An OpenCL Runtime System for a Heterogeneous Many-Core Virtual Platform

Kuan-Chung Chen and Chung-Ho Chen
Inst. of Computer & Communication Engineering, National Cheng Kung University, Tainan, Taiwan, R.O.C.
edi@casmail.ee.ncku.edu.tw, chchen@mail.ncku.edu.tw

***Abstract*—We present a many-core full system simulation platform and its OpenCL runtime system. The OpenCL runtime system includes an on-the-fly compiler and resource manager for the ARM-based many-core platform. Using this platform, we evaluate approaches of work-item scheduling and memory management in OpenCL memory hierarchy. Our experimental results show that scheduling work-items on a many-core system using general purpose RISC CPU should avoid per work-item context switching. Data deployment and work-item coalescing are the two keys for significant speedup.**

Keywords—OpenCL; runtime system; work-item coalescing; full system simulation; heterogeneous integration;

I. Introduction

OpenCL is a data parallel programming model introduced for heterogeneous system architecture [1] which may include CPUs, GPUs, or other accelerator devices. Through the OpenCL application programming interface, a programmer can offload the parallel task to the target devices by programming the control thread and the parallel kernel codes. Performance of the underlying hardware architecture is significantly affected by the efficiency of the OpenCL runtime system and program code itself.

Several previous works have implemented the OpenCL framework on existing multi-core architectures such as Intel single-chip cloud computer, IBM cell BE, and Intel Nehalem. Twin peaks proposes a work-item context switch method with low-level optimizations on the Intel Nehalem and AMD Istanbul system [4]. Jaejin Lee et al. proposed a work-item coalescing technology to reduce the context switch overheads in *multiple instruction, multiple data* CPU architecture [2][3]. Dan Connors et al. have introduced a FPGA-based MPSoC system with OpenCL runtime supporting [7]. Raphael Poss et al. proposed a heterogeneous integration methodology which offloads the heavy system calls to the host operating system [5]. Ali Bakhoda et al. implemented the GPGPU-sim that exploits the host operating system for CUDA or OpenCL runtime execution and uses the ptx-simulator to execute CUDA kernel thread [6].

To examine the interplay of an interested target platform and its runtime system, a flexible ESL (Electronic System Level) simulation platform is useful for architecture exploration and runtime system design. In this paper, we present an ARM-based many-core simulation platform and its OpenCL runtime system. To improve simulation efficiency, we propose a full system simulation platform based on a heterogeneous integration approach where the OpenCL runtime is executed on a Linux PC while the kernel code execution is simulated on an ESL many-core system.

This work was sponsored and supported in part by the National Science Council of Taiwan under grand NSC 102-2221-E-006 -272 -

The rest of this paper consists of the following sections. In Section 2, we briefly introduce the OpenCL execution model and our heterogeneous full system simulation platform. Section 3 discusses the OpenCL runtime implementation in detail and its enhancements. Section 4 presents the evaluation system and results. We conclude this paper in Section 5.

II. OpenCL Model and Virtual Platform

In this section, we first introduce the OpenCL programming model and followed by our SystemC-based virtual platform architecture.

A. OpenCL Platform Overview

Fig. 1 shows a conceptual OpenCL platform which has two execution components: a host processor and the compute devices. The host processor executes the control thread and, via OpenCL APIs, dispatches the tasks to the compute devices, which in our case are the many-core system. A kernel thread in OpenCL is referred to as a work-item. A work group consists of a various number of work-items, determined by the programmer. A work group of kernel threads can be mapped onto a compute unit, a processor core for example, either by the OpenCL runtime or by a specific hardware thread scheduler.

Besides the execution model, OpenCL also well defines the memory source hierarchy and its programming keywords (*__global, __local, __constant, and __private*). A kernel thread accesses these four distinct memory regions with the pointer variable declared in the kernel source code. OpenCL takes the relaxed consistency memory model. The visible memory region of kernel threads in a work-group is guaranteed to be consistent whenever they reach the barrier function. In addition, there are no memory synchronization mechanism between work-groups.

B. Heterogeneous Full System Simulation Platform

In order to design and verify a many-core OpenCL runtime system, we implement a many-core simulation platform based on the ARMv7a instruction set simulator with transactional level bus connections. Furthermore, we apply this full system simulation platform to analyze the integration between the runtime system and the many-core hardware execution environment.

