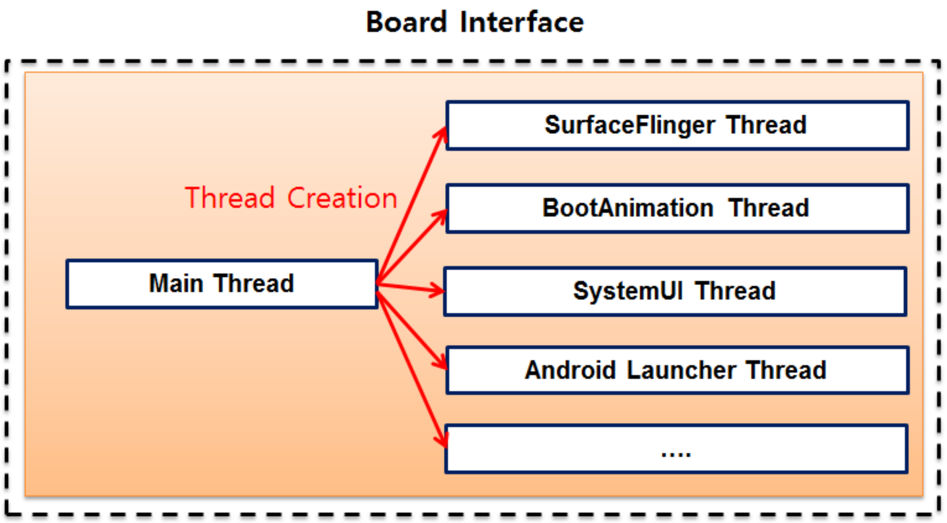


Figure 5-19. Multi-thread structure for Board Interface in OpenGL ES API GIL simulation

In GPGPU API, we assume that only target application uses the API and there is no need to consider the effect between the API call from the different simulated process in the simulator. But in OpenGL ES in Android, there are many system applications which are executed during the boot process and APIs calls from the multiple processes should be correctly handled in the API level GIL simulation.

Generally, in OpenGL ES, each thread has its own OpenGL ES context and the result of API execution is reflected only in the current context. And each context are linked with a specific window and the drawing operations performed by OpenGL

ES APIs are reflected to the window linked to the current thread. However, if the GIL simulation is implemented as a single thread, since only one window can be linked to the thread, the API requests from the multiple threads are reflected in the one specific window, drawing results from the multiple threads are mixed in that window. Therefore, in order to prevent the interference between the API requests from the different threads, multiple threads are created in the board interface and they are processed by separate threads as shown in Figure 5-19.

5.5.4.1 Thread Allocation

When the first OpenGL ES API is requested from the application, *createThreadContext* Helper API is requested first from the synthetic library. Then, the corresponding thread is generated in the board and the thread id is allocated in the host interface. And a client socket is created in the host interface, which is connected with the server socket for the generated thread in the board. The binding information between the thread id and the socket is reserved in the host interface internally, and the thread id is stored in the *thread_id* variable to pass it to the synthetic library. In the synthetic library, the thread id is stored in a per-thread data structure for later use.

5.5.4.2 Target Thread Selection

When the OpenGL ES API is requested, in the synthetic library, *thread_id* variable is set by calling *getThreadId()* function in which thread id is restored from per-thread data structure as shown in Figure 5-2. In the host interface, to find out which thread is responsible for simulating the requested API, the binding information which was previously reserved by the thread allocation process is searched to establish the socket connection. Since thread information is passed by the *thread_id* and *process_id* variables in the common structure, the target socket connection can be obtained using these values from the binding information.

5.5.5 High-level Timing Modeling for other GPUs

Since a real GPU HW is used in the simulation, the proposed technique can simulate the target platforms only if the existing GPU is re-used. For this reason, the design space exploration for the GPU can be performed by using several GPU boards with different GPUs. To overcome this limitation, we proposed a simple analytical timing modeling technique that estimates the execution time of OpenGL API when a non-existing GPU is included in the target platform.

Let CPU/GPU frequency be represented by $(\mathrm{F}_c,\ \mathrm{F}_g)$. In our simple timing model, the API execution time for given CPU/GPU frequency level $(\mathrm{F}_c,\ \mathrm{F}_g)$ consists of three parts: Idle time (T_{IDLE}), CPU execution time ($T_{CPU}^{\mathrm{F}_c}$), and GPU execution time ($T_{GPU}^{\mathrm{F}_g}$) as shown in below.

$$T_{API}(\mathrm{F}_c, \mathrm{F}_g) = T_{IDLE} + T_{CPU}^{\mathrm{F}_c} + T_{GPU}^{\mathrm{F}_g} \qquad (1)$$

The execution time of the CPU and GPU can be calculated by multiplying the total number of cycle ($CPU\_Cycles^{\mathrm{F}_c}$, $GPU\_Cycles^{\mathrm{F}_g}$) and the clock cycle time (inverse of the clock frequency) of the CPU and the GPU.

$$T_{CPU}^{\mathrm{F}_c} = \frac{CPU\_Cycles^{\mathrm{F}_c}}{\mathrm{F}_c} \quad (2)$$

$$T_{GPU}^{\mathrm{F}_g} = \frac{GPU\_Cycles^{\mathrm{F}_g}}{\mathrm{F}_g} \quad (3)$$

In our simple timing model, only the clock frequencies of CPU and GPU can be configured in the target platform and other architecture features are same with GPU board. Moreover, the same code is executed on the target platform so that the total number of cycles of the CPU and the GPU would be remained unchanged regardless of the variation in the clock frequencies of the CPU and the GPU. Let the tuple (F'_c, F'_g) represent the frequencies of the CPU and GPU in the target platform and the tuple (F_c, F_g) represent the frequencies in the GPU board used in the simulation. Then, since the total number of cycles of the CPU and the GPU are not changed depending on the clock frequencies, below equations can be obtained using Equation (2), (3).

$$T_{CPU}^{\mathrm{F}'_c} = T_{CPU}^{\mathrm{F}_c} * \frac{\mathrm{F}_c}{\mathrm{F}'_c}$$

$$T_{GPU}^{\mathrm{F}'_g} = T_{GPU}^{\mathrm{F}_g} * \frac{\mathrm{F}_g}{\mathrm{F}'_g}$$

From these equations, we can know that the execution times of CPU and GPU on the target platform ($T_{CPU}^{F'_c}$, $T_{GPU}^{F'_g}$) can calculated from the execution times measured in the GPU board. And these information can be obtained from the GPU board using the performance analyzer or the profiler such as profiler such as ARM StreamLine [1] during the simulation.

