HSAemu – A Full System Emulator for HSA Platforms

JIUN-HUNG DING, ZHOUDONG GUO, CHUNG-MING KAO,
National Tsing Hua University
WEI-CHUNG HSU, National Chiao Tung University
YEH-CHING CHUNG, National Tsing Hua University

Heterogeneous system architecture is an open industry standard designed to support a large variety of data-parallel and task-parallel programming models. Many application processor vendors, including AMD, ARM, Imagination, MediaTek, Texas Instrument, Samsung and Qualcomm are members of the HSA Foundation. This paper presents the design of HSAemu, a full system emulator for the HSA platform. The design of HSAemu is based on PQEMU, a parallel version of QEMU, with supports for HSA defined features such as a) shared virtual memory between CPU and GPU, b) memory based signaling and synchronization, c) multiple user level command queues, d) preemptive GPU context switching, and e) concurrent execution of CPU threads and GPU threads, etc. In addition to the basic requirements of the HSA-compliant features, HSAemu also includes an LLVM based binary translation engine to efficiently support translating multiple different ISAs (e.g. ARM and HSAIL — HSA Intermediate Language).

Categories and Subject Descriptors: C.1.3 [Processor Architectures]: Other Architecture Styles—Heterogeneous (hybrid) Systems: C.1.6 [Simulation and Modeling]: Type of Simulation—Parallel
General Terms: Design, Simulation
Additional Key Words and Phrases: HSA, GPU simulation, parallel simulation

1. INTRODUCTION

Over the last decade, there has been a growing interest in the use of graphics processing units (GPUs), originally designed to perform specialized graphics computations in parallel, to perform general purpose parallel computation tasks traditionally handled by the central processing units (CPUs). However, the current design by integrating CPUs and GPUs into a heterogeneous computing platform has several drawbacks. On the hardware side, current CPUs and GPUs have been designed as separate processing elements and do not work together efficiently due to each has a separate memory space. An application is required to explicitly copy data from CPU to GPU and then back again. This introduces significant execution overhead as well as programming complexity. On the software side, a program running on the CPU queues work for the GPU via system calls through a device driver stack managed by a completely separate scheduler. This introduces significant dispatch latency. Further, it is not feasible for a program running on the GPU to directly generate work-items, either for itself or for the CPU. Heterogeneous system architecture (HSA) is an emerging open industry standard, proposed by the HSA foundation, to address the issues mentioned above. The essence of the HSA strategy is to create an improved processor design to support heterogeneous computing that includes a large variety of data-parallel and task-parallel programming models by providing a unified view of fundamental computing elements for a programmer to write applications that seamlessly integrate CPUs with GPUs, while benefiting from the best attributes of each other. This single unified programming platform is a strong foundation for the development of languages, frameworks, and applications of HSA. More specifically, the goals of HSA include

- Remove the CPU/GPU programmability barrier.
- Reduce CPU/GPU communication latency.
- Open the programming platform to a wider range of applications by enabling existing programming models.
- Create a basis for the inclusion of additional processing elements beyond the CPUs and GPUs.

New Paper, Not an Extension of a Conference Paper
To support HSA computing, software development tools are also important. However, what is missing from the current software development tools is an HSA-compliant full system emulator. A full system HSA emulator has three benefits. First, it will help application or runtime system developers to emulate unmodified HSA-compliant applications for functional debugging and testing at early stages before the available hardware. Second, a full system emulator can generate traces of important events for more detailed micro-architecture simulations which are critical for hardware designers to evaluate their designs before tape-out. Third, it can be integrated with other tools for system software analysis since a complete software stack, such as Android, could be run on top of it.

This paper presents the design of a full system HSA emulator, called HSAemu, which follows the specifications of the HSA standard. The goals of HSAemu are

1. Support parallel emulation models for both CPU and GPU computations.
2. An HSA-compatible GPU applications could be seamlessly executed without modification by the capabilities of full-system emulation, HSAIL binary translation, HSA runtime library support, and hUMA emulation.
3. Support configurable emulation components, such as kernel function binary translator and GPU thread scheduling models.

To meet the first goal, we use PQEMU [Ding et al. 2011] for parallel CPU computation emulation. PQEMU is a parallel version of QEMU [Bellard 2005] to enable multi-threaded CPU emulation. It provides efficient synchronization models to reduce emulation overhead. For parallel GPU computation emulation, each GPU compute unit is simulated by a thread, called GPU computation thread. Two kinds of thread schedulers, static and dynamic, are proposed to assign jobs to these GPU computation threads that are running the work items in parallel.

To meet the second goal, HSAemu fully emulates a whole computer system, including multi-core processor, GPU, memory subsystem and I/O subsystem. To support unmodified HSAIL-compatible applications, an LLVM [Lattner et al. 2004] based translator is provided to translate HSAIL binary code to the host binary code. The main purpose of HSA runtime library is to define the communication between CPU and GPU. To support HSA runtime library, we provide an architected queuing language (AQL) queue to dispatch AQL issued by CPU into the GPU. For hUMA emulation, separated soft-MMUs for CPU and GPU are provided to allow both CPU and GPU to share a virtual address space by accessing identified guest page tables.

To meet the third goal, two kinds of configurability are considered in the HSAemu. One is the external binary translator configurability in which the LLVM binary translator is loosely-coupled integrated with HSAemu. The LLVM binary translator can be easily replaced by an alternative translator. The other is the internal GPU module emulation configurability in which the GPU emulation scheduling algorithm can be replaced by another scheduling algorithm.

The design of HSAemu is evaluated on a 32-physical-core Dell Poweredge host machine. Two HSAIL benchmarks, the Nearest Neighbor benchmark (total work item size is 4096 * 4096) and the K-means benchmark (total work item size is 819200) are used for measure the scalability of GPU emulation. The experimental results show that the performance of GPU emulation is scale up when the number of GPU computation thread is less than or equal to the number of physical CPU cores. The results also show that the dynamic scheduling algorithm is superior to the static scheduling algorithm for GPU emulation. Furthermore, the performance of an optimization method that uses the host SIMD instructions for kernel code execution
is evaluated. We have found that K-means benchmark gains a 1.8X to 2.7X speedup and the benchmark Nearest Neighbor only gains 1.02X to 1.05X speedup.

In section 2, the related works of GPU simulation are presented. Section 3 introduces the background of HSA. Then an overall architecture design and implementation of HSAemu will be described in Section 4. Some preliminary experiment results are presented and discussed in section 5.

2. RELATED WORK

This work is related to several research areas, such as binary translation, GPU simulation, and system emulation.

