HSAemu 2.0: Full System Emulation for HSA platforms with Soft-MMU

Hao-Che Hsu
Computer Science
National Tsing Hua University
Hsinchu, Taiwan
joshsyu@sslab.cs.nthu.edu.tw

Chih-Wei Yeh
Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan
medicinehy@gmail.com

Shih-Hao Hung
Computer Science
National Taiwan University
Taipei, Taiwan
hungsh@csie.ntu.edu.tw

Wei-Chung Hsu
Computer Science
National Taiwan University
Taipei, Taiwan
hsuwc@csie.ntu.edu.tw

Chung-Ta King
Computer Science
National Tsing Hua University
Hsinchu, Taiwan
king@cs.nthu.edu.tw

Yeh-Ching Chung
Computer Science
National Tsing Hua University
Hsinchu, Taiwan
ychung@cs.nthu.edu.tw

ABSTRACT

With the increasing computing complexity and the proliferation of data, the world demands efficient, next-generation system architecture to enable large-scale applications at acceptable costs. Heterogeneous computing has become a hot topic and a solution to achieve the goals of high performance and efficient power consumption, especially when graphical processing units (GPU's) are constantly integrated into systems-on-chips (SoC's) and are widely used for mobile devices. Heterogeneous System Architecture (HSA) is a series of standards provided by the HSA Foundation and designed to support heterogeneous computing, including runtime software and hardware specifications. To support the development and optimization of HSA-compliant systems and applications, we developed a full-system emulator, called HSAemu 2.0, which meets the latest HSA 1.0 system specifications and supports application development with OpenCL 2.0 features, such as shared virtual memory, device enqueue and pipe. As a hardware/software co-design tool, HSAemu 2.0 not only supports the development of heterogeneous applications, but also assists system vendors in designing and evaluating the HSA runtime libraries, HSAIL compiler, and HSA hardware.

CCS Concepts

•**Computer systems organization** → **Heterogeneous (hybrid) systems;**

Keywords

HSA; OpenCL; Heterogeneous Computing; Emulation

RACS '16 , October 11-14, 2016, Odense, Denmark

DOI: http://dx.doi.org/10.1145/2987386.2987431

1. INTRODUCTION

Multi-cores have flourished as the speed increase of single processor cores has encountered bottlenecks such as heat issues when the clock frequency increases [15]. Even with multi-cores, the world demands better computational efficiency in solving large problems, and many recent research and development works have moved intensive computational CPU tasks onto the other types of computing devices, such as GPU [14], and FPGA [19], for higher performance and/or better power efficiency. Such a concept of *heterogeneous computing* basically advocates the use of a variety of processor architectures to improve the performance/efficiency of a system in executing applications, as opposed to only utilizing the same processor architecture in traditional multi-core systems [16].

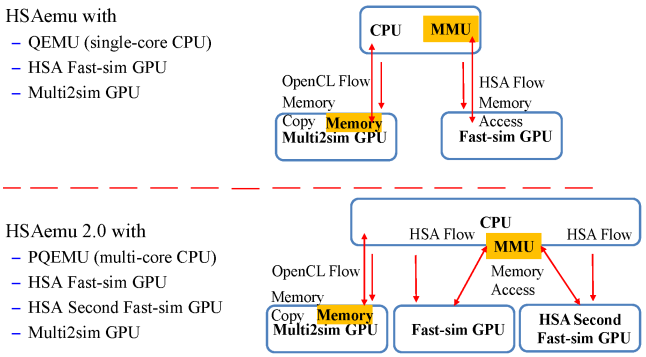

Figure 1: Enhancements of HSAemu 2.0

However, while heterogeneous computing is attractive, the portability of an application would suffer if it is designed for a specific heterogeneous system, as heterogeneity introduces different application programming interfaces (API's), application binary interfaces (ABI's) and instruction set architecture (ISA's). Thus, a set of specifications, called *Heterogeneous System Architecture* (HSA) were proposed by the HSA Foundation, with a goal to offer an industrial standard for heterogeneous computing. The HSA foundation is an industry standard organization that consists of some major

companies such as AMD, ARM, Qualcomm, MediaTek, etc, and serves as a consortium to develop device-agnostic runtime software and hardware interfaces for a wide range of applications, from embedded systems to supercomputers.