To provide an intact OpenCL execution environment, the host processor has to run an operating system which provides the device drivers and the OpenCL runtime. In this case, the simulation overheads are likely to hinder the system

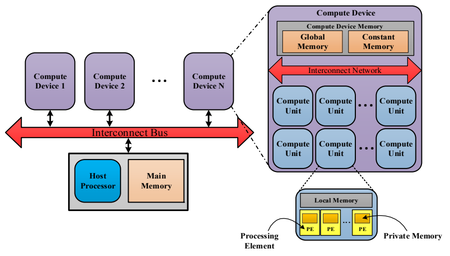


Fig. 1. OpenCL conceptual platform model

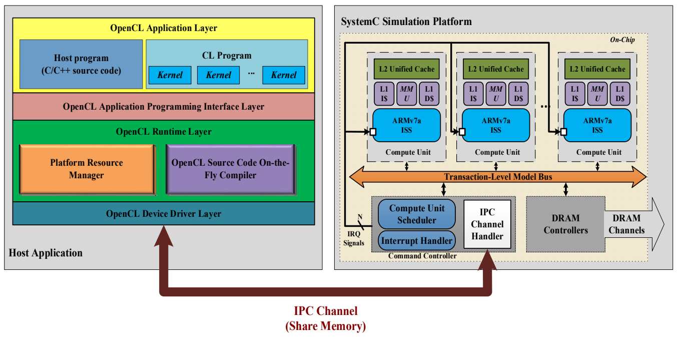


Fig. 2. OpenCL System Simulation Platform

verification and analysis tasks. Since we are interested in the kernel thread execution efficiency in the many-core system, we employ a heterogeneous simulation methodology to improve the simulation speed for architecture exploration. The virtual platform only emulates the many-core system while OpenCL runtime system and operating system are offloaded onto a Linux PC. Specifically, we implement the OpenCL runtime system as a user process which runs on the Linux operating system. And this OpenCL runtime connects with our many-core virtual platform through the shared memory inter-process communication (IPC) mechanism.

Fig 2 depicts the overview of our heterogeneous simulation platform. The virtual platform emulates a compute device in the OpenCL conceptual platform model. The compute device has various number of compute units, each of which is mapped to an ARM core. A compute unit is an ARM-based instruction set simulator with a memory management unit (MMU), private L1 I/D caches, and a L2 unified cache. The ARM cores, up to 32, are connected with a Transaction-Level Model (TLM) interconnection which models the arbitration for memory controllers and data transfer with fixed cycle delay. There are at most four DRAM controllers in our simulation platform. Each DRAM controller can access up to 4GB memory region by setting the simulation parameter. This implies that at most four memory accesses can be proceeded in parallel.

A typical OpenCL runtime system supports three command types: kernel, memory, and event. A kernel command is used for assigning work-groups to the compute devices; a memory command is used for data transfer among the memory space of the host processor and the compute devices. An event command is used for controlling the execution sequence between commands. Since the OpenCL runtime and the control thread are both processes run on the host Linux, a command controller in the many-core virtual platform is required to receive and interpret the commands from the two processes. The command controller has implemented three major functions: compute unit scheduler, interrupt handler, and IPC channel handler.

Whenever the OpenCL runtime receives a kernel or memory command requested from the application, the device driver passes the command to the command controller via the IPC channel. The IPC channel handler decodes the command type and takes the corresponding action as follows. When it is a kernel command, the compute unit scheduler serves this command by appointing the ARM cores to execute the work-groups of kernel codes. When it is a memory command from a control thread, the command controller accesses the device DRAM module through the TLM interconnection directly and finishes the memory request. Finally, an event command is only handled inside the OpenCL runtime system.

III. OpenCL Runtime Implementation and Enhancement

In this section, we describe the detailed implementation of our OpenCL runtime. Since we choose the ARM core as the compute-unit, kernel thread emulation and memory management are two main challenges and will be discussed in this section.

A. *OpenCL Runtime Overview*

In order to explain the OpenCL execution flow, we divide the host application into four layers as shown in Fig. 2. An OpenCL application includes a host program and kernel codes that are to execute on compute devices. The host program manages the command queues and the abstract memory objects through OpenCL APIs defined in the OpenCL specification. For instance, a programmer uses the clCreateBuffer() API to create an abstract memory object for controlling the device memory on the target device.