The binary translation techniques, in general, can be divided into two categories, static and dynamic. The static binary translation technique translates source binary to target binary before execution while the dynamic one does it on-the-fly. Binary translation can be used in various areas, such as simulator, web browser, virtual machine, etc. LLVM is a well-known compiler framework for binary translation. For instance, Android Dalvik virtual machine (DVM) uses LLVM to run bytecode on-the-fly by the technique of dynamic binary translation. LLBT [Shen et al. 2012] is a LLVM based static binary translator that translates source binary into LLVM IR and then retargets the LLVM IR to various ISAs by using the LLVM compiler infrastructure. They have shown that their ARM-based LLBT can effectively migrate EEMBC benchmark Suite from ARMv5 to Intel IA32, Intel x64, MIPS, and other ARMs such as ARMv7. In [Perez et al. 2012], the authors developed a method-based JIT compiler based on the LLVM framework that delivers performance improvement comparable to that of an ahead-of-time compiler. They have shown that their method-based JIT is better than a basic trace-based JIT provided by Ardroid, and by sharing profiling and compilation information among each other, a smart combination of both JIT techniques can achieve a great performance gain.

For GPU simulation, several simulators have been proposed in the literature. GPGPU-sim [Bakhoda et al. 2009] provides a detailed simulation of a contemporary GPU running CUDA and OpenCL [Stone et al. 2010] workloads with an integrated energy model at micro-architectural level. It also supports simulation in PTX and NVIDIA native ISA SASS. Barra-sim [Collange et al. 2010] is a functional level GPU simulator based on the UNISIM [August et al. 2007] framework. It simulates CUDA programs at the assembly language level (Tesla ISA) and is highly compatible with NVIDIA G80-based GPUs. Ocelot [Diamos et al. 2010] is a modular dynamic compilation framework for heterogeneous system. It targets to several back-ends with self-developed translator. Ocelot also implemented its own compiler from the IR like PTX to CPU code. AMD FusionSim [Zakharenko et al. 2013] is based on PTLsim [Yourst 2007] and GPGPU-sim to simulate an X86 out-of-order CPU, a CUDA-capable GPU and, a CPU/GPU interconnect memory system. GPGPU-sim and AMD FusionSim both are micro-architectural level simulators while Ocelot and Barra-sim simulate at functional level.

A full-system simulator is an architecture simulator that simulates an electronic system at such a level of detail that complete software stacks from real systems can run on the simulator without any modification. A full system simulator provides virtual hardware that is independent of the nature of the host computer. The full-system model includes processor cores, peripheral devices, memories, buses, and network connections. Architectural simulations include micro-architectural and functional simulations. Examples are SimpleScalar [Austin et al. 2002] and Wattch [Brooks et al. 2000], and ZSim [Sanchez et al. 2013]. SimpleScalar is a well-known micro-architectural simulation for cycle simulations. Wattch is a well-known micro-
architectural simulation for power consumption analysis and optimizations. ZSim includes three novel techniques, speed up detailed core models, bound-weave, and lightweight user-level virtualization, to make thousand-core simulation practical. Functional simulations allow the interactions among processors, memory and peripherals to be observed. Recent functional emulators, such as Embra [Witchel et al. 1996], Mambo [Bohrer et al. 2004], QEMU, Simics [Magnusson et al. 2002], etc., usually equip with dynamic binary translation for increased simulation efficiency. In today's multi-core environment, parallelism exploitation becomes a major issue in emulator designs. Examples are PQEMU, COREMU [Wang et al. 2011], Parallel Mambo [Wang et al. 2008], and Parallel Embra [Lantz 2008]. PQEMU is a parallel version of QEM. It provides unified code cache and separated cache code models to enable multi-threaded CPU emulation and provides efficient synchronization models to reduce emulation overhead. COREMU emulates multiple cores by creating multiple instances of existing sequential emulators, and uses a thin library layer to handle the inter-core and device communication and synchronization. Parallel Mambo is a multi-threaded implementation of Mambo. It proposed a multi-scheduler to adapt Mambo's simulation engine to multi-threaded execution. Parallel Embra uses the round-robin scheduling method to dispatch the execution of guest cores to physical cores of the host machine.

3. PRELIMINARIES
3.1 HSA
In the last couple of decades, mainstream computer systems typically include other processing elements in addition to CPU. GPU, as the most prevalent one, originally designed to perform specialized graphics computations in parallel. After a long time improvement, GPUs have become more powerful and more generalized, and its usage for running general purpose parallel programs has become more and more popular, by using standard programming languages such as OpenCL. For data parallel programs, using GPU often achieves higher execution and power efficiency. However, since CPU and GPU are designed as separate processing elements, so it is hard for them to work seamlessly. For example, each of them has a separate memory space, requiring an application to explicitly copy data back and forth. To fully exploit the capabilities of heterogeneous parallel execution units, the designers re-architect computer systems to tightly integrate the disparate compute elements on a platform while preserving a programming model that software developers are familiar with. This is the primary goal of the new HSA design.

With HSA, applications can create data structures in a single unified address space and initiate work items on the most appropriate hardware for a given task. Sharing data between compute elements is as simple as sending a pointer. Multiple compute tasks can work on the same coherent memory regions, utilizing barriers and atomic memory operations as needed to maintain data synchronization.

HSA also introduces a new portable programming model. It helps to simplify the process of moving applications across different hardware devices. It was hard to ask application vendor to port their software to a new kind of hardware, especially on the proprietary platforms. HSA simply bring hardware to application programmer. Since HSA includes the hardware, interfaces, common intermediate language, and standard runtime components to do all the necessary work. HSA also maintains memory coherency and manages work queues under the hood, without exposing the underlying system complexity to the application developers.

To achieve these benefits, an HSA-compliant device should meet some requirements of the HSA standard, such as a shared virtual memory space, cache coherency domains, memory-based signaling and synchronization, user mode queuing, and so on. Some of them should be strictly followed in our simulator.
Traditional GPU uses a separate memory space from the CPU so that programmers must handle explicit memory movements between CPU memory and GPU memory. Such movements often involve frequent system calls on the bandwidth limited PCIE bus. To minimize such memory movements is complex and significantly reduces the productivity. Furthermore, when the device memory is not large enough to handle the problem size, programmers must divide the work into smaller chunks so as to fit in the device memory.

HSA programming allows programmers not to worry much about explicit management of data copies and data partitioning. In addition, the range of memory that the GPU can access in HSA, is now as wide as the virtual memory space is. Thus simplified the programming on GPU.