The HSA Foundation has released three standards for architectrure [9], programmer's reference [7], and runtime software [8]. Following these standards, we have developed a full-system emulator for HSA, called HSAemu. The purpose of HSAemu is to support the development of HSA-compliant applications and help explore the design of HSA systems. We have released HSAemu 1.0 [5] as the HSA specifications were being drafted in 2014. Since then, the first official HSA specification (1.0) were finalized in 2015, and OpenCL 2.0 [12] has gained some momentum. At the same time, it also becomes a common practice that multiple GPU devices are plugged into a system as a way to increase the throughput. This paper introduces HSAemu 2.0, our latest release which meets the latest HSA standards, as shown in Figure 1, it supports the shared memory features offered by OpenCL 2.0, and is capable of emulating multiple GPU devices simultaneously. We spent some significant efforts to keep HSAemu up with the latest trends, which are to be described and evaluated with experimental results in this paper.

The remainder of this paper is organized as follows: Section II describes works related to emulators, compilers and heterogeneous computing. Section III briefly introduces HSA standards and the major concepts. Section IV discusses the architecture and design of HSAemu 2.0 discusses the architecture and design of HSAemu 2.0. Section V presents our latest experimental results on HSAemu 2.0 and discusses the difference from HSAemu 1.0. The final section concludes this work and describe potential future works.

2. BACKGROUND AND RELATED WORKS

For heterogeneous computing, there are popular programming languages, such as CUDA and OpenCL.NVIDIA's CUDA (Compute Unified Device Architecture) provides a full tool suite for offloading application kernels onto GPU devices, but it is difficult to port CUDA applications to another heterogeneous platform as CUDA is limited to NVIDIA GPU devices so far. On the other hand, OpenCL (Open Computing language) is an open, royalty-free standard for cross-platform, parallel programming of a variety of processors [12][13]. Although OpenCL is an open standard and supported by several hardware vendors, including NVIDIA, it takes time for the vendors to implement the software runtime and hardware support for OpenCL. In contrast to OpenCL, HSA defines not only the language and the API's, it also defines the software infrastructure and hardware specifications to reduce the efforts in support the standard [8]. In fact, it has been shown that several programming languages, including OpenCL, have been implemented on the top of HSA.

The HSA Foundation created a project [6] to build a series of tool chains for the HSA system, including HSA runtime, HSAIL compiler, HSA finalizer and HSAIL instruction simulator for testing execution of HSAIL Brig files. We work closely with the HSA Foundation in developing the first full-system emulator for HSA, HSAemu [5].

Although most of the HSA software infrastructure is open-source, the vendor still needs to implement its own compiler infrastructure, since the instruction set architecture (ISA) used in a heterogeneous system is vendor-specific. Instead, our HSAemu utilizes the LLVM compiler framework to compile source codes into the binary for the target processor. LLVM can primitively translate source codes into LLVM IR and then retargets the LLVM IR to various ISAs by using the LLVM compiler infrastructure. Since HSA defines its own intermediate language called HSAIL [7] with some new features which are not yet supported by LLVM, we had to modify the LLVM to meet HSA requirements by adding the support of HSAIL to the LLVM.

