HSAemu – A Full System Emulator for HSA Platforms

Jiun-Hung Ding[1], Wei-Chung Hsu[2], Bai-Cheng Jeng[1], Shih-Hao Hung[2], and Yeh-Ching Chung[1]

[1]National Tsing Hua University
Hsinchu, 30013, Taiwan
{adjunhon, bcjeng, ychung}@cs.nthu.edu.tw

[2]National Taiwan University
Taipei, 10617, Taiwan
{hsuwc, hungsh}@csie.ntu.edu.tw

ABSTRACT
Heterogeneous System Architecture (HSA) is an open industry standard designed to support a large variety of data-parallel and task-parallel programming models. Currently, most of HSA hardware and software components are still in development. It is helpful to provide various heterogeneous simulation environments for HSA developers in developing HSA software stacks. This paper presents the design of HSAemu, a full system emulator for the HSA platform, and illustrates how those HSA features are implemented in the simulator. HSAemu provides an infrastructure of heterogeneous simulation environments by supporting required HSA features, including hUMA, hQ and HSAIL. Based on the infrastructure, HSAemu provide two simulation models, FastSim and DeepSim, for high-speed functional emulation and slow cycle-accurate simulation, respectively. In our preliminary experiments, HSAemu helps test a complete HSA software stack and profile system performance. Our case studies show that HSAemu is very useful as a hardware/software co-design tool for heterogeneous systems.

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
Performance, Design, Experimentation.

Keywords
HSA, GPU simulation, parallel simulation.

1. INTRODUCTION
Over the past decade, *heterogeneous computing* has been increasingly adopted in energy efficient computing platforms. *Graphics Processing Unit* (GPUs) have been successfully used as an accelerator to increase the performance and power efficiency for applications, including servers, desktops, and embedded systems. However, the current designs by integrating CPUs and GPUs into a heterogeneous computing platform have several drawbacks. On the hardware side, current CPUs and GPUs have been designed as separate processing elements and do not work together efficiently. For example, since each computing device has its own address space, applications are required to explicitly copy data from one side to another back and forth. This introduces significant programming burden for programmers, as the programmers must handle the required data movements and manage such data transfers when the local memory of the accelerator is not large enough for containing all the data at once. The programmers also pay attention to data locality exploitation in different memory hierarchies. In addition, when a program running on a CPU to request help from a GPU, it sends the job request to a queue waiting for the GPU to process via system calls, which in turn, go through a device driver managed by a completely separate scheduler. Furthermore, it is not feasible for a program running on a 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 a tightly coupled processor design to effectively support heterogeneous computing. HSA intends to cover a large variety of data-parallel and task-parallel programming models by providing a unified view of fundamental computing elements for programmers to write applications. HSA also intends to include more types of accelerators such as ASICs and FPGAs in the future. 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.

To build up next-generation heterogeneous computing environments, HSA has provided specifications that define the hardware and software system architectures. Although there are no HSA-compliant processors available at this time, many of them are under development. In order to support software development in parallel to hardware development, the HSA community must provide a comprehensive system simulator. A full system simulator, such as *QEMU* [1] and *Simics* [2], will help developers for functional debugging and testing software stacks at early stages way before the available hardware. In addition, it can generate event traces and profiling information from complete software systems, including operating systems, runtime libraries, applications, and underlying simulation components.

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

1. Provide a simulation infrastructure of HSA to help developers in developing software for the HSA platform.
2. Develop a functional model to speed up simulation while running a complete HSA software stack.

ESWEEK'14, October 12 - 17 2014, New Delhi, India

http://dx.doi.org/10.1145/2656075.2656088

3. Develop a detailed model to collect detailed profiling information from cycle accurate HSA components, to assist micro-architecture designs of HSA-compliant processors.

To meet the first goal, HSAemu simulates a whole computer system based on HSA specifications, including a shared memory model (*hUMA*) [3], a queuing model for task dispatching (*hQ*) [3] and a virtual ISA for enhanced portability (*HSAIL*) [4]. To simulate hUMA so that both CPU and GPU can share the same virtual address space and page tables, a logically shared, physically separated soft-MMUs, are implemented. For hQ simulation, HSAemu provides a modified OpenCL runtime for HSA to bridge the communication between HSA components using *Architected Queuing Language* (AQL). For processing HSAIL, a LLVM [5] based translator is included to translate HSAIL binary to native codes of HSA components.

To meet the second goal, the functional HSA simulator, *FastSim*, is further optimized for increased simulation speed. While running HSA software applications in HSAemu, it is compelling that the underlying parallelism available in the host machine should be exploited. To speed up the CPU model of HSAemu, we use PQEMU [6] for parallel CPU computation simulation. PQEMU effectively parallelizes the *dynamic binary translation* (DBT) engine in QEMU to achieve highly efficient parallel emulation on multi-core host machines. To speed up the GPU simulation of HSAemu, each GPU compute unit is simulated by a compute thread. A thread scheduler is assigned to dispatch work-groups to compute threads, each compute thread simulates a GPU compute unit so that multiple work-groups are simulated in parallel.

To meet the third goal, we develop *DeepSim* to collect detailed micro-architecture information with a cycle accurate GPU simulation, *Multi2Sim* [7], which models AMD southern islands series. Compared to FastSim, DeepSim can gather more information about the implementation of a GPU, but is much slower than FastSim. Although FastSim does not gather such detailed profiling information as DeepSim, FastSim can gather system memory access events between CPU and GPU, such as TLB misses and page faults. HSAemu can be extended in multiple directions, for example, DeepSim may integrate with *GPGPU-sim* [8] and *GPUWattch* [9] to support GPU other than AMD devices.

To evaluate the infrastructure of HSAemu, a heterogeneous computing simulation environment is configured to run an HSA software stack and an HSA-compatible *N-body Simulation* application. N-body simulation is an ideal example to show the frequent interactions between CPU and GPU and illustrate the benefit of the HSA architecture. Both simulation models, FastSim and DeepSim, are applied in the case studies to reveal their performance and profiling capability. Finally, the parallelism of GPU simulation is measured for FastSim.

The rest of this paper is organized as follows. In section 2, the related works of heterogeneous computing 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. Section 6 summarize this work and describes the future works of HSAemu.

2. RELATED WORK

A *full-system simulator* is an architecture simulator that simulates a computer 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. *SimpleScalar* [10] is a widely used micro-architectural simulator for modeling implementation details of a processor. *Wattch* [11] is also a popular micro-architectural simulator for modeling power consumption of processors. *ZSim* [12] introduces a few novel simulation techniques, such as bound-weave and lightweight user-level virtualization, to make thousand-core simulation practical.