As shown in Equation (1), to model the API execution time in the target platform, we also need to know the idle time in the target platform (T_{IDLE}). For simplicity, we assume that the idle time is not changed depending on the clock frequencies. Thus, we can re-use the idle time obtained from the GPU board. Finally, we can obtain the following equation to model the API execution time in the target platform varying the clock frequencies of the CPU and GPU based on the profile information obtained from the GPU board.

$$\begin{aligned} T_{API}(F'_c, F'_g) &= T_{IDLE} + T_{CPU}^{F'_c} + T_{GPU}^{F'_g} \\ &= T_{IDLE} + T_{CPU}^{F_c} * \frac{F_c}{F'_c} + T_{GPU}^{F_g} * \frac{F_g}{F'_g} \end{aligned}$$

5.5.6 Porting To a New GPU Board

Since the CPU-GPU interface is performed at the OpenGL ES API level which is independent of the GPU, it can be easily extended to support various types of GPUs.

To apply the proposed technique to a new GPU board, it is necessary to modify the parts related with the native window, the native window buffer in the board interface, and the Android OS in the board.

For the native window buffer, memory synchronization part in the board interface should be modified since the structure for the native window buffer is different across the target GPU. Thus, the source code to obtain the pointer to the pixel data should be modified to apply the proposed technique to the new GPU board. This information can be inferred from the header file (*gralloc_priv.h*) of *gralloc* module which is provided by the GPU vendor as a part of the Android source code. Given this file, it takes about a day to modify the source code for the native window buffer.

For the native window, as mentioned in 5.5.3.1, the basic Android does not provide an interface to create the native window from the application. Since the native window is needed from the board interface, the original Android source code should be extended to build the library that provides the interface. For this purpose, the source code for the library in the previous framework is copied to the Android source code for the new GPU board and the protection level for the SurfaceFlinger is lowered so that it can be accessed from the application. Exceptionally, if the way to obtain the native window from the SurfaceFlinger is changed, the library source code from the previous framework may not be applicable. Nevertheless, from our experience, it is not difficult to write the library code to provide an interface for creating the native window.

For the native window, the original Android source code should be extended since the original Android does not provide an interface to create the native window from the application. It has to be extended to provide such an interface. For this purpose, the source code for the interface, or library, in the previous framework should be copied to the Android source code for the new GPU board and the protection level for the SurfaceFlinger process should be lowered so that it can be accessed from the application.

5.5.7 Experiments

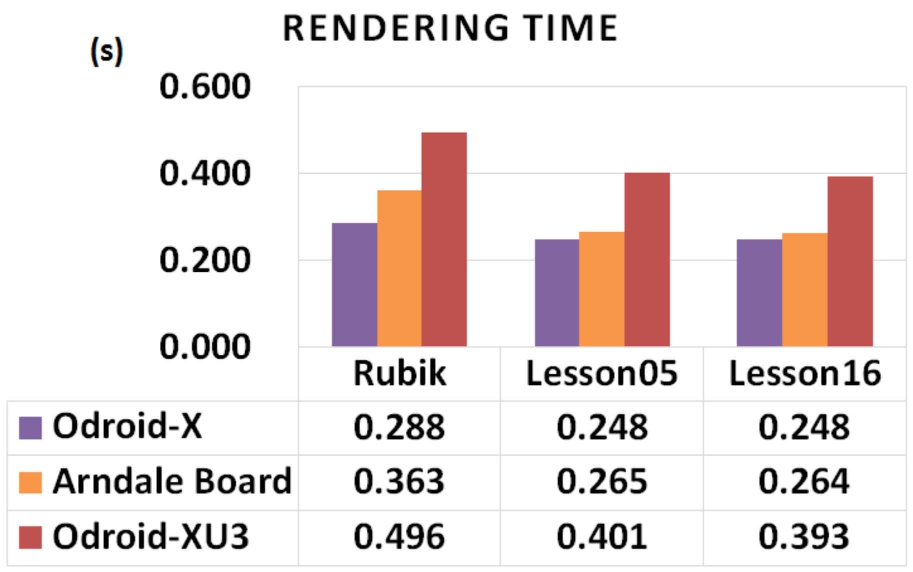
In the proposed simulation framework, the target system is based on the Exynos 4412 system and only the GPU part is changed depending on the development board used in the simulation. We used three development boards (Odroid-X [40], 5250 Arndale [42], and Odroid-XU3 [41]) with different GPUs (Mali 400MP4, Mali T604, and Mali T628 respectively). To make the full system simulation fast enough for software development and verification while sacrificing the timing accuracy, we used the *AtomicSimple* CPU model in gem5.

We ran three Android graphics applications called Rubik [31], Lesson05 and Lesson16 [30]. As explained in 4.4, the Rubik benchmark includes both the computation and the API call sequence, in which the proportions of two parts are 37.4% (computation) and 63.6 % (API call sequence) respectively. However, other benchmarks (Lesson05, Lesson16) only include the API call sequence without any computation. To measure the rendering performance for the benchmarks, we measured only the execution time of the OpenGL API call sequence in the rendering function without computation part, by inserting time stamping code to the application.