Emulators and simulators are often used in designing new systems, as a way to verify or evaluate hardware architecture design or test applications. Full-system simulators, such as QEMU [2], can simulate a full system with a set of virtual hardware resources to execute a full-blown software stack including operating system at the binary level. Unlike cycle-accurate simulators, QEMU is a functional-accurate emulator, which only emulates processor cores, peripheral devices, memories, and network connections without modeling the micro-architectural or circuit-level details. On the contrary, QEMU can emulate many architectures, including x86, ARM, MIPS, etc., with dynamic binary translation techniques and offers much higher execution speed than cycle-accurate simulators do. Furthermore, parallel versions of QEMU, e.g. PQEMU [4] and COREMU [18], allow multiple CPU cores to be emulated concurrently by parallel emulator threads to take advantage of the multi-core processing capability on the host machine.

For the design of GPGPU, GPGPUsim [1] and Multi2sim [17] are two of the most popular simulation tools. GPGPUsim provides a detailed simulation of a GPU for running CUDA and OpenCL workloads with an integrated energy model at the microarchitecture level. Similarly, Multi2sim provides an OpenCL system library that runs OpenCL applications on simulated GPU devices. For our case study, we chose to hook Multi2sim to our HSA emulator as the code of Multi2sim is easier for us to port into our environment.

In comparison, gem5 [3] is a cycle-accurate full-system emulator, which provides a highly configurable simulation framework with a detailed system model. While gem5 also supports multiple ISAs including ARM, ALPHA, and x86, its execution speed is too low for software testing purposes. The gem5-gpu [11] is modifying from the gem5 project for heterogeneous computing, which combines gem5 and GPGPUsim. Unfortunately, gem5-gpu does not supports OpenCL 2.0 and HSA.

3. OVERVIEW OF THE HSA STANDARDS

By coupling different types of processors, HSA seeks to improve *programmability, performance*, and *energy efficiency*. As illustrated in Figure 2, the HSA standards include a runtime library that defines APIs for high-level programming, a hardware architecture that defines features supported to meet the requirement of HSA for vendors, and a programmer's reference manual that describes HSA's intermediate language *HSAIL*. HSAIL can be converted to a generic file of binary format, namely, BRIG.

A HSA application is comprised of two parts: agent and kernel. An agent can dispatch tasks onto another compute device, for example, the CPU typically executes the agent code and dispatches tasks to GPU devices. As a feature of HSA, a GPU device can also dispatch a task to the CPU if it determines the CPU is more suitable for that task. The kernel code is the data-parallel code that typically executes

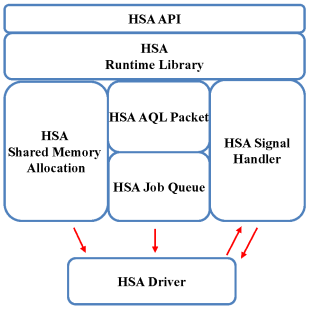

Figure 2: Overview of the HSA Runtime

on a GPU device, but CPU can also execute the kernel code if needed. In the following HSAIL and BRIG subsection, we will describe how the kernel code becomes executable for the target device.

3.1 Queues and AQL Packets

A queue is a runtime-allocated resource that contains a packet buffer. This queue is controlled by a packet processor, which enqueues, dequeues and resolves dependencies by tracking states of packets. As shown in Figure 2, all AQL (Architected Queuing Language) packets will be placed into a queue owned by a kernel agent. HSA provides this special feature to enable applications to launch kernels by enqueuing packets. When a packet is enqueued the packet processor will inform the respective device for which the packet is assigned, if there is no dependency. With AQL packets and command queues, applications can easily be dispatched on any agents. HSA defines three types of AQL packets:

- The *kernel dispatch packet* contains a pointer to the executable kernel, kernel arguments, and kernel launch information. The kernel launch information includes launch dimensions, work groups, and private memory.
- The *agent dispatch packet* is used to launch built-in functions in agents. For example, a compute-unit running a kernel can ask the host application to allocate memory on its behalf.
- The *barrier-AND packet* allows an application to specify up to five signal dependencies and requires the packet processor to resolve all dependencies before proceeding. The barrier-OR is similar to the barrier-AND differing in that if any dependency is resolved, then it can proceed.