As opposed to micro-architectural simulations, *functional emulations* allow the interactions among processors, memory and peripherals to be observed without modeling microarchitectural details. Recent functional emulators, such as *Embra* [13], *Mambo* [14], *QEMU* [1] and *Simics* [2] usually adopt *dynamic binary translation* for increased simulation efficiency. In today's multi-core environment, parallelism exploitation becomes a major issue in emulator designs. For instance, *PQEMU* [6], *COREMU* [15], *Parallel Mambo* [16], and *Parallel Embra* [17] are all emulators that allow multiple virtual CPUs to be simulated concurrently on the host machine. While MCEmu [18] supports parallel simulation and performance profiling for heterogeneous systems, it is not HSA-compliant and lacks detailed GPU models.

For GPU simulation, several simulators have been proposed in the literature. *GPGPU-sim* [8] provides a detailed simulation of a contemporary GPU running *CUDA* and *OpenCL* [19] workloads with an integrated energy model at micro-architectural level. *Barra-sim* [20] is a functional level GPU simulator based on the *UNISIM* [21] framework, which simulates CUDA programs at the assembly language level and is highly compatible with NVIDIA G80-based GPUs. *Ocelot* [22] is a modular dynamic compilation framework for heterogeneous system, which targets several back-ends with self-developed translator. Ocelot also implemented its own compiler from the IR like PTX to CPU code. AMD *FusionSim* [23] is based on *PTLsim* [24] and GPGPU-sim to simulate an x86 out-of-order CPU, a CUDA-capable GPU and a CPU/GPU interconnected memory system. GPGPU-sim and AMD FusionSim both are micro-architectural level simulators while Ocelot and Barra-sim simulate at functional level.

The *binary translation* (BT) techniques can be divided into two categories: *static* (SBT) and *dynamic* (DBT). SBT translates source binary to target binary once before execution while DBT translates on-the-fly as the execution goes. *LLVM* is a well-known re-targetable compiler framework. LLBT [25] is a LLVM based SBT translator that translates source binary into LLVM IR and then retargets the LLVM IR to various ISAs by using the LLVM compiler infrastructure. In [26], 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. To translate HSAIL code, we need not to go for the full power of DBT since HSAIL is an IR at a level higher than machine codes and does not have the difficult issues associated with machine codes such as code discovery, code locations, and self-modification/self-reference behaviors [27] where DBT would be more suitable than SBT. In HSAemu, an HSAIL kernel function is translated by the device finalizer in a JIT fashion.

3. HSA PRELIMARIES

HSA is motivated to improve *programmability*, *portability*, *manageability* and *performance* for the next-generation heterogeneous computing. HSA attempts to fully exploit the capabilities of heterogeneous parallel execution units by re-architecting computer systems to tightly integrate disparate compute elements on a platform while preserving a programming model that software developers are familiar with.

For programmability, HSA proposes *Heterogeneous Uniform Memory Access* (hUMA) to allow various computing elements to share the same address space. This would allow programmers to easily share data structures among different computing units, without worrying about the complexity of managing data transfer explicitly. Although various computing elements are sharing the same address space, they are likely to apply local caches to deal with latency and bandwidth issues. With a shared address space and local caching, the issue of cache coherence appears. HSA requires hardware devices to support coherent caches.

For increased portability, HSA proposes *HSA Intermediate Language* (HSAIL) to allow OpenCL codes to be translated into intermediate language and then distribute to various platforms. This is similar to the case where a Java program is first translated into bytecode, and the bytecode is distributed to run on various platforms. HSAIL codes are translated by the device *finalizer*, just like bytecodes could be JIT'ed into native code in a JVM.

For manageability and supporting QoS requirements, HSA requires hardware devices to be preemptable. Each computing unit could be interrupted and the interrupted job could be resumed later. This would require an HSAIL program to have a defined context, and can be context switched when a higher priority job is scheduled by the hardware.

For increased performance, HSA attempts to reduce the communication and dispatching latency between different computing devices. HSA proposes *Heterogeneous Queuing* (hQ) which uses user level queues with AQL to communicate with different computing devices. This could avoid communication delays caused by going through system calls.

These features of HSA are introduced in the following subsections.

3.1 hUMA

Traditional GPU uses a separate memory space from the CPU so that programmers must handle explicit memory movements between CPU memory and GPU memory. Instead, hUMA is a shared memory architecture used in systems with many different computing devices. hUMA refers to CPU and GPU sharing the same memory address space with a coherent view. With hUMA, 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. Thus, HSA 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 large as the virtual memory space allows. It can significantly simplify programming on GPU or other accelerators.

3.2 hQ

hQ is proposed to shorten work dispatching latency and communication delay between HSA agents and HSA components. To reduce dispatching latency, *user mode queues* can be allocated and managed by applications. The user mode queues contain AQL packets, which are inserted by applications. Every AQL packet is composed of one kernel function in HSAIL binary, the arguments of the kernel function and some additional kernel information, such as work group size and the number of work groups. Each AQL packet can be dispatched from the user mode queue to hardware queues in computing device for execution. Through hQ, user applications can directly dispatch a job to HSA components without the help of the OS, such as the kernel mode drivers. This enables low latency dispatching compared to traditional job dispatching in OpenCL. HSA-compliant agents and components can communicate with each other by recognizing the format of AQL packets and the mechanism of queuing model. It helps to coordinate the heterogeneity in the HSA computing environment.

3.3 HSAIL

HSA uses HSAIL to represent an intermediate format of GPGPU computing kernels. The HSAIL abstracts the underlying different instruction sets of HSA components into a uniform view. The HSAIL program looks like a simple RISC-like assembly code which has instructions to deal with unified memory accesses, parallel execution and synchronization. The current HSAIL defines 120 instructions, performing arithmetic, memory, branch, image-related, parallel synchronization and device function operations. HSAIL also supports vector instructions. This offers opportunity to generate native SIMD instructions (such as SSE in x86) with less translation time analysis. In addition, 4 types of register width: 1, 32, 64, and 128 bits are supported. One bit register is used for condition code, 32-bit and 64-bit support both single and double precision floating point data. 32, 64, and 128 bits registers can also be used as vector registers for various types of vector formats. For example, 128bits can be organized as 8bit×16, 32bit×4 or 64bit×2.