In the OpenCL runtime layer, we have implemented two principal functions, a platform resource manager and an on-the-fly compiler. As a programmer uses OpenCL APIs to declare an abstract object such as memory object or command queue, the platform resource manager creates a specific data structure to record the relationship of the abstract objects and device resources. The platform resource manager returns a pointer of that data structure to the host application. And then the host application uses this pointer to indicate the device resources for the kernel and memory commands that follows.

Besides resource management, the OpenCL runtime has to compile the kernel source code into an executable binary of the target compute device. Fig. 3 illustrates the on-the-fly compiler system in our OpenCL runtime system. We use the LLVM framework and the GNU GCC cross-compiler for ARM processor. As shown in the figure, the LLVM compiler translates the OpenCL source code to the intermediate representation, LLVM-IR, through its front-end compiler. Then, the LLVM compiler performs the LLVM-IR to ARM assembly code translation, and last the GCC cross compiler assembles the assembly code into the binary code. The compiling flow is triggered in runtime whenever the programmer uses the clBuildProgram() API.

B. Work-item execution: context switch & work-item coalescing

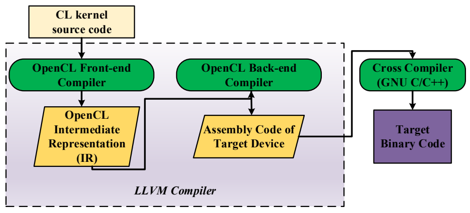
We assume that the total number of work items, i.e., the kernel threads greatly outnumbers the CPU cores as in a typical situation. In this work, we assign a work group of parallel kernel threads to one of the CPU cores in the many-core system. To perform kernel thread scheduling in a work group, we use a work-item management thread (WMT) running on each core. As shown on the left side of Fig. 4, an intuitive implementation, the WMT sequentially schedules all the work-item threads in a work group by supporting a fine-grained work-item context-switch. When a barrier function is reached, the WMT backups the CPU registers to the private memory region and then restores the next work-item's resisters. After all the work-items pass through the barrier, the WMT switches to the first work-item thread and continues the executing iteration until completing the work-group or reaching the next barrier.

However, treating each work-item as an execution thread in a general purpose CPU architecture will cause heavy context-switch overheads owing to registers backup and restore. To reduce the context switching overheads, a more attractive alternative is to combine or coalesce all work-items in a work-group into an execution thread during the compiling time. SNUCL [3] supports an OpenCL-C-to-C compiler for work-item coalescing. The OpenCL-C-to-C compiler translates the kernel source code into for-loop iterations according to the barrier function. We adopt this OpenCL-C-to-C compiler to translate kernel source code into for-loop iteration form and use the translated code for compiling into the binary code as described in Section II. As shown on the right side of Fig 4, the WMT schedules the for-loop iterations to emulate a work-group processing until reaching barrier or termination without any additional register backup and restore.

C. Memory Management

We leverage the virtual memory system to approach OpenCL memory hierarchy on a unified main memory. As shown in Fig. 5, all the work-groups scheduled by the WMTs execute on the same virtual address. Through the memory management unit (MMU), the virtual address is translated to the physical address as the WMT executes. By this manner, the private and the local memory pages of a work-group are mapped to the unique physical pages according to the execution core. In addition, the WMT setups the page table of the instruction memory before executing the work-items in a group.

Figure 6 illustrates the optimization of using private memory in exploiting multiple memory controllers. Since each core exclusively accesses their private memory region, we spread the pages of the private memory of a core in different memory channels. Furthermore, our OpenCL runtime sequentially assign a work-group to the compute unit with the core ID, so we stagger the private memory pages of the adjacent cores to the different memory channel. This ensures that the private memory accesses are scattered on all memory controllers. Finally, we arrange the pages of global memory and instruction memory to the rest of the memory space.