3.2 Signals

A signal is a runtime-allocated object used for communications between agents in a HSA system. Using signals, each agent in HSA system can inform other agents of their respective state transitions. A kernel agent can also monitor the signal and check if the computing device is finished or set a maximum waiting time to send a signal to the computing device to terminate. Using signals as communication mechanism usually perform better in terms of power or speed than using shared memory address, thanks to synchronization management in shared virtual memory.

3.3 Shared Virtual Address

Data transfer often causes bottleneck in high performance computing. OpenCL 2.0 provides a feature called Shared Virtual Memory (SVM), which enables programmers to develop applications with extensive use of pointer-linked data structures to transmit arguments or share data between the agents. Nevertheless, support of shared memory can be a challenge due to cache coherence issues. Thus, SVM is scrutinized from Intel's perspective [10] and is categorized into three levels of synchronization: *Coarse-grained buffer*, *Fine-grained buffer*, and *Fine-grained system*. In *coarse-grained buffer*, shared space occupies a region of the system's memory, coherence occurs when the OpenCL synchronization function, *clEnqueueSVMMap*, is called. This forces programmers to ensure coherence by explicitly calling memory map functions. *Fine-grained buffer* also utilizes a portion of memory with the capability to implicitly handle the issue of coherence. Each modification on the address is valid to other agents. *Fine-grained system* has no need to handle coherence; besides, its shared space includes all system memory rather than a region. In comparison, HSA's architecture standard defines a compliant HSA system that allows agents to access shared system memory through the common HSA unified virtual address space. Each agent should handle shared virtual memory address translation through page tables managed by the HSA component, called HSA-MMU (HSA Memory Management Unit). The HSA-MMU allows memory operations to easily be handled by every agent regardless of coherence or overhead. HSA's Shared memory inherently handles coherence for all agents and ensures that memory protection mechanisms cannot be circumvented. Thus, shared virtual memory is perjhaps the most important feature for the HSA platform, as SVM not only reduces communication overhead but simplifies flows of programs to enhance programmability.

3.4 HSAIL and BRIG

HSA is designed to support multiple hardware, therein encompassing multiple instruction set architectures (ISA's), which allows HSA to be *hardware-agnostic*. A HSA program is compiled to HSAIL (Heterogeneous System Architecture Intermediate Language) as an abstraction of the native instruction set. The abstracted layer is then translated into the appropriate native machine code for the target device by using the finalizer, if the hardware cannot support HSAIL natively. Upon application execution, the *loader* loads the executable file onto the computing device. The life cycle of an application can be divided into three stages: *finalization*, *loading*, and *execution*.

In the finalization stage, the *finalizer* generates code for a specific kernel agent instruction set. Then the loader manages the allocation of global and read-only segment variables and subsequently moves the finalized code onto the specific kernel agent. Lastly, it generates packets that will be executed on a kernel agent at runtime.

4. HSAEMU 2.0

HSAemu 2.0 is the lastest update of HSAemu with the support of OpenCL 2.0 and HSA 1.0 standards, as well as multiple GPU devices. The new features include shared virtual memory, image processing, device enqueue and pipe. Applications can take advantage of these features to improve performance, which was not possible in HSAemu 1.0. HSAemu 2.0 also develops a mechanism that handles shared

virtual memory translation called HSA Soft-MMU (HSA Software Memory Management Unit) that simulates memory operations in detail, thereby approaching hardware design. As illustrated in Figure 3, HSAemu 2.0 mainly consists of three layers, and the remainder of this section presents the components in these layers:

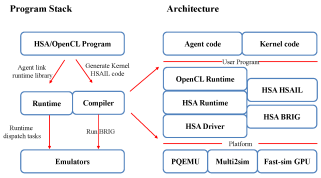

Figure 3: Overview of HSAemu 2.0

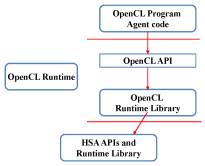

Figure 4: Support for OpenCL 2.0

- The *application layer* that consists of agent code and kernel code, which can be an OpenCL or a HSA program.
- The *communication layer* that consists of compiler, runtime and driver. The compiler generates kernel HSAIL code and then converts HSAIL into native binary. The runtime library combines OpenCL 2.0 runtime and HSA runtime for which programming libraries are provided. The driver provides an interface between emulators and applications and transmits interrupt signals.
- The *simulation platform* that contains four emulators that can serve as compute devices, namely, PQEMU, Multi2sim and Fast-sim GPU. As described in Section 2, PQEMU is a parallel QEMU that simulates CPU, and Multi2sim is a cycle-accurate GPU emulator. Fast-sim GPU is included by HSAemu to serve as a fast function-accurate GPU emulator, which is several orders of magnitudes faster than Multi2sim and is designed for software testing purposes. Programmers can select which GPU device they want to use when creating the context and command. By utilizing the HSA MMU, it is not only to reduce the overhead of all global memory access between the GPU devices and CPU but also makes it easier to hook the new computing device.

4.1 OpenCL 2.0 Runtime

As shown in Figure 4, the agent part of the OpenCL application runs on the CPU side and links essential libraries, including the OpenCL library. Following the OpenCL 2.0 specification, we implement our own *OpenCL 2.0 runtime library* that supports new features. When an OpenCL API function is called, the function links to our OpenCL runtime library and is then forwarded downstream to call to the HSA runtime, which then packets the information of the executing OpenCL functions in an AQL format and sends it to HSA packet processors. Most of OpenCL functions can be packeted as AQL packets and sent to HSA packet processors but functions that need hardware support are redirected to the respective hardware using interrupts. These functions are used in features like shared virtual memory, pipe and image processing. To build a shared virtual memory environment, we pass the addresses of global arguments, that will be used in the OpenCL kernel function, to the HSA MMU that handles memory access. The feature, pipe, in addition needs to be passed as a command to tell the emulator to prepare a special memory object. For image processing, OpenCL 2.0 defines some image APIs that need hardware support, so an image processing API may be sent as a HSA command to the emulator to request hardware information. Table I shows OpenCL 2.0 API's and their corresponding to HSA commands

Table 1: OpenCL 2.0 APIs versus HSA Commands

OpenCL 2.0 API's	HSA Commands
Shared Virtual Memory	
clSetKernelArg	hsa_abi_svm_alloc
Pipe	
clCreatePipe	hsa_create_pipe
Image processing	
clCreateImage	hsa_abi_image_info

4.2 HSA 1.0 Runtime

Our HSA runtime is implemented according to HSA Runtime Specification 1.0. HSAemu 2.0 provides full compatible HSA APIs that programmers can either directly use to design applications or OpenCL runtime calls. As shown in Figure 3, HSA runtime mainly focuses on following specification components: *HSA Shared Memory Allocation*, *HSA AQL Packets*, and *HSA Signal Handler*.

HSA runtime stores the addresses of arguments containing the global specifier in the kernel function and then sends a shared memory allocation command to the HSA emulator. Computing devices may then access this memory address. Most information of tasks will be populated into the packets using the AQL format and then enqueued. A packet processor will dispatch packets to the target device later. HSA signal handler sends a signal to the device to start computing and receives a signal upon completion from the device.

4.3 HSA MMU

HSA memory management unit (MMU) is an important component in HSAemu 2.0. The whole global memory allocation, access and operations are handled by this unit, and we use a thread, called *HSA_MMU*, to emulate the MMU of a HSA system. As illustrated in Figure 5, when PQEMU boots, HSA_MMU is initialized and keeps monitoring its task queue. HSA Runtime sends a command to PQEMU while passing kernel arguments, and then HSA_MMU is informed to handle this task, where the addresses of kernel arguments are recorded in the HSA SVM table. While the kernel program asks to access an address, it will jump to *hsa_svm_ld* or *hsa_svm_st* helper function that is implemented in the emulator to read or store data. The read/write functions get the virtual address from kernel program and then asks HSA_MMU for its physical address. Under mutex mechanism, HSA_MMU prevents race conditions, which also meets requirements of fine-grain buffer memory that has map-free features by directly reading/writing from physical address that are visible to all devices and support atomic operations.