Developers can write their own programs in high-level languages, such as OpenCL, and then translate them into either text format (HSAIL) or binary format (BRIG) by a front-end compiler. The front-end compiler can provide multiple processing and optimization techniques based on HSA architectures. After translated into HSAIL forms and distributed to various HSA platforms, the HSA runtime can dispatch the HSAIL binary to a HSAIL finalizer before running it on a HSA component. The HSAIL finalizer can be considered as a back-end compiler to translate HSAIL binary to the machine codes of the underlying HSA component. HSAIL is a low-level IR, which has a finite register set and no PHI nodes. A PHI node is an instruction used to select a value depending on the predecessor of the current block in the structure of the *Single Statement Assignment* (SSA) used by LLVM. Finite register set can simplified register mapping analysis and no PHI nodes can eliminate the SSA analysis. In addition, HSAIL excludes high-level symbols such as C structures, and most of optimizations have already applied in the phase of front-end compiler. Therefore, the code generation in the phase of HSAIL finalizer is relatively simple and fast. Once the generated codes are executed in the HSA components, it supposed to benefit from the features of HSA architecture.

As a GPGPU IR, HSAIL provides thorough parallel processing semantics, which allows programmers/compilers to take deeper understanding of the underlying architectures to what they are doing. Data movements with the work groups and lanes are well-defined. Consistency of data in memory is ensured by using acquire and release mechanisms. Whenever data is 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 assured by using atomic instructions. Other synchronization mechanisms, such as barriers, are also provided. However, the barrier is a forced mechanism for all working items to synchronize at the same time, which may unnecessarily decrease performance, so a fine-grained barrier is introduced by HSAIL to synchronize a specified number of work items within a work group.

4. HSAemu

Figure 1 shows the typical components of a HSA-compliant processor that includes a single chip with a multi-core CPU and a

many-core GPU, a memory sub-system and a peripheral I/O subsystem. The two computation units communicate with each other via an interconnection network and access data in the shared cache or main memory with the same virtual addresses. To design

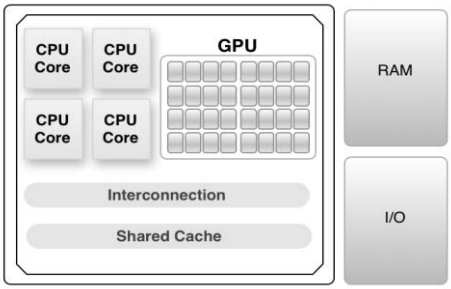

Fig. 1. Heterogeneous System Architecture.

HSAemu for such a heterogeneous system architecture, the following design issues must be considered.

1. *Meeting the HSA-compliant requirements:* Unlike a traditional system where the GPU module is considered as a peripheral device on a PCI bus, in a typical HSA-based system, GPU is controlled directly by CPU through anon-chip interconnection network with the required hardware/software support for running HSA-compliant applications.

2. *Constructing a fast functional simulator*: To accelerate the simulation, we developed *FastSim*, which parallelizes the emulation of the CPU and GPU cores by running multiple threads on the underlying multi-core host machine. To achieve good speedup, it is critical to efficiently manage the emulation threads and handle synchronizations among the threads.
3. *Enabling performance analysis with more detailed micro-architecture simulators and profiling support*: We developed *DeepSim*, which provides the facilities for the user to incorporate cycle-accurate CPU/GPU simulation models. This enables the system developers to monitor and analyze detailed hardware/software interactions for HSA applications and middleware executed on both CPU and GPU.

The remaining of this section presents the overall architecture and simulation flow of HSAemu and then discusses the design issues and explain our solutions for each major component.

4.1 The Architecture of HSAemu

The block diagram shown in Figure 2 illustrates the overall architecture and simulation flow of HSAemu. As the details of each component in HSAemu are to be further discussed in the following subsections, this subsection briefly explains the simulation flow as the following:

- *HSAemu Initialization:* When HSAemu boots up, it is initialized to emulate an HSA-compliant machine with a configuration which describes the number of CPU cores, the type of GPU simulation model, the size of main memory and the type of peripherals. For the CPU-side simulation, *CPUSim* creates multiple CPU simulation threads to simulate the guest multi-core processor in parallel at the functional level. For the GPU side, either *FastSim* or *DeepSim* can be selected to simulate GPU. HSAemu also starts the *GPU Command Monitor* (GCM) daemon thread, which serves to bridge the CPU simulator and the GPU simulator and monitor the GPU simulator. After the initialization, the emulated machine boots up a guest operating system, such as Linux and Android.

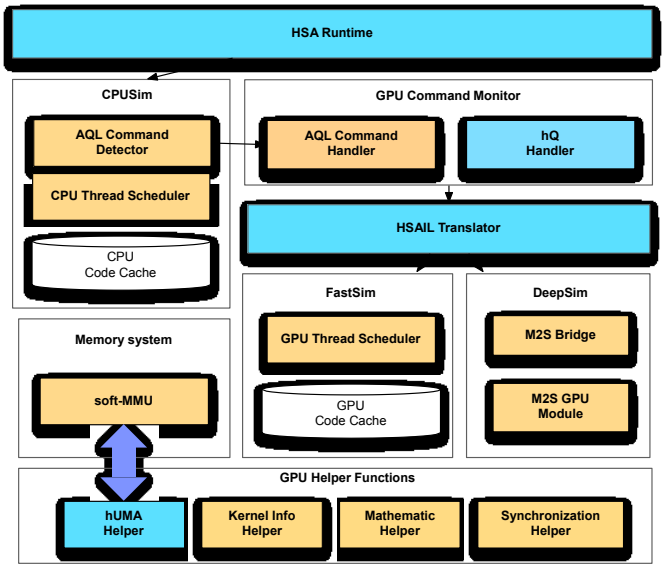

Fig. 2. The overall architecture of HSAemu.

- *HSA-compliant Application Execution:* On top of the emulated machine, HSA-compliant applications can be executed with the *HSA Runtime* which enables the applications to utilize the features of the underlying HSA system. For example, when an OpenCL program starts, it utilizes HSA runtime API's to (1) query the hardware configuration information, (2) allocate input/output memory buffers to create *user mode AQL queues* for dispatching *kernel* jobs to the GPU, and (3) signal the GPU to acquire AQL packets from the user mode queue and execute kernel jobs when the GPU is ready. One of the features of HSA is the hUMA shared memory architecture, and the main program on the CPU, a.k.a. the *agent code*, can utilize hUMA to share data with GPU by passing pointers, without coping data back and forth. The AQL queues in user mode and device hardware both must follow the specifications of hQ. When the agent code calls an HSA API, an AQL command is issued to the user mode queue, and the AQL Command Detector (ACD) in CPUSim immediately detects the AQL command and notifies the AQL Command Handler (ACH) in the GCM, which also emulates the hQ with a *hQ Handler* and controls the execution of commands on the GPU side. A version of *HSAIL Translator Module* (HTM) is used to translate HSAIL's BRIG binary into the native binary for FastSim or DeepSim.
- *GPU Helper Function Invocation:* For the simulation of GPU side, *GPU Helper Functions* may be invoked to assist the GPU simulator in handling HSA-specific operations or features missing in the simulator. The *hUMA Helper* implements the shared memory mechanism defined by HSA. When the GPU simulator accesses the global memory, hUMA Helper may be invoked to handle virtual address translation. HSAemu maintains a *Software Memory Management Unit* (*Soft-MMU)* to emulate a hardware MMU for carrying out the address translation. The *Kernel Info Helper* keeps track of the kernel execution and may be called by the GPU simulator and profiling tools to obtain the configuration and status of the on-going kernel function. The *Mathematic Helper* provides the mathematic functions which require special attention or are not supported by the GPU simulator, e.g. transcendental functions. The *Synchronization Helper* facilitates the synchronization operations across CPU and GPU, e.g. barriers.