CL kernel source code
OpenCL Front-end Compiler
OpenCL Back-end Compiler
Cross Compiler (GNU C/C++)
OpenCL Intermediate Representation (IR)
Assembly Code of Target Device
Target Binary Code
LLVM Compiler
Fig. 3. On-the-fly Compiler System

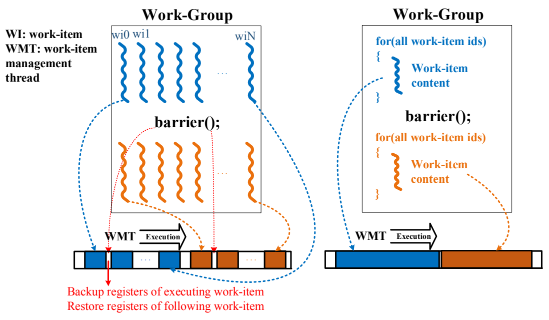


Fig. 4. Work-item emulation techniques

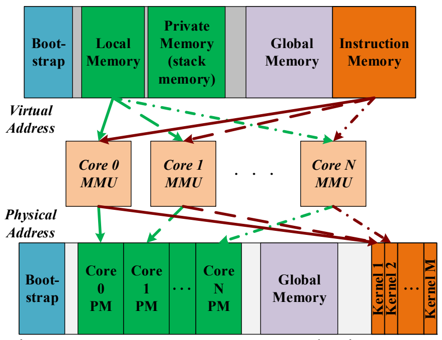


Fig. 5. Memory Management Mechanism

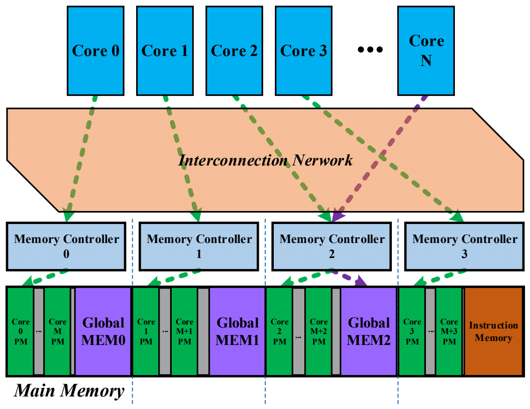


Fig. 6. Staggered Memory Mapping

D. SExperimental System and Evaluation Results

Table 1 shows the system configuration of the full system simulation platform. We implement an ARM instruction set simulator (ISS) [8] based on ARMv7a and VFP 3.0, caches also included. Since there is no synchronization between work-groups, cache coherence is not required here. The consistency of the global memory is handled by the command controller.

TABLE I. SYSTEM CONFIGURATION

Platform Component	Configuration
Virtual Platform Configuration	
No. of cores	1, 4, 8,16, 32
Processor Model	ARM ISS with VFP 3.0 @ 1.25 GHz
L1 I/D-Cache	2-way, 32B line, size 32 KB, 3 cycles delay
L2 Unified Cache	8-way, 32B line, size 256KB, 20 cycles delay
I/D-TLB	32 entries, 20 cycles miss penalty
DRAM	4 memory controllers, 2GB main memory, 110 ns access time
Interconnection	TLM bus with constant 10 cycles NoC delay
Host System	
Host Machine	Intel i7-4770 @ 3.40GHz, 16GB main memory
Operating System	OpenSUSE 12.3 64bit
Host Compiler	GCC 4.7.2
On-the-fly Compiler	LLVM-3.3, arm-elf-gcc 4.8.1 and SNUCL

E. Work-item Coalescing Enhancement

For the test suite, we choose MatrixMul, Histogram, Transpose, DCT8x8, Convolution-Separable (CS), BitonicSort and BoxFilter that are available from NVIDIA and AMD OpenCL SDK. Fig. 7 shows the speedup of work-item coalescing compared with fine-grained work-item context switching in the NoC interconnection platform. The speedup, ranging from 2 to 8, is obtained based on the simulated benchmark time for both cases.

Because the computing unit is an ARM-based core with VFP support, there are 16 regular registers, a CPSR, and 32 vector registers needed to backup or restore in the per work-item context-switch approach. On the other hand, the work-item coalescing saves the overheads of these context-switches. As an example, for the Transpose program, the kernel thread loads the target operand, reaches the barrier, and then writes back the result without any computing. This makes the ratio of context-switching instructions to the computing instructions relatively high and as a result, the work-item coalescing has outstanding performance improvement. In summary, work-item coalescing can achieve about four times faster than per work-item context-switching in the ARM-based many-core environment.