It is known that measuring the rendering time for the OpenGL ES application is not a simple problem [33] due to the asynchronous behavior in the OpenGL ES APIs and the window buffer limitation. For asynchronous behavior, if there is no synchronization API at the end of the API sequence, the measured API time only includes the queuing overhead without rendering time. Thus, we inserted the synchronization API at the end of the APIs in the original application to measure the actual rendering time. In Android OpenGL ES application, each application obtains the native window buffer from the buffer queue in the SurfaceFlinger. If there is no free buffer available, the application should wait until the buffer is composed and the measured time can include the waiting time for the buffer. To avoid this problem, we insert a redundant *glClear* API before the first API in each application to guarantee the availability of free buffers.

5.5.7.1 Simulation Speed

The proposed technique can achieves high simulation performance up to about 10 Mcps (cycles per second). Since the *AtomicSimple* model does not simulate the CPU when it becomes idle [32] when the *usleep* function in the synthetic driver is called, gem5 just skips the period without any simulation. In case the application spends most of the time in the GPU, as the benchmark programs used in the experiments do, we can obtain higher GIL simulation speed than the gem5 simulator. Since the performance for the system call-level GIL simulation is about 1 Mcps, the API-level GIL simulation technique can achieve significant speed-up compared with the system call-level GIL simulation.

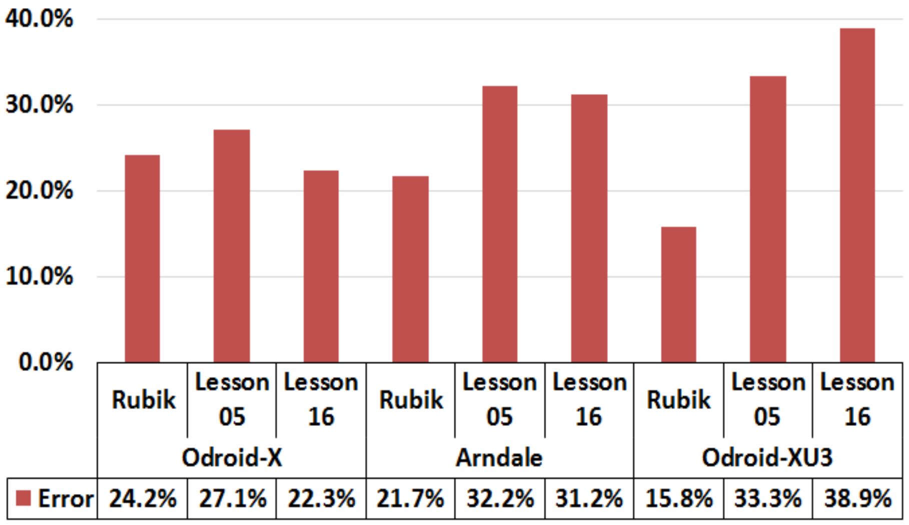


	Rubik	Lesson05	Lesson16
🟪 Odroid-X	0.288	0.248	0.248
🟧 Arndale Board	0.363	0.265	0.264
🟥 Odroid-XU3	0.496	0.401	0.393

Figure 5-20. Rendering times for three benchmarks with three boards

5.5.7.2 Rendering Performance

Figure 5-20 shows the rendering performance (i.e., simulated time) result for three benchmarks when they are simulated using three development boards. In Odroid-X, the execution times for the rendering are 0.248 to 0.288 seconds. As 200 frames are displayed, the rendering performance is about 694.6 ~ 807.6 fps and we can expect that all the benchmarks can display the image to screen without any delay since it is faster than the display rate (60 fps). In other boards, compared to Odroid-X, the execution time is increased by about 1.07 ~ 1.26 and 1.59 ~ 1.72 times for Arndale and Odroid-XU3 board respectively.



	Odroid-X			Arndale			Odroid-XU3		
	Rubik	Lesson 05	Lesson 16	Rubik	Lesson 05	Lesson 16	Rubik	Lesson 05	Lesson 16
🟥 Error	24.2%	27.1%	22.3%	21.7%	32.2%	31.2%	15.8%	33.3%	38.9%

Figure 5-21. Accuracy results for three benchmarks with three boards

5.5.7.3 Accuracy Evaluation

To evaluate the timing accuracy of the proposed framework, we first measured the rendering time of the native execution, and compared with the simulated time from the proposed framework. As shown in Figure 5-21, the timing error is in range of 15.8 % ~ 38.9 %. From the result, we can know that there exist some timing errors compared with the native execution. As mentioned in section 5.2.1, to reduce these timing errors, the simple linear regression analysis is performed based on the rendering time results from two benchmarks: Rubik and Lesson05. From the result shown in Figure 5-22, the ratio factor of 1.232 is obtained. Then, the simulation for Lesson16 benchmark is performed, dividing the measured API time by this factor. Figure 5-23 shows the timing error before and after the ratio factor is applied to the simulation result. From the result, we can know that the average error ratio is decreased from the 30.8 % to 10.8 %. It is confirmed that the proposed approach with regression-based timing modeling provides good timing accuracy with about 10% of timing error for the graphics benchmark examples.