4.2 The CPUSim Module

While simple emulators uses a time-sharing scheme to emulate a multi-core CPU, CPUSim leverages the parallel CPU simulation model of PQEMU to take advantage of the CPU cores in the host machine [6]. As the number of cores in the CPU continues to grow, it is essential to employ a parallel model; otherwise, the speed gap between the actual processor and the emulated processor will increase and hurt the usability of simulation.

PQEMU parallelize the execution of *virtual CPU* (VCPU) based on two synchronization models, one is *unified code cache* (UCC) model and another one is *separate code cache* (SCC) model. The two models and respective hierarchical locking schemes effectively address the needs of synchronization in a parallel system emulator. Based on PQEMU, we have added AQL Command Detector (ACD) to detect a command issued by the application via the HSA API. When an AQL command is received, CPUSim forwards this command to GPU Command Monitor. Notice that, in our experimental platform, the HSA Runtime uses software interrupt instructions, e.g. SWI instruction for ARM processor, to identify the occurrence of an AQL command. The detection mechanism really depends on the implementation by the platform vendor.

4.3 The GPU Command Monitor (GCM)

The GCM handles AQL packets with two main components: AQL Command Handler (ACH) and hQ Handler. The ACH receives the AQL commands from CPUSim. It is implemented using a conditional wait mechanism of the *Pthread* library. The hQ Handler is used to dispatch work groups from AQL packets to one of the GPU simulation models. After parsing the command, the ACH passes the address of user mode AQL queue to hQ Handler, and then the hQ Handler will copy the content of user mode AQL queue in HSA runtime to the device AQL queue in hQ Handler. Each AQL packet contains a kernel function in BRIG format, 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. For dispatching an AQL packet, the hQ Handler will dequeue AQL packets from the device AQL queue, and then the hQ Handler repeats the following three steps until the copied AQL queue is empty:

1. Interpret the AQL packet at the top of the queue and copy the kernel function (HSAIL code in BRIG format) and arguments (whose addresses are stored in the AQL packet) to the internal memory of GCM.
2. Invoke the HSAIL Finalizer to translate the BRIG into either host native binary for FastSim or GPU native binary for DeepSim
3. Invoke FastSim or DeepSim to execute the translated kernel function based on the kernel information.

Before the GPU simulation completes the execution of the current translated kernel function, the status of the hQ Handler is set to busy. Once the execution of the current translated kernel function is completed by the GPU simulation, the status of the hQ Handler will be reset to free and the hQ Handler is allowed to fetch the next AQL packet. If no AQL packets in the copied AQL queue, the GCM will block itself and waits for AQL command to start dispatching the next command.

4.4 The HSAIL Translator Module

The HSAIL Translator Module (HTM) is called to translate kernel functions in BRIG format, to run in one of the GPU simulation models. HSAIL Translator consists of two components: an *external translator* and a *linking loader*. When receiving BRIG from the GCM, the HTM starts translating it to unlinked host binary code by using an external translator. After the external translation, the HTM calls the linking loader to link GPU-related helper functions to the unlinked translated code. 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 might decrease flexibility, the translator of HTM is designed as a dynamic linked 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 translated codes.
- *Ease of reconfiguration:* The external translator design has more flexibility to reconfigure. For instance, the current external translator in HSAemu is implemented for HSAIL. But it could be replaced by another external translator for SPIR or PTX.
- *Ease of optimization:* Our current external translator is based on LLVM. LLVM can generate better optimized host native codes than the tiny code generator (TCG) in QEMU. Other than a comprehensive set of optimization phases, LLVM can also translate a whole kernel function once at a time, instead of one block at a time, as done in TCG. HSAIL is more like bytecode than machine code. Its control flow instructions are designed to expose all control flow paths within a function. This feature enables complete code discovery and there the entire function can be translated at once. Unlike the simulation in CPUsim, which is based on DBT via TCG translation in QEMU, HSAIL finalization via SBT allows for more efficient simulation in FastSim.
- *Ease of porting:* LLVM is used as the compiler framework of the external translator, so it is easy to port to machines other than x86.

The external translator we designed for HSAemu is called LLVM HSAIL translator. It is like a Just-in-Time (JIT) translator that uses static IR translation techniques to translate kernel functions 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 trees to HDecoder. The HDecoder is used to translate HSAIL code to LLVM bitcode based on the control flow trees. The HAssembler is used to convert the LLVM bitcode to an unlinked object file.

There is a register allocation requirement when translating 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 in *static single assignment* (SSA) format. To conform to SSA requirement, 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. We are working on a different approach using renaming to achieve the same purpose.

In an unlinked object file, there may have calls to the external helper functions. The linking loader is used to resolve those external helper functions in the unlinked object file to form an executable. 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 possible approach to resolve external helper function references. However, *inlining* may incur 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 that inlining of helper functions may significantly increase the code size. It is difficult to predict the size of a program when inlining is applied. Therefore, we prefer to use the linking approach over the inlining one in the current design.

4.5 The FastSim Module

The purpose of the FastSim module is to efficiently perform the execution of work groups from an AQL packet dispatched by GCM. As GPU typically may have many compute units (CUs) and each CU also contains many processing elements (PEs), parallelizing the execution of GPU seems quite natural, but it can still be tricky, especially when global memory and special operations are encountered. To support an execution environment of a simulated GPU, two implementations of CU thread schedulers (*static* and *dynamic*) and several GPU helper functions are provided. The CU thread scheduler first interprets kernel information fetched from GCM to obtain the number of work groups and the size of work group. Then, the CU thread scheduler runs work groups by the GPU CU threads in parallel.