3.2 HSAIL

The HSA uses an intermediate representation called the HSAIL (Heterogeneous System Architecture Intermediate Language) to represent an intermediate format of GPGPU computing kernels. The high-level language compiler handles most of the optimization processes. A light-weighted native generator called finalizer will do the back-end code generation. The finalizer aims to translate the HSAIL to different native ISAs in order to be capable running on various devices. To access the special memory model, the HSAIL has the corresponding syntax to access the spill segment for spilling and loading the data to or from register. In the practice of CUDA, spilling is worked as optimization from certain level of analysis.

Barrier syntax is defined as barrier and fine-grained barrier. Barrier is a forced syntax for all working work items to synchronize at a time. Such work may decrease performance in some practices if such synchronization is needed only by a subset of work items. Fine-grained barrier is introduced by HSAIL to synchronize the specified number of work items within a work group. Programmers are allowed to have more flexibility in their GPGPU practices.

The HSAIL consists of 120 operation codes performing arithmetic operations, memory operations, branch operations, image related operations, parallel synchronization operations and function related operations, etc. Supporting vector instructions give chances in generating native SIMD instructions with less analysis. In addition, 4 types of register width which are 1, 32, 64, 128 bits are available. One bit register used as condition code, 32-bit and 64-bit supports both single and double precision floating point data. Besides, 32, 64, 128 bits register can be used as vector registers for various types of vector format such as 8bit x 4, 32bit x 4 and 64bit x2 etc.

The HSAIL has finite register set and no PHI nodes. Finite register set makes register mapping analysis easier. With no PHI nodes, no SSA (Single Statement Assignment) analysis is needed. The LLVM IR contains high-level symbols such as C structure, whereas HSAIL does not. Such features speed up the native code generation of HSAIL finalizer. As a GPGPU IR, LLVM IR fails to provide a thorough parallel processing semantics. HSAIL provides such semantics allowing programmers takes more sophisticated understanding to what they are doing. Data movement with the work groups and lanes are well-defined. Consistency of data in memory is ensured by acquire and release syntax formally. Whenever data is claimed acquired by a certain work group, others are unable to acquire such data until the data is released. Data consistency within work items can be ensured by using atomic instructions.
4. HSAemu

Figure 1 shows the typical components of a heterogeneous system architecture that includes a single chip with a multi-core processor and a graphical processing unit, a memory sub-system and a peripheral I/O. The two computation units communicate with each other by an interconnection protocol. They could access data fetched from shared cache or main memory by using the same virtual address.

![Fig. 1. Heterogeneous System Architecture.](image)

To design HSAemu for such heterogeneous system architecture, several design metrics need to be taken into considerations. The first metric is the emulation speed. We will face the challenges on how to parallelize the multi-core and GPU emulation. How to manage the emulation threads and handle synchronization efficiently are the main issues. The second metric is the HSA-compatible emulation issues, including how to support the HSA intermediate representation, HSAIL, and the specification of runtime library. The benefit of compatibility results in that unmodified HSA-compatible applications could run directly on HSAemu. The third metric is the flexibility issue which means how to make HSAemu more general and configurable to support other binary formats for kernel function running on GPU, such as CUDA code. In addition, the whole emulation architecture could help GPU designers evaluate scheduling models and inner memory architectures.

By taking these metrics into account, the main idea of our design of HSAemu is to emulate an HSA-compatible single chip which seamlessly integrates multi-core processors and GPUs to work together concurrently. Traditionally, GPU module is considered as a peripheral device on a PCI bus. In our design, GPU is controlled directly by CPU through an on-chip interconnection protocol. In addition, CPU and GPU have the same memory address space to facilitate data sharing and communication. Both CPU and GPU can work concurrently and run in parallel to achieve high emulation speed. To satisfy the requirements of HSA standard, HSAemu has two external components, LLVM HSAIL translator and HSAemu runtime library, which are used to support HSA-compatible applications. While running a HSAIL kernel binary, GPU module has three components including GPU Thread Monitor (GTM), GPU Translation Engine (GTE) and GPU Execution Engine (GEE). Fig. 2 shows the system architecture of HSAemu that consists of a CPU simulation module, a GPU simulation module, a HSAemu runtime library, and an LLVM HSAIL translator. In the following, we will describe each part in details.
4.1 The CPU Simulation Module

The CPU simulation module simulates each target CPU core by one host thread in parallel. All of CPU threads can execute OpenCL agent to issue commands to the parallel GPU simulation module. The design of the CPU simulation module is based on PQEMU. The traditional emulators adopt a time-sharing scheme to emulate a multi-core CPU. In PQEMU, it provides two efficient synchronization models, unified code cache (UCC) and separate code cache (SCC), to emulate a multi-/many-core CPU. The two models solve the parallelization issues of CPU simulation in a full system emulator. For example, it can parallelize the dynamic binary translation (DBT) engine and manage thread execution in the code cache without any race condition. Based on PQEMU, we have added two features, a command detector and a GPU work interrupt, to co-simulate CPU and GPU. A command detector is used to detect a command delivered by the HSAemu runtime library. It is implemented by a
software interrupt instruction of target CPU, such as SWI instruction in the ARM ISA. The GPU work interrupt is used to notify the GPU command monitor that a new job is arrived.

4.2 The GPU Simulation Module

4.2.1 The GPU Command Monitor (GCM)

The GPU command monitor is used to handle AQL packets. It has two main components, command monitor and AQL packet worker. The command monitor is used to receive the GPU work interrupt from the CPU simulation module. It is implemented by a conditional wait mechanism of Pthread library. The AQL packet worker is used to dispatch work groups from AQL packet to the GPU execution engine (GEE). For AQL packet dispatching, the AQL packet worker will first dequeue an AQL packet from the copied AQL queue in GCM, which is a copy of AQL queue managed by the HSAemu runtime library. Each AQL packet contains BRIG (kernel function), ARG (arguments of kernel function) and kernel information, such as the number of work groups, the dimension of a work group, and the work group size. It then will invoke the GPU translation engine (GTE) to translate the BRIG in an AQL packet to host native binary. After the translation is done and the host native binary is stored in GPU shared code cache, the AQL packet worker will notify GEE to execute the translated kernel function based on the kernel information. Each AQL packet will be handled by the AQL packet worker one at a time. Before GEE completes the execution of the current translated kernel function, the status of the AQL packet worker is set to a busy state. Once the execution of the current translated kernel function is completed by GEE, the status of the AQL packet worker will be set to a free state and the AQL packet worker is allowed to fetch next AQL packet. If no AQL packet in the copied AQL queue, GCM will block itself and waits for CPU command to start next command dispatching.