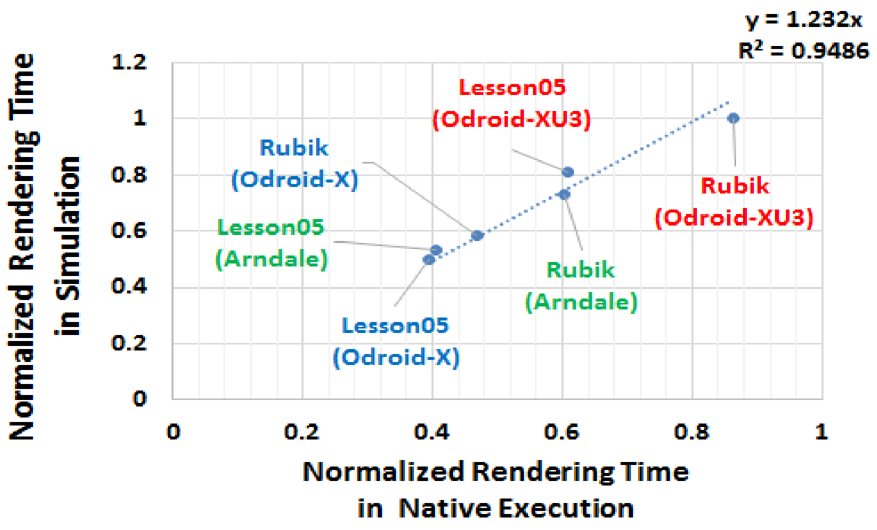
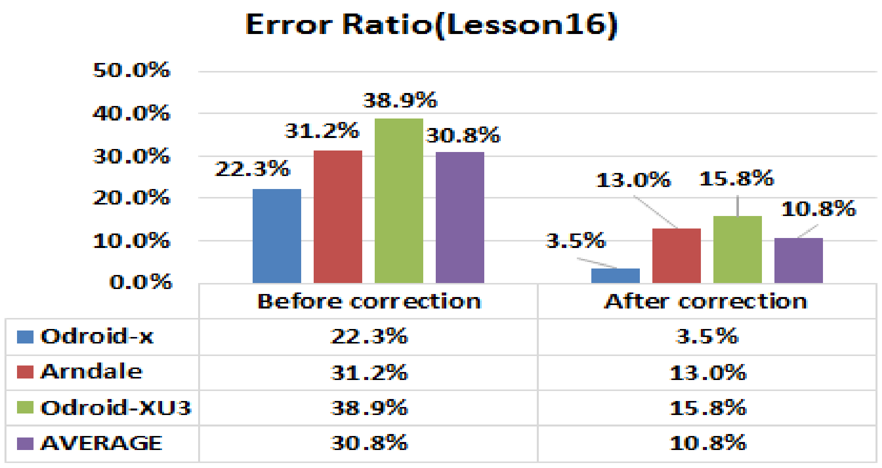


Figure 5-22. Linear regression analysis result for two benchmarks



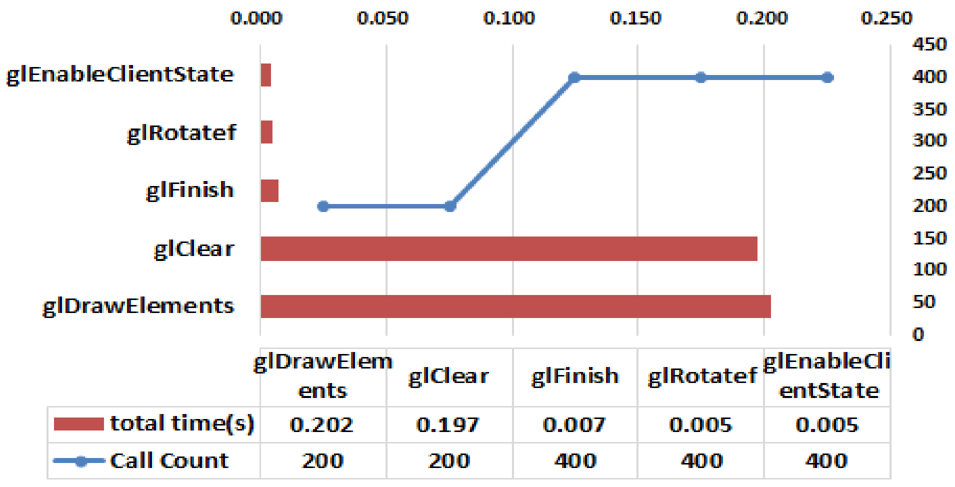
	Before correction	After correction
🟦 Odroid-x	22.3%	3.5%
🟥 Arndale	31.2%	13.0%
🟩 Odroid-XU3	38.9%	15.8%
🟪 AVERAGE	30.8%	10.8%

Figure 5-23. The error ratio before and after the correction factor is applied

5.5.7.4 Dynamic Behavior Profiling

To efficiently optimize the rendering performance, it is important to figure out which API has the longest execution time. Since the OpenGL ES APIs are executed asynchronously in the board, the measured API time during the simulation only includes the queueing overhead without the actual execution time. To obtain the execution time information for actual rendering, synchronous model is implemented in which a synchronization API such as *glFinish* is appended to each API to enforce waiting until the actual execution of the API is completed. Figure 5-24 shows the detailed performance profile information for the Rubik benchmark when the simulation is performed with Odroid-X board in the synchronous model. The information includes the number of API calls and the total execution time for the

APIs. From the results, we observe that *glDrawElements* and *glClear* API take the longest execution times which are 0.202 and 0.197 seconds, respectively, for 200 executions.



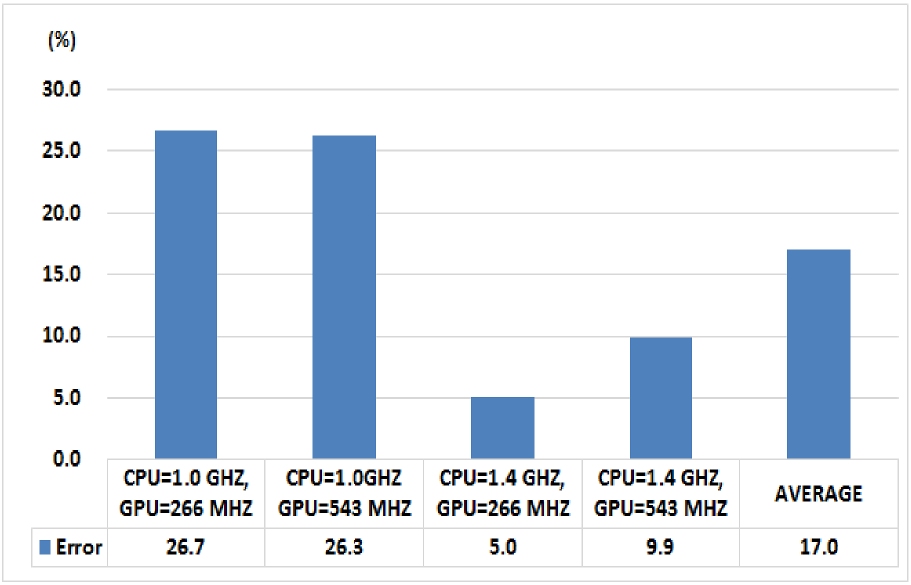
	glDrawElements	glClear	glFinish	glRotatef	glEnableClientState
🟥 total time(s)	0.202	0.197	0.007	0.005	0.005
● Call Count	200	200	400	400	400