One can parallelize the execution of CUs, or parallelize the execution of PEs, or parallelize the execution of both CUs and PEs. In FastSim, we choose to parallelize the execution of CUs and leave the execution of PEs within each CU sequentially. The reasons are two-fold. First, parallel execution of all PEs in a GPU may require way too many threads which incur excessive context switch overhead. Second, the GPU simulation may use host hardware to speed up the simulation, such as host GPU or SIMD instructions available in CPU. To parallelize the execution of CUs makes the mapping of the simulated CUs to physical CUs with real SIMD compute unit easier. The implementation of FastSim uses the SSE instruction of the host CPU to speed up the execution of GPU simulation.

For the static scheduler, the work items are evenly distributed to the CU threads by using the block partition method at 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 issues. 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 the work item queue due to multiple CU threads may compete for the queue at the same time. The lock overhead may become the bottleneck of GPU simulation. The experimental results in Section 5 shed some lights on the simulation speed under different scheduling policies, but the user still needs to select the scheduler.

Note that the fact that the work items of a work group are executed in a CU sequentially by FastSim may raise an issue for the use of barrier instruction in a work group. To resolve the synchronization issue, FastSim implements 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. Since we still want to limit the number of threads, 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.

4.6 The DeepSim Module

HSAemu provides an interface for the user to bridge an external cycle-accurate GPU simulator. With the interface, we are able to plug-in the Multi2Sim (*M2S* in short) GPU simulator. M2S starts as a process virtual machine, which can run OpenCL applications with M2S's runtime library without a guest operating system. M2S has two simulation models, functional model and timing model. Both of them sequentially simulate each workgroup, each wavefront and each work item while executing one kernel function. The timing model can further simulate each pipelined stage, including decode, read, execute, write and complete, to caculate simulation cycles. More detailed simulation depends on the model of GPU architecture.

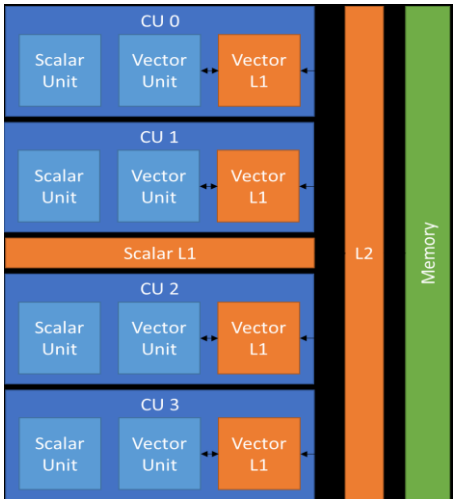

Fig. 3. The AMD Southern Island Series GPU

In our case study, the AMD Southern Island series GPU is chosen to used for cycle-by-cycle simulation. As shown in Figure 3, each CU has one scalar unit and one vector unit. Each vector unit has one vector L1 cache. Four scalar units share one scalar L1 cache. Four CUs share one unified L2 cache. The L2 cache is attached to a GPU memory.The detailed GPU is modeled with 32 CUs.The DeepSim has two main components, *M2S Bridge* and *M2S GPU Module*, to co-simulate with M2S. M2S is compiled as a library to be linked with our M2S GPU Module. M2S Bridge will wait for a kernel jobfrom GCM. GCM packs the necessary kernel information for M2S GPU Module to execute, such as kernel binary, kernel arguments, workgroup size and so on.

When M2S GPU Module starts to execute a kernel function, global memory accesses will be redirected to Memory Helper Function for hUMA simulation. By providing the DeepSim module, more detailed profiling information can be collected, including instruction counts of each CU, the utilization of each CU, L1/L2 cache accesses, pipleine stalls and so on. In addition, HSAemu can provide shared memory access information, such as global memory access count and TLB miss count. The DeepSim gives an example to extend HSAemu to support more accurate simulation models, such as cycle-accuare CPU model or cache model based on the infrastruction of HSA simulation.

4.7 GPU Helper Functions

HSAemu uses helper functions to support HSAIL instructions, which may not be supported by a GPU simulator, such as shared memory access, mathematical functions, kernel information operation, and synchronization operations, When the HSAIL translator encounters one of these instructions, it creates the corresponding external helper function call. When an external helper function call is executed during GPU simulation, the

execution will be directed to the respective helper function. The following four types of helper functions are implemented so far:

- *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 walker and a TLB, is used. In HSA, a GPU may redirect accesses of a local segment memory to a private memory for better performance. With the memory helper function, HSAemu can also support this kind of hardware implementation properly by redirecting the virtual address references 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 functions by calling the corresponding functions in standard math 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 the 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. In FastSim, 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 required synchronizations, in FastSim, 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 turns the execution of PEs of a CU from sequential to parallel during the synchronization.

5. EXPERIMENTAL RESULTS

Table 1. Experimental Environment

HSA Application	Bitonic Sort, Fast Walsh Transform, Reduction, N-Body
OpenCL Runtime	A HSA compatible OpenCL runtime
Guest OS	Linux-3.5.0-1-linaro-vexpress
Guest Machine	ARM vexpress-a9, GPU simulation, and 1 gigabyte memory
Host OS	Ubuntu Linux release 12.04
Host Machine	Intel(R) Core(TM) i7 CPU 920 @ 2.67GHz 4 cores with 8 Hyper Threads and 12 gigabytes memory

In this section, HSAemu is evaluated in three aspects: 1) functionally correctness and usefulness, 2) simulations accuracy for both the functional model and the micro-architecture simulation model, and 3) parallel speedup available from multicore host machines. The results are discussed respectively in Section 5.1, 5.2, and 5.3.

The configurations of our experimental environment are listed in Table 1. As the software infrastructure of HSA is still being developed as of this writing, we managed to adopt four OpenCL benchmark programs as our test workload. The target system is an ARM-based embedded system running Linux. In these case studies, we show that HSAemu is valuable to the development of HSA software stack and applications in an early hardware/software co-design project. It enables software development before the availability of the hardware.

5.1 Developing HSA Software with HSAemu

As Figure 4 shown, the infrastructure has been implemented to support the three fundamental features defined in the HSA

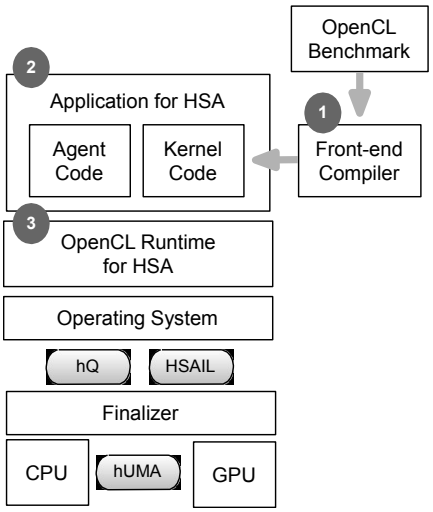

Fig. 4. A basic HSA software stack tested by HSAemu.