4.4 The PQEMU CPU Emulator

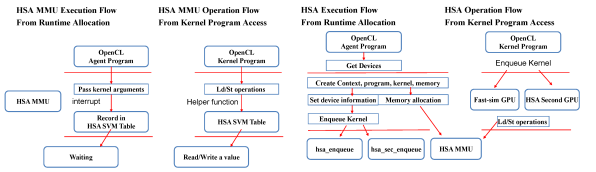

Figure 5: Design of HSA MMU

Figure 6: Support of multiple GPU Devices

In HSAemu 2.0, PQEMU serves to emulate multi-core CPU on a multicore host and is responsible for booting the operating system, handling the interrupts from the HSA driver, and communicating with other HSA devices. PQEMU is based on QEMU 1.7, which uses round robin (RR) mechanism to simulate the CPU core sequentially. Thus, if there is a CPU that has dual cores, QEMU only uses one thread to emulate this CPU. To improve the speed of emulation, PQEMU parallelizes the dynamic binary translation (DBT) process for each virtual CPU. The PQEMU adopted in HSAemu 2.0 uses the unified code cache (UCC) model, which uses thread lock to handle synchronization between virtual CPUs and shares most components in the DBT process. Compared to separate code cache (SCC), UCC may be slower but it is easier to implement and has less duplication of components.

4.5 The Multi2sim GPU Emulator

HSAemu 2.0 provides a friendly interface for the user to plug in GPU emulators, and Multi2sim is one of the GPU emulators that have been integrated into HSAemu for cycle-accurate simulation. Multi2sim is capable of simulating a variety of architectures, and we choose the AMD Southern Island series GPU as the computing device in our case study. When PQEMU initializes devices, Multi2sim starts its processes and waits for tasks. Multi2sim also provides its compiler (M2C) for translating an OpenCL kernel program into executable file for Southern Island GPU. When an OpenCL program is dispatched from runtime level, Multi2sim will be informed by the packet processor and then prepares execution environment including memory allocation, numbers of work items, etc. After receiving HSA computing signal, Multi2sim starts loading binary of the kernel program and performs computation. When the kernel program finishes, Multi2sim shows profiling information that is produced by its timing model.

4.6 The HSA Fast-sim GPU Emulator

Fast-sim GPU is a configurable and programmable functional GPU emulator which provides a simple, generic GPU model to emulate GPU's behavior and develop HSA architecture. Fast-sim GPU can be set to emulate a number of computing units, where each computing unit can be executed by a thread to take advantage of the processing capability of the host. Note that the work items in a work group is executed in a round-robin fashion by a single thread, where the barrier instruction within a work group may not be emulated correctly. In that case, the user should use Multi2sim to simulate the GPU. As a functional emulator, Fast-sim does not guarantee timing accuracy and is not suitable for detecting timing-dependent bugs.

4.7 Multiple GPU Emulation Support

HSAemu 2.0 allows the HSA Fast-sim GPU emulator to be executed in parallel to emulate multiple GPU devices the system. Let us use a two-GPU case to illustrate how HSAemu 2.0 makes it possible, As illustrated in Figure 6, there are two Fast-sim GPU emulator instances, and each GPU emulator has a group of threads that represents as the GPU's computing units (CUs) and uses *__thread* identifier to record each computing unit's information including id, group size, finish state, etc. In addition, these two GPU emulators are supported by the HSAemu 2.0 compiler and runtime library. By configuring the Fast-sim GPU and HSA Second GPU as computing devices, programmers can utilize these two GPU emulators at the same time.