4.2.2 The GPU Translation Engine (GTE) and LLVM HSAIL Translator

The purpose of the GPU translation engine (GTE) is to translate BRIG, the binary format of HSAIL, to host native binary and put the translated host native binary into GPU shared code cache in GCM. GTE consists of two components, an external translator and a linking loader. When receiving BRIG from GCM, GTE starts translating it to unlinked host binary code by using an external translator (LLVM HSAIL translator). After the external translation, GTE calls the linking loader to link related helper functions to the unlinked host binary code. The linked host binary code is then stored in the GPU shared code cache. In the following, we describe the external translator and linking loader in details.

Since it is not easy to implement a complete binary translator in a full system emulator and a tightly-coupled design will lose flexibility, the translator of GTE is designed as a dynamic link library, called external translator. This loosely-coupled design of translator has several advantages, including

- **Ease of implementation**: The external translator design can reduce the implementation complexity of a translator since the developer does not need to know the simulation mechanism of HSAemu at all. The only thing to do is translating source target codes to unlinked host binary codes.

- **Ease of configuration**: The external translator design has more flexibility to reconfigure the translator. For instance, the current external translator in HSAemu is implemented for HSAIL. But it could be replaced by another external translator for CUDA.
- **Optimization**: The most attractive part of the external translator design is the translated code optimization. Current external translator based on LLVM can generate more highly optimized host native codes than the tiny code generator (TCG) in QEMU. The key reason is that TCG translates small piece of codes, called basic block, once at a time. Comparatively, our external translator can translate a whole kernel function once at a time because the code going to be executed on the GPU is relatively small and the whole substance is always ready when the command is issued. A similar method proposed by HQEMU [Hong et al. 2012] also shows speed up of translated codes.

- **Portability**: LLVM is used as the compiler framework of the external translator design. The portability of the design is high since the HSAIL can be translated into different native ISAs via LLVM.

The external translator we designed for HSAemu is called LLVM HSAIL translator. It is a runtime static compiler that uses static binary translation technique to translate kernel function to an unlinked object file. The external translator consists of three components, Flow Constructor, HDecoder, and HASsembler. The Flow Constructor is used to reconstruct the control flow of HSAIL code in BRIG and feeds the control flow tree to HDecoder. The HDecoder is used to translate HSAIL code to LLVM bitcode based on the control flow tree. The HASsembler is used to translate the LLVM bitcode to an unlinked object file.

There is a register allocation issue when translates HSAIL code to LLVM bitcode. In HSAIL code, the number of registers used is finite and the same as that of hardware. However, LLVM bitcode uses infinite virtual registers in order to keep bitcode as static single assignment (SSA) instructions. To keep SSA property when translating HSAIL code to LLVM bitcode, the registers used by HSAIL code are represented as a stack in HDecoder. The load/store operations to these registers are implemented as push/pop stack operations. No doubt, this implementation incurs extra overhead. Fortunately, the overhead can be eliminated by LLVM optimization pass.

In an unlinked object file, there may have some external helper functions, such as memory, kernel information, mathematics, and synchronization helper functions. The linking loader is used to resolve those external helper functions in the unlinked object file to form an executable host native binary. To link external helper functions, the linking loader scans its symbol table to find the addresses of external helper functions. To inline the helper functions when translating BRIG is another approach to resolve external helper functions issue. But it will cause some side effects. The first one is the inline technique cannot be applied to the helper functions that deal with virtual memory access since some QEMU global variables cannot be accessed directly. The second one is the inline of helper functions increases the code size. It is difficult to predict the size of a program when the inline technique is applied. Therefore, we prefer to use the linking loader technique over the inline one in our design.

4.2.3 **The GPU Execution Engine (GEE)**

The purpose of the GPU execution engine is to perform the execution of work groups from an AQL packet dispatched by GCM. To support an execution environment of simulated GPU, two implementations of CU thread schedulers (static and dynamic) and several GPU kernel helper functions are provided in the GEE. The CU thread
schedulers will firstly interpret kernel information fetched from GCM to obtain the number of work groups and the size of work group. Then, the CU thread schedulers will run work groups by the GPU CU threads in parallel. The GPU kernel helper functions are used to simulate complex GPU instructions that are considered as external helper functions by GTE. These GPU instructions are related to memory access, mathematical operation, kernel information operation, and synchronization. In the following, we will describe the CU thread schedulers and the GPU kernel helper functions in detail.

A physical GPU may have many compute units (CUs) and each CU also contains many processing elements (PEs). To speed up the GPU simulation speed, parallelize the execution of GPU is the best choice. One can parallelize the execution of CUs, or parallelize the execution of PEs, or parallelize the execution of both CUs and PEs. In GEE, we choose to parallelize the execution of CUs and leave the execution of PEs within each CU sequentially. The reasons are twofold. The first one is to parallelize the execution of all PEs in a GPU by creating the same number of threads may beyond the limitation of the host OS. The second one is the GPU simulation may use host hardware to speed up the simulation, such as host GPU or SIMD compute unit. To parallelize the execution of CUs makes the mapping of the simulated CUs to physical CUs with real SIMD compute unit easier. In HSAemu, we did do this by using the SSE instruction of host CPU to speed up the execution of GPU simulation. The details will be described in Section 4.5.

To parallelize the execution of CUs, GEE has implemented two schedulers, static and dynamic, for the CU threads scheduling. For the static scheduler, the work items are evenly distributed to the CU threads by using the block partition method in the beginning of GPU execution. In this manner, each CU gets the same number of work items. But the execution time of each CU may be different due to various workloads of working items. Therefore, the static scheduler may suffer from load unbalancing problem. In the dynamic scheduler, the working items are stored in a queue. Working items are distributed to CU threads dynamically based on the availability of CU threads. A lock is needed for working item queue due to multiple CU threads access the queue concurrently. The lock overhead may become the bottleneck of GPU execution. The experimental results obtained in Section 5 show some clues about the simulation speed under different scheduling policies. However, we need to conduct more benchmarks before making any conclusion. Therefore, at the current design, we let the users to decide what scheduler they want to use.

For the GPU's available today, the work items of a work group are parallel executed by PEs of a CU. But in GEE, the work items of a work group are executed in a CU sequentially. This may raise a synchronization issue from the use of barrier instruction in a work group. To handle the synchronization issue, in GEE, we implement a light weight barrier thread to guard the execution within CU. This light weight barrier thread will make the execution of PEs of a CU in parallel during the synchronization. As mentioned above, if parallelize all the PEs, we may face the situation that the number of threads created beyond the limitation of the host OS. So when we do the synchronization using the light weight barrier thread in one CU, we will block the execution of other CUs so that there will not be too many threads.