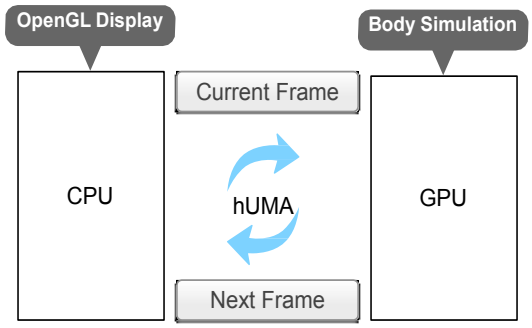

Fig. 5. An N-Body simulation application simulates 2048 bodies running on the HSAemu.

specifications, i.e. hUMA, hQ, and HSAIL. In our case studies, HSAemu runs a complete HSA software stack which is comprised of a front-end compiler, an OpenCL runtime for HSA, an operating system and a finalizer. Using the following steps, we develop and test HSA-compliant applications:

1. *Compile kernel function of application programs to HSAIL's BRIG binary format.* Since the official front-end compiler of HSA has not yet been released for OpenCL, it is difficult to directly compile kernel functions of OpenCL programs to BRIG. To avoid hand-writing HSAIL binary for all kernel functions, a source-to-source translation tool, called PTX2HSAIL is developed. Consequently, the kernel functions of all benchmarks are compiled to PTX assembly code first by NVIDIA *ncc* compiler, and then *PTX2HSAIL* is used to translate the PTX codes to HSAIL.
2. *Modify OpenCL code to utilize the features of HSA.* With HSA's hUMA, OpenCL programs are not required to copy data between main memory and GPU device memory. Hence several OpenCL APIs, such as *clCreateBuffer, clEnqueueWriteBuffer,* and *clEnqueueReadBuffer* are considered as normal main memory accesses to allocate and

read/write. Here we use *N-Body* simulation program to illustrates frequent data interactions between CPU and GPU. As shown in Figure 5, the N-Body program is executed to simulate 2048 bodies. All computations for calculating a new position and a new velocity of each body are handled by the kernel function executed in the GPU. On the other hand, everybody in a visual frame will be painted to screen by the agent code executed in the CPU. This implementation requires frequent data interactions between CPU and GPU. However, such data interactions are no longer explicit in HSA environment. It is possible that implicit data transfers between CPU/GPU are needed, depending on the implementation of the GPU. For example, if local caches are implemented to exploit data locality, an HSA compliant GPU may need to transfer data from the shared memory to the local caches, and transfer updated data from local caches back to memory. In HSA programming, this burden of data transfer management is shifted from programmers to processors.

3. *Modify an OpenCL runtime for HSA.* Since no official HSA runtime has been released for OpenCL yet, we attempt to implement a simplified HSA-compatible OpenCL runtime to enable HSA features. This also illustrates the usefulness of HSAemu as a hardware/software co-design tool. The main changes in the OpenCL APIs are listed in Table 2. For hUMA, it has been mentioned in the previous paragraphs that main memory can be easily shared between CPU and GPU. Hense, *clCreateBuffer*, *clEnqueueWriteBuffer* and *clEnqueueReadBuffer*can simply allocate and read/write buffers by normal main memory access methods without handling GPU device memory access. For hQ, our HSA compatible OpenCL runtime needs to rewrite *clCreateCommandQueue*, *clEnqueueNDRangeKernel* and *clFinish* to satisfy the specifications of hQ, including a user mode queue, an AQL packet handler and a completion object handler. In the last feature, *clCreateProgramWithBinary*is implemented to load HSAIL binary into main memory and then *clBuildProgram* is implemented to translate HSAIL binary to native binary for a specific HSA component. With this runtime support, HSAemu can run OpenCL applications with HSA features. Ideally, when the official HSA compatible OpenCL runtime is released, it will replace our current simplified HSA runtime in HSAemu.

The case studies also serve as a step to verify the functional correctness of HSAemu as a hardware/software prototyping tool. As the hardware and software of HSA become more mature, we continue to refine HSAemu and work with vendors/developers to make the HSA tools chain more complete.

Table 2. Modified OpenCL APIs for HSA

HSA Feature	OpenCL API	Description
hUMA	clCreateBuffer	Create buffer by *malloc* syscall
	clEnqueueWriteBuffer	Write buffer by *memcpy* syscall
	clEnqueueReadBuffer	Read buffer by *memcpy* syscall
hQ	clCreateCommandQueue	Create user mode queue for HSA
	clEnqueueNDRangeKernel	Enqueue AQL packets to user mode queue and signal HSA component to execute
	clFinish	Wait for completion object set by HSA components to finish computation
HSAIL	clCreateProgramWithBinary	Load HSASIL binary format (BRIG) to memory
	clBuildProgram	Translate BRIG to native binary of specific HSA component

5.2 Comparison of FastSim and DeepSim

In this section, we first evaluate the simulation speed for FastSim and DeepSim, and then describe what profiling information can be collected by these two simulation models.

5.2.1 Simulation Performance Evaluation

The simulation time of FastSim and DeepSim was measured by running four OpenCL benchmark programs, *Bitonic Sort*, *Fast Walsh Transform* (FWT), *Reduction* and *N-body*. The first three programs, obtained from AMD OpenCL benchmark suite, were executed with data input size 65536. The N-body program simulated 2048 bodies for one iteration. The simulation time for each benchmark is listed in Table 3, which shows that FastSim is faster than DeepSim up to 85 times and the speedup depends on the workload of kernel functions to be executed in GPU.