Figure 5-24. The total execution time and the call count for each API

5.5.7.5 Design Exploartion varying CPU/GPU frequencies

To overcome the limitation in the proposed technique, a simple timing modeling technique that can model the timing of OpenGL API varying the frequencies of the CPU and the GPU is proposed in 5.5.5. To evaluate timing accuracy of this modeling technique, we first measured the rendering times of the native execution for 4 different CPU/GPU frequency combinations as shown in Figure 5-25. In the simulation, first the simulation is performed by fixing the CPU/GPU frequencies to 1.0 GHZ and 266 MHZ respectively and the profile information required in the modeling technique is obtained using the ARM streamline performance analyzer [1].

After that, the rendering times for the other three configurations are estimated by the proposed modeling technique based on the profile information and the estimated API time is divided by the regression factor obtained in 5.5.7.3, which is 1.232. As shown in Figure 5-25, the timing error is in range of 5.0 % ~ 26.3 % and the average error is 17.0 %.

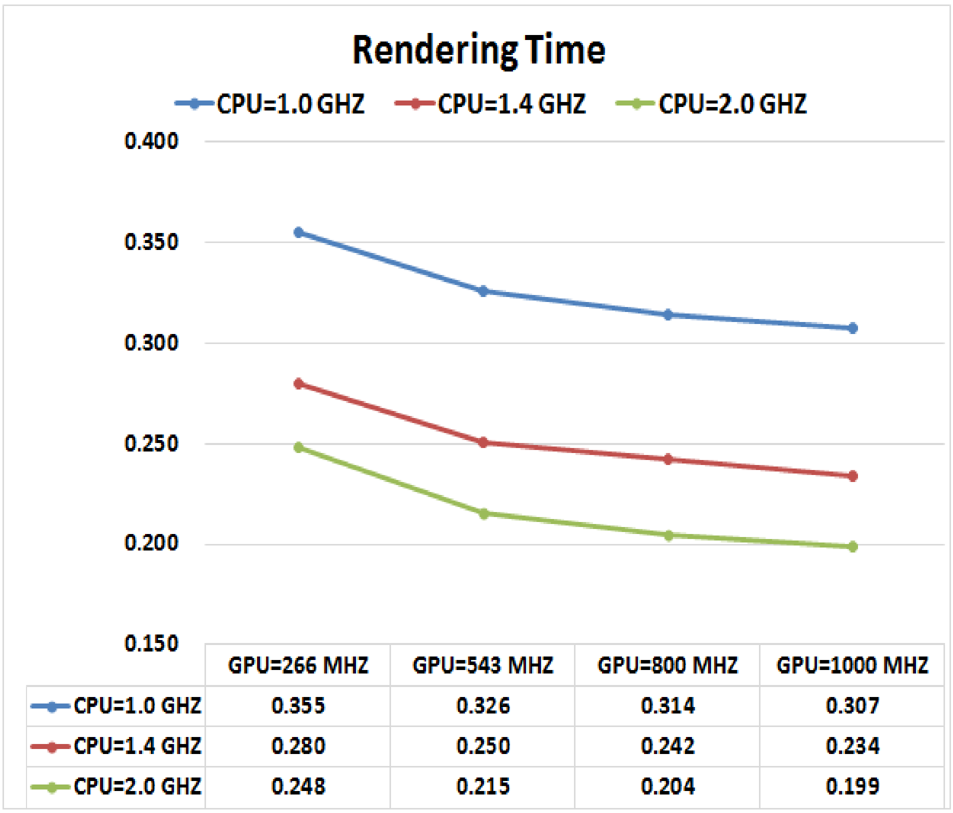


	CPU=1.0 GHZ, GPU=266 MHZ	CPU=1.0GHZ GPU=543 MHZ	CPU=1.4 GHZ, GPU=266 MHZ	CPU=1.4 GHZ, GPU=543 MHZ	AVERAGE
■ Error	26.7	26.3	5.0	9.9	17.0

Figure 5-25. The error ratio for Lesson16 benchmark with 4 combinations of CPU/GPU frequencies on the Odroid-XU3 board

In the Odroid-XU3 board, 5 CPU clock frequencies (1.0, 1.1, 1.2, 1.3 and 1.4 GHZ) and 6 GPU clock frequencies (177, 266, 350, 420, 480, 543) are supported. To show that the proposed technique can be used for design space exploration, additional CPU/GPU frequencies which are not available in the Odroid-XU3 board are considered in the proposed modeling technique. From the result shown in Figure

5-26, in the highest CPU/GPU frequencies (CPU=2.0 GHz, GPU = 1000 MHZ), about 1.26 times higher performance (0.199 seconds) can be achieved compared with the highest performance (0.250 seconds) in the Odroid-XU3 board (CPU=1.4 GHZ, GPU=543 MHZ).



	GPU=266 MHZ	GPU=543 MHZ	GPU=800 MHZ	GPU=1000 MHZ
CPU=1.0 GHZ	0.355	0.326	0.314	0.307
CPU=1.4 GHZ	0.280	0.250	0.242	0.234
CPU=2.0 GHZ	0.248	0.215	0.204	0.199

Figure 5-26. The rendering time for Lesson16 benchmark on the odroid-xu3 board varying CPU/GPU frequencies

5.6 Summary

In this section, we have proposed a fast and extensible GPU-in-the-loop simulation technique that integrates a real GPU hardware with the full system simulator at the API level to make a best compromise between the simulation speed and the timing accuracy. To provide an easily extensible interfacing mechanism between the simulator and the GPU board, a synthetic library is defined for the GIL simulation.