4.8 Compiler

HSAemu 2.0 provides the OpenCL 2.0 enabled HSA compiler by modifying LLVM 3.4 to compile a kernel program into HSAIL codes and then generate the BIRG file for the target device. We first use Clang 3.4 front-end to translate kernel program into LLVM IR, an immediate language for LLVM, and then follow the definition of HSAIL [7]. After generating HSAIL codes, the compiler's back-end, i.e. the finalizer, can translate HSAIL code into an executable file for the target device by decoding and converting those HSAIL instructions to native binary code. Once compiled, a HSA or OpenCL kernel program can be dispatched to the target device in HSAemu 2.0.

5. EXPERIMENTAL RESULTS

The first part of our experiments, as discussed in Section 5.1, is to prove that HSAemu 2.0 is compliant with HSA and OpenCL 2.0, we use benchmarks provided by AMD SDK to verify the correctness of the platform and simulate the execution flow. The second part, as discussed in Section 5.2, carried out experiments to compare the performance of HSAemu 2.0 with HSA MMU and without HSA MMU by using the *NBody* benchmark provided by AMD SDK 2.9. Finally, in Section 5.3, we show the performance scalability with two Fast-sim GPU emulators enables, where each GPU has 8 computing units, and each computing unit's work group size is 256. The experimental environment is showed in Table 2.

Table 2: Experimental Environment

Host	
CPU	Intel(R) Xeon(R) CPUX7550
Frequency	2.0GHz
Physical Processors	4
Sibling	16
CPU Cores	8
Virtual Processors	2
Total thread	64
RAM	128G
OS	Ubuntu 14.04 Linux 3.13

Guest	
QEMU	1.7
CPU	ARM Cortex-A9
Machine Board	Vexpress-A9
CPU cores	4
SMP	4
RAM	1G
OS	Linaro Ubuntu 14.10 Linux 3.10
PQEMU	enable
HSA Second GPU	enable

5.1 OpenCL 2.0 Features

HSAemu 2.0 is validated with AMD SDK 2.9 and AMD SDK 3.0. Among the 29 benchmarks that We have successfully executed, 20 are written in OpenCL 1.2, 5 uses OpenCL 2.0 API's, and the remaining benchmarks are mainly for image processing. As mentioned in Section 4.1, OpenCL 2.0 features requires corresponding support from the HSA runtime and emulator. Give that the HSA Fast-sim GPU is

capable of support most of the features listed in Table 1, it is capable of exeucting all the OpenCL 2.0 benchmarks. On the other hand, Multi2sim can successfully execute 23 benchmarks, because Multi2sim's compiler (M2C) does not support some of OpenCL 2.0 features so far.

5.2 Software MMU

The following experiments use the NBody benchmark to show that the inclusion of HSA MMU is able to improve the performance of HSAemu 2.0. First, as shown in Figure **??**, we measure the performance of HSAemu 2.0 with HSA MMU and compare it to the one without HSA MMU, as the numbers of the computing body in the benchmark increases from 1024 to 2048. The results show with HSA MMU, the HSAemu performs better than without HSA MMU, especially when the numbers of bodies is large. This experiment indicates that if the HSA MMU handles memory access between the CPU and GPU, the overhead of the address translation is less than directly using the CPU MMU.

5.3 Multiple GPUs

To illustrate the capability of emulating two GPU deivces in HSAemu 2.0, we run the Nbody benchmark by using SVM mechanism to reduce the data transfer from CPU to GPU devices. We fix the numbers of the computing body to 1024 and repeat the Nbody benchmark for 10, 20, 30 and 40 times. As shown in Figure 7, it is obvious that running the Nbody benchmark on two GPU devices reduces the total exeuction time on the HSAemu, because the HSAemu is capable of emulating the GPU devices with more threads. In this case, our host machines has 32 physical processor cores and 64 virtual cores, which significantly improves emulation speed.