Table 3. Simulation Speed of FastSim and DeepSim

Benchmark	FastSim (sec)	DeepSim (sec)	Speedup
Bitonic Sort	7.47	153.07	20.49x
FWT	13.91	49.96	3.59x
Reduction	1.84	5.52	3.00x
N-body	4.02	341.74	85.0x

Table 4. Profiling information for N-Body by FastSim workgroup size = 512

CU ID	Global Memory (ld/st)	Local Memory (ld/st)	TLB Miss	TLB Miss Rate
0	20480/4096	4194304/16384	101	0.41%
1	0	0	0	0
2	20480/4096	4194304/16384	95	0.39%
3	20480/4096	4194304/16384	93	0.38%
4	0	0	0	
5	0	0	0	0
6	0	0	0	0
7	20480/4096	4194304/16384	98	0.40%
Total	81920/16384	16777216/65536	387	0.39%

Table 5. Profiling information for N-Body by FastSim workgroup size = 256

CU ID	Global Memory (ld/st)	Local Memory (ld/st)	TLB Miss	TLB Miss Rate
0	10240/2048	2097152/8192	48	0.39%
1	10240/2048	2097152/8192	54	0.44%
2	10240/2048	2097152/8192	48	0.39%
3	10240/2048	2097152/8192	49	0.40%
4	10240/2048	2097152/8192	50	0.41%
5	10240/2048	2097152/8192	48	0.39%
6	10240/2048	2097152/8192	48	0.39%
7	10240/2048	2097152/8192	48	0.39%
Total	81920/16384	16777216/65536	393	0.40%

5.2.2 FastSim Profiling Information

We first run N-Body Simulation to collect profiling information from FastSim. The N-Body Simulation is configured to simulate 2048 bodies in a single iteration. The workgroup size is configured to 256 or 512. As shown in Table 4, when workgroup size is set to 512, the number of work group is equal to 4 (i.e. 2048/512). Under this configuration, the CU utilization is only 50%, CU 1, 4, 5, and 6 are idle, as indicated by zero activity in Table 4. When the workgroup size is equal to 256, the number of work group is equal to 8 (i.e. 2048/256). This time, the CU utilization is full, as shown in Table 5. Based on Table 4 and 5, we can observe that the memory accesses are consistent, no matter how workgroups are partitioned and assigned to different CUs. For example, in both configurations, the total global memory ld/st access count is equal to 81920/16384 and the total local memory

ld/st access count is equal to 16777216/65536. The TLB miss count and TLB miss rate are also similar for both configurations.

5.2.3 DeepSim Profiling Information

Using the N-Body Simulation with the same configurations as stated in the previous section, DeepSim reports profiling information for AMD Southern Island Series GPU, as shown in Table 6 and 7. The tables also show detailed instruction break downs so that we may conduct sanity check on the number of global and local memory references between FastSim and DeepSim. Multi2Sim reports many implementation detailed information such as cache misses and pipeline stalls. In this paper, we describe the functionality and capability of HSAemu, so more detailed micro-architecture related discussions are left out.

Table 6. GPU profile collected by DeepSim for N-Body

Type	Count	Type	Count
SimTime	1986299.67 ns	Total Instructions	2827168
Frequency	925 MHz	Branch Instr.	66112
NDRangeCount	1	LDS Instr.	263168
WorkGroupCount	8	ScalarALU Instr.	788864
		ScalarMem Instr.	448
		VectorALU Instr.	1708192
Cycle	1812119	VectorMem Instr.	384

Table 7. A CU profile collected by DeepSim for N-Body

Type	Count	Type	Count
Total Instructions	353396	ScalarRegReads	4886264
Branch Instr.	8264	ScalarRegWrites	657992
LDS Instr.	32896	VectorRegReads	24699392
ScalarALU Instr.	98608	VectorRegWrites	15245312
ScalarMem Instr.	56	LDSReads	2097152
VectorALU Instr.	213524	LDSWrites	8192
VectorMem Instr.	48	Scalar L1 Hit Ratio	50.43%
		Vector L1 Hit Ratio	16.88%
		L2 Hit Ratio	80.84%

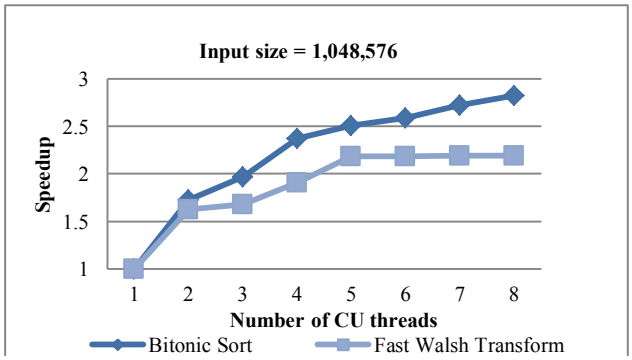

Fig. 6. Two OpenCL benchmarks, Bitonic Sort and FWT, are simulated in the generic functional HSA model.

5.3 Speedup with Parallelized Simulation

Figure 6 shows the speedup of simulation from increasing the number of CU threads in FastSim, which utilizes up to 8 cores to perform fast CPU and GPU simulation. At this early stage, we use a few hand-written kernel functions in standard HSAIL to test 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.

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 "**taskset**" to limit the number of physical CPU cores for the emulation runs.

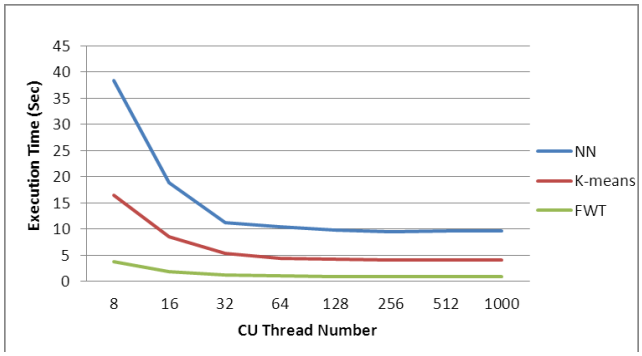

Fig. 7. Benchmarks running on the 32 physical cores.

To further accelerating the simulation, we exploit the *SIMD* capability available in each host CPU, where *SSE3* instructions can perform four floating point computations at a time. Figure 8 shows the resulted speedup after adding this feature. Our BRIG finalizer 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 instruction. However, since the host

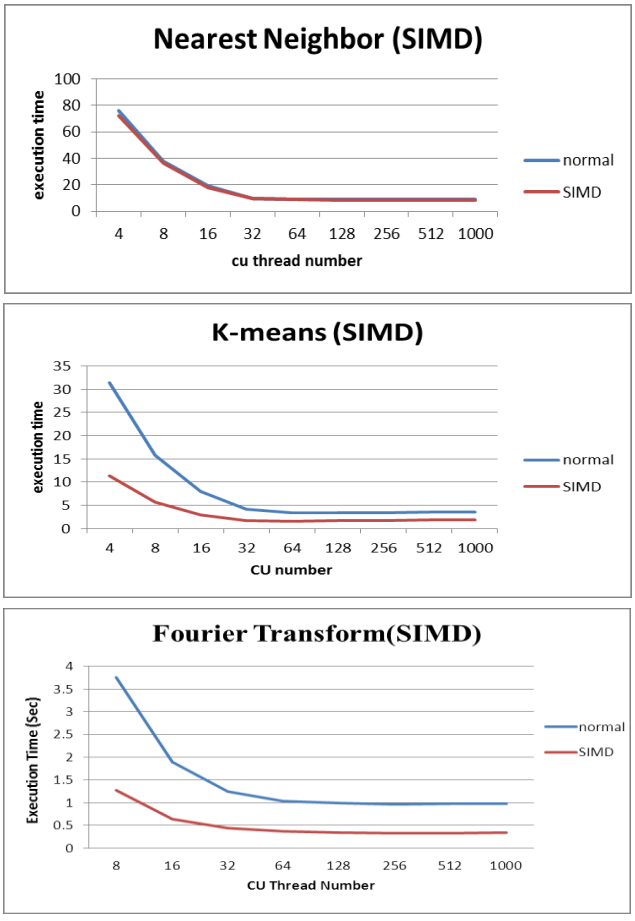

Fig. 8. Execution time with SIMD.