In HSAIL, some instructions, such as memory access, mathematical operation, kernel information operation, and synchronization, cannot be simulated by CPU instructions. For these instructions, we use helper functions to simulate their execution. When the LLVM HSAIL translator finds one of these instructions, it creates the corresponding external helper function call for this instruction. When an external helper function call is executed during the GPU simulation, the execution will be directed to the corresponding helper function. After the execution of the corresponding helper function is completed, the execution will go back to the caller
with some return values. In GEE, the following four types of helper functions are implemented:

- **Memory Helper Function**: To meet the requirement of shared virtual address space between CPU and GPU, memory access in GPU must be performed through an MMU. The memory helper function, a separate GPU soft-mmu with a page table worker and a TLB, is used for such purpose. In HSA standard, a GPU may redirect access of a local segment memory to a non-shared private memory for performance consideration. With the memory helper function, HSAemu can also support this kind of hardware implementation properly by redirect the virtual address in the soft-mmu.

- **Mathematical Helper Function**: For real GPU architecture, it may be more efficient to support special mathematical instructions such as trigonometric instructions and so on. However, these mathematical instructions may not be supported in host CPU ISA. The mathematical helper function is used to simulate such mathematical instructions by calling the corresponding mathematical functions in standard library.

- **Kernel Information Helper Function**: To assist GPU applications running adaptively to the underlying GPU, GPU will provide query instructions about current hardware situation and information. To simulate these query instructions, the kernel information helper function is used to collect and return information of GPU simulation module and current execution state.

- **Synchronization Helper Function**: In the current GPU design, work items of a work group are parallel executed by PEs of a CU. But in GEE, the work items of a work group are executed in a GPU CU thread sequentially. This may raise a synchronization issue from the usage of barrier instruction in a work group. To handle the synchronization issue, in GEE, we implement a synchronization help function to guard the execution within a GPU CU thread. The synchronization help function is a lightweight GPU PE barrier thread. It will turn the execution of PEs of a CU from sequential to parallel during the synchronization. As mentioned above, if parallelize all the PEs, we may face the situation that the number of threads created beyond the limitation of the host OS. When the synchronization helper function is invoked in a GPU CU thread, it will block the execution of other GPU CU threads so that there will not be too many threads.

### 4.3 The HSAemu Runtime Library

In general, an HSA-compliance application consists of two parts, CPU functions and GPU kernel functions. The CPU functions can be directly executed by the CPU simulation module in HSAemu. But, the GPU kernel functions need to be passed to the GPU simulation module by the HSAemu runtime library. To communicate with the GPU simulation module, the HSAemu runtime library supports an AQL protocol defined in the HSA standard. The AQL protocol uses AQL packets as commands to dispatch GPU kernel functions to the GPU simulation module. Currently, there are two components, AQL queue manager and AQL command dispatcher, in the HSAemu runtime library. The AQL queue manager is used to handle all incoming GPU kernel function invocations from HSA-compliance applications. For each GPU kernel function invocation, the AQL queue manager will first pack it into an AQL packet. Then, the generated AQL packet will be enqueued to the AQL queue in the HSAemu runtime library. When a new AQL packet is enqueued to the AQL queue,
the AQL queue manager will also prepare some resources (memory, kernel function arguments and so on.) required to execute the corresponding kernel function. Finally, the AQL queue manager will send a command to the AQL command dispatcher. The AQL command dispatcher is responsible for dispatching AQL commands to the CPU simulation module by software interrupt of target CPU architecture. After the AQL command is processed by the CPU simulation module, the GPU kernel function will be scheduled to the GPU simulation module for execution.

4.4 The Emulation Flow of HSAemu

We have described the components of HSAemu in the previous three subsections. In this section, we will explain the emulation flow of HSAemu. The description will follow the number signs shown in Figure 2.

- **HSAemu Initialization:** In the initialization phase, the CPU simulation module creates multiple CPU simulation threads to simulate multi-core processor and the GPU simulation module creates two kinds of threads: one GPU monitor thread and several GPU CU threads. After the initialization is done, HSAemu starts to run the emulated machine. The emulated machine can run a complete guest operating system, such as Ubuntu Linux, Android System, and so on. Under the guest operation system, an HSA-compliance application can be executed via the HSAemu runtime library.

- **AQL Packet Generating:** When an HSA-compliance application calls its kernel functions, the HSAemu runtime library is invoked to generate the AQL packet that includes the kernel function and related information. And then the AQL packet is enqueued to the AQL queue in the HSAemu runtime library.

- **AQL Command Dispatching:** Once a new AQL packet is issued, the AQL command dispatcher will dispatch the command to the CPU simulation module by software interrupt. The address of the AQL queue is passed through the AQL command.

- **GPU Work Interrupt:** When the AQL command detector receives an incoming command from the AQL command dispatcher, it triggers a GPU work interrupt to GCM in the GPU simulation module. This will notify the GPU simulation module to work for this AQL command.

- **AQL Packet Worker Invocation:** After receiving an AQL command, the AQL command monitor forwards the AQL command to the AQL packet worker. The AQL packet worker retrieves the address of the AQL queue from the AQL command. The whole AQL queue is copied into GCM from the HSAemu runtime library. Then the AQL packet worker does the following three jobs until the copied AQL queue is empty:
  1. Interpret the AQL packet at the top of the queue and copy the kernel function (in BRIG format) and argument whose addresses are stored in the AQL packet to the internal memory of GPU simulation module.
  2. Invoke the GPU translation engine to translate the BRIG into host native binary.
  3. Invoke GPU execution engine to execute the translated kernel function based on the kernel information.

When the copied AQL queue is empty, the AQL packet worker blocks itself and waits for new AQL command to come.

- **BRIG Translation:** The GPU translation engine is called to translate BRIG to host native binary. Instead using an internal translator, we have an external interface to serve the HSAIL translation. After the external translation is done, the Linking Loader links the unlinked object file with the GPU kernel helper functions and store the host native binary to the GPU shared code cache.
- **External Translator Invocation**: The external translator, an LLVM HSAIL translator, builds the control flow of HSAIL code and translates the HSAIL code to LLVM bitcode. By the LLVM backend, the LLVM bitcode can be translated into the unlinked object file.

- **GPU Execution**: When the GPU execution engine is invoked by the AQL packet worker, the CU thread scheduler distributes work items to GPU CU threads for execution. After all the GPU CU threads have finished all the work items assigned, the control flow goes back to the AQL packet worker for next available AQL packet.