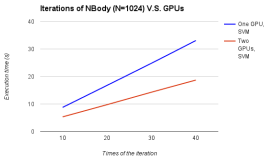

Figure 7: Performance scalability with multiple GPU emulators.

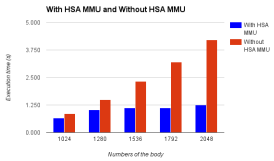

Figure 8: Speed of HSAemu versus HSA MMU

6. CONCLUSION AND FUTURE WORKS

Heterogeneous computing has become important, and HSA provides a viable solution for high programmability, high performance and less power consumption. To drive this new architecture further, we have continued to develop HSAemu to emulate HSA behavior more accurately and provide a platform for developing HSA software. In the near future, we will use HSAemu to search the hardware-software co-design space for emerging application areas such as machine such as machine learning, cloud computing, and even portable devices that may be benefited with heterogeneous designs. We hope this effort of developing a full-system emulator can contribute to heterogeneous computing as a useful hardware/software co-design tool.

7. ACKNOWLEDGMENTS

This research was supported by MediaTek, Taiwan under the grant MTKC-2016-0264 and Ministry of Science and Technology (MOST), Taiwan under the grant No.105-2218-E-007-001 and 104-2622-8-002-002.

8. REFERENCES

[1] A. Bakhoda, G. L., W. W. Fung, H. Wong, and T. M. Aamdot. Analyzing cuda workloads using a detailed gpu simulator. pages 163–174, 2009.

[2] F. Bellard. Qemu, a fast and portable dynamic translator. *USENIX ATC*, pages 41–46, 2005.

[3] N. Binkert, B. Bechmann, G. Black, S. K. Reinhardt, et al. The gem5 simulator. *ACM SIGRACH Computer Architecture News*, 39:1–7, 2011.

[4] J. Ding, P. Chang, W. Hsu, and Y. Chung. PQEMU: a parallel system emulator based on QEMU. *ICPADS*, pages 276–283, 2004.

[5] J.-H. Ding, W.-C. Hsu, B.-C. Jeng, S. H. Hung, and Y.-C. Chung. HSAemu - a full system emulator for HSA platforms. *Hardware/Software Codesign and System Synthesis(CODES+ISSS)*, pages 1–10, 2014.

[6] H. Foundation. *HSA Foundation Creating New Solutions with Heterogeneous Computing.*

[7] H. Foundation. *HSA Programmer Reference Manual Specification 1.01*, 2015.

[8] H. Foundation. *HSA Runtime Specification 1.0*, 2015.

[9] H. Foundation. *HSA System Architecture Specification 1.0*, 2015.

[10] Intel. *OpenCL 2.0 Shared Virtual Memory Overview*, 2014.

[11] H. J., O. M.S., H. M. D., and W. D.A. gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters*, 14:34–36, 2014.

[12] G. Khronos. *OpenCL 2.0 API Specification*, July 2015.

[13] G. Khronos. *OpenCL 2.0 C Language Specification*, Spetember 2015.

[14] J. Nickolls and W. J. Dally. The gpu computing era. *IEEE Micro*, pages 56–59, 2010.

[15] P.E.Ross. Why cpu frequency stalled. *IEEE Specturm*, 45:72, 2008.

[16] G. Teodoro, R.Scachetto, O.Sertel, and M. N. Gurcan. Coordinating the use of gpu and cpu for improving performance of compute intensive applications. *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–10, 2009.

[17] R. Ubal, J. Sahuquillo, S. Petit, and P. Lopez. Multi2sim: a simulation framework to evaluate multicore-multithread processors. *HPCA*, pages 62–68, 2007.

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

[19] W.Tang, B.Duan, and C. Zhang. Accelerating millions of short reads mapping on a heterogeneous architecture with fpga accelerator. *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 184–187, 2012.