In the GPGPU API-level GIL simulation technique, we simulated a real-life example of face detection applications which both utilize CPU and GPU. Through the simulation, we could estimate the execution time of the face detection. The results show that GPGPU version can increase the performance compared to the CPU only version by 5.7X ~ 10.5X for CUDA version, and 1.29X ~ 1.56X faster for OpenCL version, depending on the input image size. We could also confirm that the proposed approach can easily adopt a new GPGPU in the GIL simulation. We used two different types of GPGPUs, GTS450 and GTX480, without any modification in the simulation framework. From the simulation time profiling, only 4.0 % of the total time is spent on the GPGPU simulation.

In OpenGL ES API-level GIL simulation technique, the proposed technique achieved up to about 10 Mcps, which is 10 times of speedup for Android graphics benchmark applications compared to the system call-level GIL simulation technique. We could apply the proposed technique successfully to three development boards with a little modification of the board interface and Android source code in the board for the native window management. The most challenging problem in the proposed framework is to synchronize two distinct memories in the GPU board and the simulation host. We proposed a novel method to keep the native window and the native window buffers consistent. For timing accuracy, we propose a simple linear regression analysis to compensate the difference between the measured execution time and the simulated time, without the detailed information for the OpenGL ES driver code. Moreover, for the design space exploration varying CPU/GPU frequencies, a simple timing modeling technique is proposed to model the timing of the API execution GPU platforms with the frequencies not supported in the existing GPU.

Chapter 6 Conclusion and Future Work

Emerging mobile devices are likely to adopt CPU-GPU heterogeneous architecture where an embedded GPU executes offloaded computations from the CPU as well as rendering tasks. Thus, building a full system simulator for a CPU/GPU heterogeneous architecture recently draws keen attention of mobile device developers for design space exploration or SW development at the early design stage.

For these purposes, since it is very desirable to run the same application software on a full system simulator, simulation performance is really important. However, all known GPU simulators are mainly developed for architectural exploration, those simulators are prohibitively slow. Moreover, for some mobile GPUs such as Mali or PowerVR, since GPU simulators does not exist, it is impossible to build a full system simulator for the target platforms consisting of these GPUs.

To solve these problems, this thesis propose a GPU-in-the-loop (GIL) simulation technique, which integrates a real GPU Hardware with a full system simulator. Since real HW is used, it can provide fast simulation speed enough for SW development purpose and can build a full system simulator even if a GPU simulator is not available.

There are two major challenges in the proposed technique. First, since the on-chip shared memory in the target system is modeled with the two separate memories in the simulator and the board, we must synchronize the duplicated shared memory models to maintain the coherence. Second, since the detailed behavior of the GPU cannot be observed in the board, it is not easy to model the timing of the GPU in the proposed technique. To handle these challenges, two novel interfacing techniques that interact with a real GPU at system call and API level are proposed in this thesis.

In the system call-level GIL simulation technique, since GPU virtual address is used in the system call argument, address translation table is maintained for memory synchronization. And to model the GPU execution, the interrupt handing mechanism is modeled.

In API-level GIL simulation technique, to provide an easily extensible interfacing mechanism between the simulator and the board, a synthetic library is defined and original SW stack is modified not to simulate the device driver. Since the device driver in the original SW stack is not simulated, interrupt-based timing modeling technique can't be applied. Instead, the API execution is modeled by simply calling a sleep function in the synthetic driver. Since the type of API varies depending on the GPU functionality, two types of APIs for GPGPU and Graphic are considered in this thesis and several API-specific challenges such as asynchronous behavior modeling and memory synchronization for complex data structures are properly handled.

Since the two interfacing techniques have different features, depending on the purpose of the simulation, the suitable interfacing level may be different. To monitor the internal behaviors of the GPU device driver or the API library, the system-call level simulation technique is proper since the device driver and the API library are actually simulated. However, if we are only interested in the high-level performance information on the application such as API-execution time, the API-level simulation technique is proper since it can provide faster and extensible simulation than the system-call level technique.

From the experimental results, we can confirm that the proposed technique successfully make a best compromise between the simulation the timing accuracy so that it can be used for early SW development and system performance estimation.

In the proposed technique, it has a limitation that it can only model the target platform with existing GPU. To overcome this limitation, we proposed a simple timing modeling technique which models the frequencies that is not supported in existing GPU using the profiling tool. Since current model is too simple and only clock frequency can be configured, more sophisticated model is required to consider other high-level architecture characteristics of the GPU such as the number of GPU cores and this is left as a future work.

Also, to verify the effectiveness of current GPU power governors, the current GIL simulation technique will be extended to model the power as well as the performance. In the current experiment results, a small number of benchmarks and GPUs are used. Thus, to faithfully verify the proposed technique, we will perform the experiments using more benchmarks and various types of GPUs in the future work.

Bibliography

[1] Streamline Performance Analyzer, http://www.arm.com/products/tools/software-tools/ds-5/streamline.php

[2] PVRTune, http://community.imgtec.com/developers/powervr/tools/pvrtune/

[3] Snapdragon Profiler, https://developer.qualcomm.com/software/snapdragon-profiler

[4] V. M. del Barrio, C. Gonzalez, A. Fernandez, R. Espasa, "ATTILA: A Cycle-Level Execution-Driven Simulator for Modern GPU Architectures" IEEE International Symposium on Performance Analysis of System and Software (ISPASS), pp. 10-12, Mar. 2006.

[5] Wang, P. H., Lo, C. W., Yang, C. L., & Cheng, Y. J. (2012, April). A cycle-level SIMT-GPU simulation framework. In Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on (pp. 114-115). IEEE.

[6] Bakhoda, G. Yuan, W. Fung, H. Wong, T. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator", in IEEE

International Symposium on Performance Analysis of Systems and Software, Apr. 2009.