### 4.5 Optimization for GPU Emulation by Host SIMD Instruction

In this Section, we describe how to use the host CPU’s SIMD unit to speed up the execution of the GPU execution engine. In our case, the instruction set of host CPU’s SIMD unit is Streaming SIMD Extension 3 (SSE3). When SSE3 instructions are used to simulate the GPU execution, each GPU CU thread should follow the same control flow and executes the same instruction with different data. Since each GPU CU thread may have different branch result depending on its ID and data it deals with, the LLVM HSAIL translator needs to reconstruct the control flow graph of kernel function and do bitmap masking to ensure the execution result of each GPU CU thread is correct. In the following, we will describe how to reconstruct the control flow graph of kernel function and do bitmap masking in details.

#### 4.5.1 The Control Flow Graph Reconstruction

The LLVM HSAIL translator is a one-pass code generator. It scans the whole program to reconstruct the control flow graph of a kernel function. A control flow graph is composed of nodes that contain block name, jump label, and SIMD flag. According to the number of jump label and SIMD flag, a node can be classified into the following categories:

- **Return node**: A node has no jump label and its SIMD flag is false.
- **Direct jump node**: A node has one jump label and its SIMD flag is false.
- **Conditional jump node**: A node has two jump labels, taken and non-taken; and its SIMD flag is false.
- **Direct jump for SIMD node**: A node has one jump label and its SIMD flag is true.
- **Conditional jump for SIMD node**: A node has two jump labels, taken and non-taken; and its SIMD flag is true.

Given the control flow graph of a kernel function, the algorithm to perform the reconstruction of control flow graph is as follows:

```java
Algorithm CFG_Reconstruction(KF)
{ /* KF is a given kernel function */

1. Let CG be the control flow graph of KF, RCG = {} be the reconstructed control flow graph of CG, and S = {} be an empty stack;

2. /* Traverse CG by using the depth first search (DFS) */
3. Let N be the current traversed node in CG;
4. while (True)
5. { switch of (N)
6. Case **Return Node**:
7. If S is empty,

```
8.  { Create a return node \( A \):
9.      Set the jump label of \( N_C \) to \( A \):
10.     Insert \( A \) and \( N_C \) to \( RCG \):
11.     Stop the traversal: }
12. else
13.  { Create a conditional jump for SIMD node \( A \):
14.      Duplicate a return node \( A' \) from \( N_C \):
15.      Change \( A' \) to direct jump for SIMD node:
16.      Set the jump label of \( A' \) to \( A \):
17.      Insert \( A' \) to \( RCG \):
18.      Set \( N_C \) to \( A \):
19.      continue: }
20. Case Conditional Jump Node:
21.   { Insert \( N_C \) to \( RCG \):
22.      Push the non-taken jump label of \( N_C \) (the right child of \( N_C \)) to \( S \):
23.      Let \( N_C \) be the taken jump label of \( N_C \) (the left child of \( N_C \)):
24.      continue: }
25. Case Direct Jump Node:
26.   { Insert \( N_C \) to \( RCG \):
27.      Let \( N_C \) be the jump label of \( N_C \) (the child of \( N_C \)):
28.      continue: }
29. Case Conditional Jump for SIMD Node:
30.   { if \( S \) is empty,
31.      { Change non-taken jump label of \( N_C \) to return node \( ret \):
32.        Set the non-taken jump label of \( N_C \) to \( ret \):
33.        Insert \( N_C \) to \( RCG \):
34.        Break: }
35.    else
36.      { Pop \( N_s \) from \( S \):
37.        if \( N_s \) is not a conditional jump for SIMD node
38.          { Set the taken jump label of \( N_s \) to \( N_s \):
39.            Create a conditional jump for SIMD node \( A \):
40.            Push \( A \) to \( S \):
41.            if the jump label of \( N_s \) is return node,
42.              { Set the taken jump label of \( N_s \) to \( N_s \):
43.                Set \( N_C \) to \( N_s \):
44.                if \( N_C \) is return node,
45.                  { Set the taken jump label of \( N_C \) to \( N_s \):
46.                    Insert \( N_C \) to \( RCG \):
47.                    Set \( N_C \) to \( N_s \):
48.                } }
49.            continue: }
50.    } End_of_Algorithm CFG_Reconstruction

Take the kernel function shown in Fig. 3 as an example, we explain how algorithm Algorithm CFG_Reconstruction(KF) works. The control flow graph of the kernel function given in Fig. 3 is shown in Fig. 4(a). By using the depth first search (DFS) to traverse Fig. 4(a), node B0 is the first node to visit. We have the following execution steps:

Step 1: Since node B0 is a conditional jump block, node B0 is inserted to \( RCG \), node B4 is pushed onto stack, and node B1 is set as the next node to visit.
Step 2: Since node B1 is conditional jump block, node B1 is inserted to \textit{RCG}, node B3 is pushed onto stack, and node B2 is set as the next node to visit.

Step 3: Since node B2 is a direct jump block, node B2 is inserted to \textit{RCG} and node B5 is set as the next node to visit.

Step 4: Since node B5 is a return block and the stack is not empty, create a conditional jump for SIMD node A2, duplicate A1 from B5 and A1 is changed to direct jump for SIMD node, set its jump label to A2: node A1 is inserted to \textit{RCG}, and node A2 is set as the next node to visit.

Step 5: Since A2 is a conditional jump block for SIMD: the stack is not empty, B3 is popped and B3 is not a conditional jump block for SIMD. The taken jump label of A2 is set to node B3 and jump label of B3 is to original return block B5, the jump label of B3 is set to node A2: create a conditional jump for SIMD node A3 and push A3 to stack. Node B3 is set as the next node to visit.

Step 6: Since B3 is directly jump block, insert B3 to \textit{RCG}, and node A2 is set as the next node to visit.

Step 7: Since A2 is a conditional jump block for SIMD: the stack is not empty, A3 is popped and A3 is a conditional jump block for SIMD. The non-taken jump label of A2 is set to node A3, insert A2 to \textit{RCG} and node A3 is set as the next node to visit.

Step 8: Since A3 is a conditional jump block for SIMD: the stack is empty, B4 is popped and B4 is not a conditional jump block for SIMD. The taken jump label of A3 is set to node B4 and jump label of B4 is to original return block B5, the jump label of B4 is set to node A3: create a conditional jump for SIMD node A4, push A4 to stack. Then insert A3 to \textit{RCG}, and node B3 is set as the next node to visit.

Step 9: Since B4 is directly jump block, insert B4 to \textit{RCG}, and node A3 is set as the next node to visit.

Step 10: Since A3 is conditional jump for SIMD node and stack is empty, the non-taken jump label of A3 is change to a return node ret, and Set the non-taken jump label of nc to ret; Insert NC and ret to \textit{RCG}.