F. Scalability

Fig. 8 shows the scalability of the work-item coalescing approach in the many-core platform using NOC or bus for interconnection. The comparison basis is the single bus platform with one CPU core. In the NoC platform with four memory controllers, because we distribute the private memory pages across the available memory channels, the speedup of arithmetic benchmarks such as MatrixMul, Histogram, and CS is more linear than the single bus platform. However, BitonicSort and BoxFiler do not scale well and reach the speedup upper bound quickly. This is because each of the work-items only performs a few operations and then completes the write back for global memory where contention occurs.

IV. CONCLUSIONS

We have presented a many-core full system simulation platform and its OpenCL runtime system in this paper. By the heterogeneous integration method, we design an OpenCL runtime system, including an on-the-fly compiler and resource manager, on an ARM-based many-core virtual platform. Furthermore, we use this platform to evaluate various approaches in work-items execution and memory management. The results show that running work-items on a general purpose RISC CPU should avoid per work-item context switching. Work-item coalescing through CL code translation has achieved about four times of speedup on average. This is accomplished along with the deployment of a staggered memory mapping strategy for private memory region.

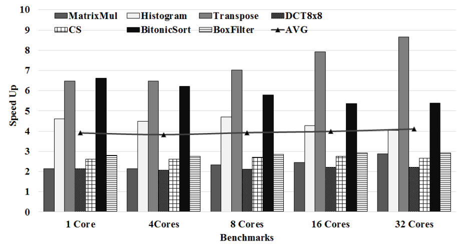


Fig. 7. Comparison of work-item coalescing and fine-grained context switch

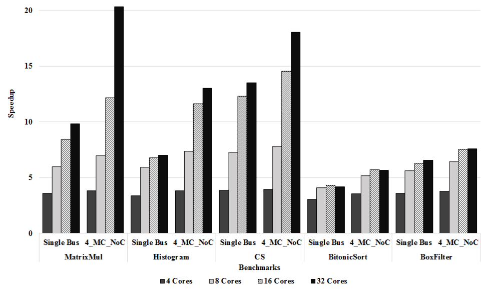


Fig. 8. Speed up between multiple memory controllers and single bus

REFERENCES

[1] Khronos OpenCL Working Group, "The OpenCL Specification Version 1.2," 2012, http://khronos.org/opencl.

[2] J. Lee, J. Kim, S. Seo, S. Kim, J. Park, H. Kim, T. T. Dao, Y. Cho, S. J. Seo, S. H. Lee, S. M. Cho, H. J. Song, S.-B. Suh, and J.-D. Choi, "An OpenCL Framework for Heterogeneous Multicores with Local Memory," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT'10*, Sep. 2010, pp. 193-204.

[3] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters," in *Proc. of the 26th ACM international conf. on Supercomputing, ICS'12*, Dec. 2012, pp. 341-352.

[4] J. Gummaraju, L. Morichetii, M. Houston, B. Sander, B. R. Gaster, and B. Zheng, "Twin Peaks: A Software Platform for Heterogeneous Computing on General-Purpose and Graphic Processors," in *Proc. of the 19th International Conf. on Parallel Architectures and Compilation Techniques, PACT'10*, Sep. 2010, pp. 205-216.

[5] R. Poss, M. Lankamp, M. I. Uddin, J. Sýkora, and L. Kafka, "Heterogeneous integration to simplify many-core architecture simulations," in *Proc. of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO'12*, Jan. 23, pp. 17-24.

[6] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *IEEE International Symp. on Performance Analysis of Systems and Software, ISPASS 2009*, Apr. 2009, pp. 163-174.

[7] D. Connors, E. Grover, and B. Caldwell, "Exploring Alternative Flexible OpenCL (FlexCL) Core Designs in FPGA-based MPSoC Systems," in *Proc. of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO'13*, Jan. 21, Article No.: 3.

[8] C.-T. Liu, K.-C. Chen, and C.-H. Chen, "CASL Hypervisor and its Virtualization Platform," in *IEEE International Symp. on Circuit and Systems, ISCAS 2013*, May, 2013, pp. 1224-1227.