[7] Collange, S., Daumas, M., Defour, D., & Parello, D. (2010, August). Barra: A parallel functional simulator for gpgpu. In Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on (pp. 351-360). IEEE.

[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. GILl, and D. A. Wood, "The gem5 simulator," ACM SIGARCH Computer Architecture News, vol. 39, no. 2, 2011.

[9] Fabrice Bellard, "QEMU, a fast and portable dynamic translator", USENIX Annual Technical Conference, 2005.

[10] R. Ubal, B. Jang, P. Mistry, D. Schaa, D. Kaeli, "Multi2Sim: A Simulation Framework for CPU-GPU Computing", Parallel Architectures and Compilation Techniques, Sep. 2012.

[11] Kim, Hyesoon, et al. "MacSim: a CPU-GPU heterogeneous simulation framework user guide." "Georgia Institute of Technology (2012).

[12] Zakharenko, V., Aamodt, T., & Moshovos, A. (2013, March). Characterizing the performance benefits of fused CPU/GPU systems using FusionSim. In Proceedings of the Conference on Design, Automation and Test in Europe (pp. 685-688). EDA Consortium.

[13] H. Wang, V. Sathish, R. Singh, M. Schulte, N. Kim, “Workload and Power Budget Partitioning for Single Chip Heterogeneous Processors,” IEEE Conf. on Parallel Architecture and Compilation Techniques, Sep. 2012.

[14] J. Power, J. Hestness and M. S. Orr, “gem5-gpu: A Heterogeneous CPU-GPU Simulator”, IEEE Computer Architecture Letters, DOI 10.1109/LCA.2014.2299539

[15] J Ma, J., Yu, L., John, M. Y., & Chen, T. (2015). MCMG simulator: A unified simulation framework for CPU and graphic GPU. Journal of Computer and System Sciences, 81(1), 57-71.

[16] S. Shen, S.Lee, C. Chen, “Full System Simulation with QEMU: an Approach to Multi-view 3D GPU Design”, Circuits and Systems (ISCAS), May, 2010.

[17] S. Raghav, C. Pinto, , M. Ruggiero, A. Marongiu, D. Atienza, L. Benini, “Full System Simulation of Many-Core Heterogeneous SoCs using GPU and QEMU Semihosting,” in Proceedings of the 5th Annual Workshop on General-Purpose Processing with Graphics Processing Units (GPGPU-5), pp. 101-109, Mar. 2012

[18] Arnau, Jose-Maria, Joan-Manuel Parcerisa, and Polychronis Xekalakis. "TEAPOT: a toolset for evaluating performance, power and image quality on mobile graphics systems." Proceedings of the 27th international ACM conference on International conference on supercomputing. ACM, 2013.

[19] Li, Sheng, et al. "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures." Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on. IEEE, 2009.

[20] H. Kim, D. Yun, S. Ha, “Scalable and Retargetable Simulation Techniques for Multiprocessor Systems”, CODES+ISSS pp 89-98 Oct, 2009.

[21] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald III, N. Beckmann, C. Celio, and A. Agarwal, “Graphite: A distributed parallel simulator for multicores”, In Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA), pp. 1–12, Jan. 2010.

[22] S. Lee, W. Ro, “Parallel GPU architecture simulation framework exploiting work allocation unit parallelism”, IEEE International Symposium on Performance Analysis of System and Software (ISPASS), 2013

[23] Lee, Sangpil, and Won Woo Ro. "Parallel GPU Architecture Simulation Framework Exploiting Architectural-Level Parallelism with Timing Error Prediction.", IEEE Transactions on Computers, 10.1109/TC.2015.2444848

[24] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically characterizing large scale program behavior”. In ASPLOS, 2002.

[25] T. E. Carlson, W. Heirman, and L. Eeckhout. “Sampled simulation of multi-threaded applications” In ISPASS, 2013

[26] J. Huang, L. Nai, H. Kim and H. Lee, “TBPoint: Reducing Simulation Time for Large-Scale GPGPU Kernels”, IEEE International Conference on Parall and Distributed Processing Symposium (IPDPS), pp 437-446, May. 2014.

[27] Z. Yu, et al. “GPGPU-MiniBench: Accelerating GPGPU Micro-Architecture Simulation”, IEEE Transactions on Computers, doi. 10.1109/TC.2015.2395427

[28] K. Fang, Y. Ni, J. He, Z. Li, S. Mu, Y. Deng, “FastLanes: An FPGA Accelerated GPU Micro-architecture Simulator”, IEEE International Conference on Computer Design (ICCD), pp. 241-248, Oct. 2013.

[29] GPUOcelot, http://gpuocelot.gatech.edu/

[30] Lesson05 – 3D shapes, Lesson 09 – Moving bitmaps in 3D space, Lesson16 – Cool Looking Fog http://insanitydesign.com/wp/projects/nehe-android-ports/

[31] Rubik Cube Animation Example In Android, http://www.edumobile.org/android/rubik-cube-animation-example-in-android/

[32] http://comments.gmane.org/gmane.comp.emulators.m5.users/11513

[33] http://blog.imgtec.com/powervr/micro-benchmark-your-render-on-powervr-series5-series5xt-and-series6-gpus

[34] Pin. A Binary Instrumentation Tool. http://www.pintool.org.

[35] Yourst, Matt T. "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator." Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on. IEEE, 2007.

[36] MARSSx86: Micro-ARchitectural and System Simulator for x86 based Systems. www.marss86.org

[37] Cain, Harold W., et al. "Precise and accurate processor simulation." Workshop on Computer Architecture Evaluation using Commercial Workloads, HPCA. Vol. 8. 2002.