The reconstructed control flow graph of Fig. 4(a) is shown in Fig. 4(b).

Fig. 3. An HSAIL sample code of a kernel function that has computation instructions and conditional branch instructions.
4.5.2 How to Do Bitmap Masking?

Even though the reconstruction of control flow graph can help HSAemu use SIMD instructions to speed up the GPU simulation, it may create wrong result since each GPU CU thread may have different branch result depending on its ID and data it deals. As a result, keep the simulation result correct with SIMD instruction becomes a critical issue. The HSAemu uses the bitmap masking approach to ensure the correctness of the simulation. Given a constructed control flow graph, the bitmap masking setting for each node is as follows:

- **Return node**: No setting.
- **Direct jump node**: For the jump label, its bitmap is the same as that of the direct jump node.
- **Conditional jump node**: For the taken jump label, its bitmap is \((\text{bitmap} \And \text{conditional result})\). For the non-taken jump label, its bitmap is \((\text{bitmap} \And \neg \text{conditional result})\).
- **Direct jump for SIMD node**: If its jump label is a direct jump for SIMD node, the bitmap of the jump label is the same as that of the direct jump node; Otherwise, no bitmap setting for the jump label.
- **Conditional jump for SIMD node**: Its bitmap is the same as that of the taken jump label. For the non-taken jump label, if it is a direct jump for SIMD node, its bitmap is the same as that of the taken jump label; Otherwise, no bitmap setting for the non-taken jump label.

In the following, we give an example to show how to set the bitmap masking for a given reconstructed control flow graph shown in Fig. 5. Assume that work item 0 and work item 1 traverse B0, B1, B2 and B5. Work item 2 traverses B0, B1, B3 and B5. Work item 3 traverses B0, B4 and B5. By using the depth first search, the bitmap setting for each node is performed as follows:

Step 1: In B0, its bitmap is 1111. Since B0 is a conditional jump node, its taken and non-taken jump labels are B1 and B4, respectively. The bitmap of B1 is \((1111 \And 1110) = 1110\) and the bitmap of B4 is \((1111 \And 0001) = 0001\).
Step 2: In B1, its bitmap is 1110. Since B1 is a conditional jump node, its taken and non-taken jump labels are B2 and B3, respectively. The bitmap of B2 is (1110 AND 1100) = 1100 and the bitmap of B3 is (1110 AND 0011) = 0010.

Step 3: In B2, its bitmap is 1100. Since B2 is a direct jump node, the bitmap of its jump label, A1, is 1100.

Step 4: In A1, its bitmap is 1100. Since its jump label, A2, is a conditional jump for SIMD node, no bitmap setting for A2.

Step 5: In A2, it is a conditional jump for SIMD node. Its bitmap is the same as that of its taken jump label B3, that is, 0010. Since the non-taken jump label is A3, a conditional jump for SIMD node, no bitmap setting for A3.

Step 6: In B3, its bitmap is 0010. Since B3 is a direct jump node, the bitmap of its jump label, A2, is set to 0010.

Step 7: In A2, the setting is the same as that of Step 5.

Step 8: In A3, it is a conditional jump for SIMD node. Its bitmap is the same as that of its taken jump label B4, that is, 0001. Since the non-taken jump label is B5, a return node, no bitmap setting for B5.

Step 9: In B4, its bitmap is 0001. Since B4 is a direct jump node, the bitmap of its jump label, A3, is set to 0010.

Step 10: In A3, the setting is the same as that of Step 8.

Step 11: In B5, since it is a return node, no bitmap setting.

With the bitmap setting for each node in the reconstructed control flow graph, the HSAemu can keep the simulation result correct. Take instructions in block B2 as an example. Assume that the source register is an unmodified destination register and the temp register is used to store the computation results as shown in Fig. 5. The original instruction in B2 is

```
add_u32 $s5, $s1, $s2;
```

In HSAemu, this instruction will be simulated by the corresponding SIMD instruction in SSE3. Before writing computation results into the destination register, HSAemu uses the bitmap of B2, 1100, to keep unused data remain the same. In the given example, the data stored in the source register is [v1, v2, v3, v4]. After the execution of SIMD instruction, the computation results stored in the temp register are [n1, n2, n3, n4]. By applying the bitmap of B2 to the source register and temp register, the computation results stored in destination register is [n1, n2, v3, v4]. The simulation result is correct since only work item 0 and work item 1 traverse node B2.
HSAemu is implemented based on both QEMU-1.3 and PQEMU [Ding et al. 2011]. Using PQEMU as a base, HSAemu could simulate multiple agent threads running on multiple host cores. However, due to the lack of HSA compiler tools, we are more concerned about the correctness of CPU-GPU co-simulation at this stage rather than testing the capability of a multi-threaded agent. Therefore, in the CPU-GPU co-simulation mode, HSAemu uses only one host CPU core to simulate the guest CPU and leaves all remaining CPU cores for emulating the execution of kernel functions on the compute intensive GPGPU. Table 1 lists the details of the experimental set up.

### Table 1. Experimental Environment

<table>
<thead>
<tr>
<th></th>
<th>Linux:3.5.0-1-linaro-vexpress</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guest OS</td>
<td></td>
</tr>
<tr>
<td>Guest Machine</td>
<td>ARM vexpress-a9 and GPU simulation</td>
</tr>
<tr>
<td>System Emulator</td>
<td>QEMU-1.3 and PQEMU</td>
</tr>
<tr>
<td>Host OS</td>
<td>CentOS Linux release 6.0</td>
</tr>
<tr>
<td>Host CPU</td>
<td>Intel(R) Xeon(R) CPU X7550 @ 2.00GHz 32 cores with 64 Hyper Threads</td>
</tr>
</tbody>
</table>

#### 5.1 Benchmarks

Due to the lack of HSA compliant OpenCL compilers, we are not able to test run HSAemu using large set of OpenCL benchmarks. Instead, at this early stage, we use a few hand-written kernel functions in standard HSAIL to test drive the preliminary HSAemu. Three kernel functions tested are Nearest Neighbor, Kmeans, and FWT. The first two benchmarks are selected from the Rodinia benchmark suite 2.3. FWT is taken from the AMD OpenCL benchmark suite.
The Nearest Neighbor (NN) benchmark calculates the Euclidean distance from the target latitude and longitude, and finds the nearest neighbors of each node. The number of nodes is 4096, so the total work item size is 4096x4096, and the location of each node is in 64 dimensions, the work group size is 16 * 16.

Kmeans is a clustering algorithm. In this benchmark, the size of objects is 819200, with 32 features and using a group size of 4.