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. 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. CONCLUTION AND FUTURE WORK

HSA is an emerging open industry standard to support high-performance, energy-efficient heterogeneous computing. To accelerate the design of HSA-compliant hardware and software, many tools are needed, including compilers, runtime libraries, simulators, and profiling tools. Our work on HSAemu serves to bring these pieces together for hardware/software co-design projects. We have shown the methodologies and strategies to achieve functional correctness and usefulness, performance modeling, and parallel speedup. In our case studies, we demonstrate the use of HSAemu to model an HSA-compliant system, develop platform-specific software stacks, and carry out performance analysis for OpenCL applications.

As the early development of HSAemu has met our initial goals, by releasing HSAemu as an *open-source project*, we continue to enhance HSAemu together with vendors and researchers. We are currently integrating performance modeling, monitoring, and profiling tools into HSAemu to enable comprehensive performance and power analysis on heterogeneous systems.

7. Acknowledgement

This work was partially supported by MOST of ROC, MediaTek, and AIC under contract MOST-102-2218-E-007-004.

8. REFERENCES

[1] Bellard, F. QEMU, a fast and portable dynamic translator. In *USENIX ATC*, 41-46, 2005.

[2] Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F. Simics: A full system simulation platform. *Computer*. 35, 2 (2002), 50–58.

[3] AMD. A revolutionary, new architecture pioneered by AMD. Retrieved April 20, 2014 from http://www.amd.com/en-us/innovations/software-technologies/hsa

[4] HSA Foundation. *HSA programmer's reference manual: HSAIL virtual ISA and programming model, compiler writer's guide, and object format (BRIG) v.95*. Retrieved April 20, 2014 from http://hsafoundation.com/standards/

[5] Lattner, C., and Adve, V. LLVM: a compilation framework for lifelong program analysis & transformation. In *CGO*, 75–86, 2004.

[6] Ding, J.H., Chang, P.C., Hsu, W.C., and Chung, Y.C. PQEMU: a parallel system emulator based on QEMU. In *ICPADS,* 276–283, 2011.

[7] Ubal, R., Sahuquillo, J., Petit, S., and López, P. Multi2Sim: a simulation framework to evaluate multicore-multithread processors. In *HPCA*, 62-68, 2007.

[8] Bakhoda, A., Yuan, G. L., Fung, W. W. L., Wong, H., and Aamodt, T. M. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*, 163–174, 2009.

[9] Leng, J., Hetherington, T., ElTantawy, A., Gilani, S., Kim, N. S., Aamodt, T. M., and Reddi, V. J. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *ISCA*, 2013.

[10] Austin T., Larson E., and Ernst D. SimpleScalar: an infrastructure for computer system modeling. *Computer*. 35, 2 (2002), 59–67.

[11] Brooks, D., Tiwari, V., and Martonosi, M. Wattch: A Framework for Architectural-Level Power Analysis and Optimization. In *ISCA,* 83-94, 2000.

[12] Sanchez, D., and Kozyrakis, C. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In *ISCA*, 2013.

[13] Witchel, E. and Rosenblum R. 1996. "Embra: fast and flexible machine simulation," In *Proc. ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems,* 68-78, 1996.

[14] Bohrer, P., Peterson, J., Elnozahy, M., Rajamony, R., Gheith, A., Rockhold, R., Lefurgy, C., et al. Mambo: a full system simulator for the PowerPC architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31, 4 (2004), 8–12.

[15] Wang, Z., Liu, R., Chen, Y., Wu, X., Chen, H., Zhang, W., and Zang, B. 2011. COREMU: a scalable and portable parallel full-system emulator. In *PPoPP,* 213–222, 2011.

[16] Wang, K., Zhang, Y., Wang, Y., and Shen, X. Parallelization of IBM Mambo System Simulator in Functional Modes. *ACM SIGOPS Operating Systems Review*, 42, 1 (2008). 71-76.

[17] Lantz, R.E. Fast Functional Simulation with Parallel Embra. In *Proceedings of Workshop on Modeling, Benchmarking, and Simulation (MoBS)*, 2008.

[18] Hung, S.-H., Shih C.-S.,Kuo T.-W, Tu C.-H., Tu, and Chang C.-W. A Real-Time, Energy-Efficient System Software Suite for Heterogeneous Multicore Platforms. In *CODES+ISSS*, 23-32, 2012.

[19] Stone, J. E., Gohara, D., and Shi, G. OpenCL: a parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*. 12, 3 (2010), 66–73.

[20] Collange, S., Daumas, M., Defour, D., and Parello, D. 2010. Barra: a parallel functional simulator for GPGPU. In *Proc. IEEE Intl. Symp. on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*. 351–360.

[21] August, D., Chang, J., Girbal, S., Gracia-Perez, D.,Mouchard, G., Penry, D., Temam, O., and Vachharajani, N. UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development. *IEEE CAL*, 6, 2(2007), 45-48.

[22] Diamos, G. F., Kerr, A. R., Yalamanchili, S., and Clark, N. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *PACT,* 353–364, 2010.

[23] Zakharenko, V., Aamodt, T., and Moshovos, A. Characterizing the performance benefits of fused CPU/GPU systems using FusionSim. In *DATE,* 685–688, 2013.

[24] Yourst, M. T. PTLsim: a cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS,* 23–24, 2007.

[25] Shen, B.-Y., Chen, J.-Y., Hsu, W,-C., and Yang, W. LLBT: An LLVM-Based Static Binary Translator. In *CASES,* 51-60, 2012.

[26] Perez, G. A., Kao, C.-M., Hsu,W.-C.,and Chung, Y.-C. A Hybrid Just-In-Time Compiler for Android. In *CASES,* 41-50, 2012.

[27] Smith, J. E., and Nair, R.. *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.