[38] Mali-400 MP, http://www.arm.com/products/multimedia/mali-cost-efficient-graphics/mali-400-mp.php

[39] Samsung Exynos 4 Quad (Exynos4412), http://www.samsung.com/global/business/semiconductor/file/product/Exynos_4_Quad_User_Manaul_Public_REV1.00-0.pdf

[40] Odroid-X board, http://www.hardkernel.com/main/products/prdt_info.php?g_code=G133999328931

[41] Odroid-XU3 Board, http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127

[42] Arndale Board, http://www.arndaleboard.org/wiki/index.php/Main_Page

[43] OpenCV, http://opencv.org/

[44] SystemC, http://accellera.org/downloads/standards/systemc

[45] AtomicSimpleCPU model in gem5, http://www.m5sim.org/SimpleCPU

[46] ARM Semihosting Interface, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0471g/Bgbjjgij.html

[47] http://developer.Android.com/guide/components/fundamentals.html#Components

초록

복잡한 3D 게임을 처리하거나, 높은 반응성을 가지는 유저인터페이스를 제공하기 위해서, 대부분의 임베디드 시스템에서 모바일 GPU 가 사용되고 있다. 게다가, 모바일 GPU 의 계산 능력이 높아지고, GPU 에 대한 프로그래밍이 가능해짐에 따라, 모바일 GPU 가 하나의 보조 연산 장치로서 여겨지고 있다. 모바일 GPU 의 경우, 서버 환경과 달리 제약된 파워 상에서 수행되어야 하므로, 대게 적은 수의 코어를 포함한다. 그러므로, 주어진 성능과 파워 제약 조건을 만족시키기 위해서는 CPU 와 GPU 모두를 효율적으로 활용하는 것이 매우 중요하다.

CPU/GPU 이종 병렬 아키텍쳐를 설계하는 초기 단계에서 SW 에 대한 오류를 검출하거나 또는 다양한 설계 공간 탐색을 위해서, 가상 프로토타이핑 시스템을 사용하는 것이 일반적이다. 가상 프로토타이핑 시스템에서는 대상하는 시스템의 모든 구성요소에 대한 시뮬레이션 모델을 포함하므로, CPU 와 GPU 가 포함되는 이종 병렬 아키텍쳐를 위해서는 GPU 에 대한 시뮬레이션 모델이 반드시 필요하다. 그러나 일부 GPU 의 경우, 시뮬레이션 모델이 존재하지 않고, 있는 경우에도 주로 마이크로 아키텍쳐 수준에서의 아키텍쳐 탐색을 위한 목적으로 개발되어, 시뮬레이션 성능이 좋지 않다.

이러한 문제를 해결하기 위해서, 본 논문에서는 실제 하드웨어와 시뮬레이터를 결합하는 GPU-in-the-loop (GIL) 시뮬레이션 기법을 제안하려고

한다.

제안하는 방법의 경우, 다양한 수준에서 CPU 와 GPU 간의 연동이 가능한데, 첫번째 방법으로 시스템 콜 수준에서 시뮬레이터와 GPU 보드 간의 연동하는 기법을 제안한다. 제안하는 기법에서는 대상 시스템에 있는 공유 메모리가 시뮬레이터와 보드 상에 존재하는 서로 다른 두개의 메모리를 통해 시뮬레이션이 되므로, 두 메모리 간의 일관성을 유지하기 위한 메모리 동기화가 가장 중요한 문제이다. 시스템 콜 기반 기법에서 이 문제를 다루기 위해서, 주소 변환 테이블을 통해서 공유 되는 메모리 영역에 대한 정보를 저장하고, 실제 보드 상의 GPU 를 수행시키는 System Call 이 요청될 때마다, 해당 테이블을 이용하여 공유 되는 영역에 대한 동기화가 수행된다. GPU 의 수행을 시뮬레이터 상에서 모델링하기 위해, 인터럽트 기반 모델링 기법을 제안하였는데, 이 기법에서는 보드에서 측정된 GPU 수행시간을 고려하여, 시뮬레이터 상에서 GPU 인터럽트를 발생하도록 한다.

두번째 방법으로 API 수준에서 시뮬레이터와 보드 간의 연동하는 기법을 제안한다. 기존 Software Stack 에 포함된 디바이스 드라이버가 시뮬레이션 되는 경우, 다양한 GPU 를 지원하도록 확장하는 것이 어려우므로, API 기반 기법에서는 시뮬레이션 용도로 사용되는 새로운 라이브러리를 정의하고, 기존 SW stack 상에 존재하는 GPU 라이브러리를 대체하도록 하여, 디바이스 드라이버가 시뮬레이션 되지 않도록 한다. 그리고 API 수행시간을 시뮬레이터 상에서 모델링하기 위해서, 시뮬레이션을 위한 새로

운 디바이스 드라이버를 정의하여, 해당 드라이버 내에서 sleep 함수를 호출하여, 보드에서 측정된 API 시간이 시뮬레이터상에 반영되게 된다.

현존하는 GPU API 중에서, 본 논문에서는 가장 많이 사용되는 OpenCL, CUDA 그리고 OpenGL ES API 에 대한 API 기반 시뮬레이션 기법을 제안한다. 그리고 올바른 시뮬레이션을 위해서, 비동기 동작, 멀티프로세스 지원, 복잡한 데이터 구조에 대한 메모리 동기화와 같은 어려운 문제들을 다양한 기법들을 통해 해결하였다.

실험 결과를 통해서, 제안된 기법이 적절한 수준의 정확도를 보장하면서, 빠른 시뮬레이션 성능을 제공할 수 있음을 확인할 수 있다. 그러므로, 제안된 기법은 SW 개발 용도뿐만 아니라, 시스템 수준에서의 성능 예측을 위한 용도로서 사용이 가능하다. 게다가, 제안된 기법의 경우, 실제 하드웨어가 사용되므로, GPU 에 대한 시뮬레이터가 제공되지 않는 경우에도 CPU/GPU 이종 병렬 시스템을 위한 가상 프로토타이핑 시스템을 구축하는 것이 가능하다.

주요어 : CPU/GPU 이종 병렬 플랫폼, GPU 시뮬레이션, 가상 프로토타이핑 시스템, GPU-in-the-loop 시뮬레이션, 시스템 콜, API

학 번 : 2009-20750