FWT computes the discrete Wavelettransform. It is taken from the AMD APP SDK. The problem size used in this benchmark is 4x4096x4096x.

The kernel functions of the above three benchmarks are coded in HSAIL, and translated from HSAIL into BRIG (the standard executable format for HSAIL programs) using an HSAIL assembler developed in house. The BRIG file and the respective arguments sent from the agent are specified in an AQL packet. The AQL packet will be dispatched to GPU for execution.

The execution time in all figures of this section represents only the execution time of the GPU to avoid interferences from the simulation of the guest machine which is relatively slow due to dynamic binary translation.

5.2 Experimental results

HSAemu is a functional simulator. It is used primarily for software development and testing, and is not suitable for microarchitecture design evaluations. To increase the speed for functional emulation of kernel function execution on GPU, HSAemu allows each CU thread to run on a different physical core. This enables HSAemu to take advantages of abundant parallelism existing on current multi-core based desktops and servers to achieve very fast CPU-GPU co-simulation.
As Figure 6 shows, when more CU threads are used in GPU simulation, the total simulation time is dropped quickly until the number of CU thread exceeds the number of physical cores. When the number of CU thread is greater than the number of physical cores, all computation resources are in use, and only marginal performance can be further gained through hyperthreading of Intel Xeon processors. The trend of such simulation performance curves are similar no matter how many physical cores are used. For example, in Figure 7, the performance curves of both NN and Kmeans benchmark level off at 8 cores when 8 physical cores are in use, level off at 16 when 16 physical cores are in use, and level off at 32, when 32 cores are in use. For this set of experiments, we used the Linux command "\texttt{taskset}" to limit the number of physical CPU cores for the emulation runs.

As discussed in section 4.2.3, GPU Execution Engine, HSAemu has two ways to schedule parallel execution of the CU threads. Figure 8 shows the performance of the two schedulers for benchmark NN and Kmeans. The performance curves of dynamic scheduling are appended with "\_dyn" suffix, and curves with static scheduling are appended with the "\_sta" suffix. Here NN uses a work group size of 16*16. We can see that the two schedulers perform differently at some points, each
has its own down side. Figure 9 presents the performance of benchmark NN with
different workgroup size and using dynamic scheduling. As the figure shows, the
performance tends to decrease as the workgroup size decreases. A CU thread fetches

![Nearest Neighbor](image)

![kmeans](image)

Fig. 8. Simulating two benchmarks on HSAemu with different schedulers.

a group of work items at once, hence more fetches would incur when the group size
decreases. More fetches would require more accessing to the lock for dynamic
scheduling, which is likely to increase the waiting time of the CU threads.

Static scheduling requires no locking, yet it suffers more from the load
unbalancing problem. Figure 10 shows the time gap between the first finished CU
thread and the last finished one. As it shows, the time lag peaks out when the
number of CU threads is near 64. At that point, the number of thread of emulation,
including the CPU thread, the IO thread and the monitor thread, plus the CU
threads, exceeds the number of physical CPU cores.

Dynamic scheduling needs to pay for locking overhead, especially when the
amount of work assigned for each CU thread is small. This is why dynamic
scheduling performs worse than static scheduling for the benchmark Kmeans, when
the number of CU threads are greater than the physical cores, as shown in Figure 8.
The Kmeans benchmark has a smaller work group and low kernel complexity, so the
work dispatched to each CU thread is less than other benchmarks, and the impact to performance becomes more pronounced in the simulation runs.

Also shown in Figure 8, load-unbalance causes static scheduling to underperform dynamic scheduling when the number of CU near the number of physical cores. At this point, the lag of finishing time peaks out as shown in Figure 10.

Figure 11 shows the speedup of simulation by exploiting the SIMD parallelism available in each host CPU. In Xeon X7550 processors, SSE3 instructions can carry out four floating point computations at a time. Our BRIG finalizer (i.e. the LLVM based HSAIL translator) is capable of generating SSE3 instructions to speed up the simulation of CU threads. It delivers a 1.8X to 2.7X speedup for Kmeans benchmark, and 2x to 3x speedup for FWT. However, for the NN benchmark, the speedup is only 1.02X to 1.05X. This unimpressive speedup of the NN benchmark is due to frequent calls to help functions in NN. In the NN benchmark, the HSAIL code contains SQRT instructions. However, since the host machine does not have the SQRT instruction,
the BRIG finalizer (i.e. the LLVM based translator) generates a help function call to QEMU in order to simulate this SQRT instruction. When a vector data, i.e. a 128 bit register, is passed to the help function, extra packing and unpacking operations are required, as illustrated in the following diagram. A 128 bit register is first unpacked to four 32bit data items, and then passed to the help function. After the help function finishes the work, the result must be packed back to the 128 bit
register. Unfortunately, in the NN benchmark, such calls to the SQRT help function are quite frequent, so the speed up from SSE3 instruction is pretty much offset by help function overhead, and yield much less speedup than the Kmeans and the FWT benchmark.

6. CONCLUSION AND FUTURE WORK

HSA is an emerging open industry standard to support more efficient heterogeneous computing. At this early stage of development, many tools, such as compilers, functional and micro-architectural simulators, profiling tools, and runtime libraries are all critical to the understanding and the evaluation of this new platform. This work describes the early development of a full system simulator for the HSA platform, called HSAemu. HSAemu is developed based on the popular QEMU and the paralleled version, PQEMU. HSAemu includes a CPU simulation module, a GPU simulation module, a LLVM based HSAIL translator, and many supporting software components. The primary goals of HSAemu at this stage are correct emulation and a reasonably good emulation speed. The CPU simulation module leverages the dynamic translation engine in QEMU to maintain fast simulation of various guest machines (e.g. ARM or MIPS). The GPU simulation module adopts static translation to turn kernel functions (in HSAIL code) into host binaries for parallel execution using many simulated CU threads. In addition to exploiting available parallelism of multi-core processors in the host machines, HSAemu also exploits SIMD parallelism commonly available in host processors such as SSE/AVX in x86 and Neon in ARM. Due to the lack of HSA-ready OpenCL compilers, we have used a few manually coded kernel functions in HSAIL to verify the simulation correctness and simulation speed of HSAemu. For benchmarks with few barriers, the simulation of GPU has a speed up ratio close to the number of physical cores. With additional exploitation of SIMD parallelism, the GPU simulation speed can get a further boost of 2 to 3 times on some benchmarks. The early development of HSAemu has met our initial goals. In the future, we plan to exploit existing GPU parallelism on the host machines to further speed up the GPU simulation in HSAemu. For example, to exploit existing AMD/Visia/Intel GPUs to simulate future HSA compliant devices.

REFERENCES


