The Support of an Experimental OpenCL Compiler on HSA Environments

Chun-Chieh Yang, Shao-Chung Wang, Chou-Chuan Chen and Jenq-Kuen Lee
Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan
{jet, scwang, ccchen}@pllab.cs.nthu.edu.tw
jklee@cs.nthu.edu.tw

Abstract—In recent years, with the increasing computing power and programmability on GPU, GPU has become an important role on hardware accelerator. Heterogeneous System Architecture (HSA) announced by HSA Foundation is an approach to benefit both CPUs and GPUs advantages. Open Computing Language (OpenCL) is one of the well-known programming frameworks for parallel computing on heterogeneous architecture. In this paper, an OpenCL framework is designed and implemented on HSA platform. An OpenCL compiler for HSA uses low-level virtual machine (LLVM) and an OpenCL runtime extends Portable Computing Language (PoCL) framework are built. PoCL is a portable OpenCL implementation for different parallel hardwares. Furthermore, HSA-related tools released by HSA Foundation are also integrated in our framework. Experimental results indicate that our framework provides enough features to support for advanced research.

Keywords: HSA, OpenCL, Compiler, Runtime, LLVM

1. Introduction

To satisfy the performance and energy consumption constraints, heterogeneous multi-core architecture has become the popular design on many devices. Especially, GPUs has become a popular hardware accelerator with increasing programmability and computing power. Many applications are benefited from parallel execution, such as medical image [1], bioinformatics [2], and fluid dynamics [3].

Recently, Heterogeneous System Architecture (HSA) has been widely discussed [4], [5]. HSA is announced by HSA Foundation which is an approach to benefit advantages of both CPUs and GPUs [6]. HSA can integrate properties of CPUs, GPUs, and other various processing units. In HSA, CPUs and GPUs share the same Shared Virtual Memory (SVM). They could directly access data on other devices by using pointers and user applications could dispatch the kernels to GPU without data movement overhead. HSA utilizes the computing ability of GPUs to dramatically increase performance. Furthermore, it also provides low level runtime API [7] to manage tasks, resources, and Heterogeneous System Architecture Intermediary Language (HSAIL) that is a LLVM intermediary representation (IR) to represent parallel kernel from high level language like Open Computing Language (OpenCL), Open Multi-Processing (OpenMP) and even Java.

However, it is not easy to program on heterogeneous architecture. Many programming frameworks are proposed to solve this issue such as Halide [8], C++ AMP [9], OpenACC [10], and APARAPI [11]. OpenCL is one of the famous framework for heterogeneous and parallel computing [12], [13], [14]. It is an open standard maintained by Khronos Group [15], and designed to run on different kinds of parallel hardware resources like CPU, GPU, DSP, and FPGA. OpenCL involves host program and kernel code. Programmers write the host program by using some runtime APIs to get the information of computing devices, allocate memory buffer, create parallel kernel, and dispatch tasks. Kernel code executed in parallel is written by OpenCL C Language with some C99 extension such as memory qualifier and vector type.

In this paper, an OpenCL framework is designed and implemented on HSA platform. OpenCL C compiler and runtime that support OpenCL 1.2 on a HSA platform are built. Our compiler supports some OpenCL feature extensions, such as memory qualifiers and vector operations. The compiler translates the OpenCL kernel into HSAIL and the HSAIL is encoded into BRIG binary code format by using the tool released from HSA Foundation [16]. Furthermore, Portable Computing Language (PoCL) is used to build our OpenCL runtime [17]. PoCL is an open source OpenCL framework that targets various parallel hardware resources. In our framework, a HSA GPU device is added in the PoCL device list, and a bridge is written between HSA runtime APIs and OpenCL Runtime APIs that handle some special HSA features for OpenCL running on HSA GPUs. For example, the runtime arranges kernel arguments to execute kernels and dispatches kernels by filling the packet of Architecture Queuing Language (AQL). Furthermore, dynamic and local memory usages from the OpenCL program are handled to work on HSA platform.

In our experiments, OpenCL programs from AMD SDK 2.8 benchmarks are executed on AMD HSA-enabled processor [13]. From experimental results, the execution time of our framework of the OpenCL program is better than that of HSA-HLC-Stable compiler released by HSA Foundation, but is worse than the OpenCL compiler. The execution time of
our framework of the OpenCL kernel is better than the other two compilers in some applications, but is worse than in some other applications. Our framework successfully passes some benchmarks and is enough for advanced research such as vector operations and memory layout optimizations on OpenCL kernels and HSA task scheduling between CPUs and GPUs.

The remainder of this paper is organized as follows. Section 2 presents details on HSA and OpenCL. Section 3 describes the design and implementation details, and some experimental results are shown in section 4. Finally, conclusions and future work are described in section 5.

2. Overview of HSA and OpenCL

In this section, some introductions of HSA and OpenCL are introduced. Our compiler and OpenCL runtime is developed by using technologies of HSA and OpenCL. Some related properties of them will be introduced.

2.1 HSA Overview

The design goal of HSA can be shown in Fig. 1. The two hardware technologies Heterogeneous Uniform Memory Access (hUMA) and Heterogeneous Queuing (hQ) are introduced in HSA. hUMA can make both CPUs and GPUs share a single memory space that GPUs can directly access CPU memory addresses include reading and writing data and CPUs also read and write at the same time. Traditionally, users should view the CPUs and GPUs memory as completely separate memory even through CPUs and GPUs are integrated into the same chip. hQ lets CPUs place tasks of GPUs directly into GPU task queue without OS help and GPUs can also place its tasks into the GPU or CPU queue to be dispatched. hQ also uses Architecture Queuing Language (AQL) to define a standard Queuing Format. All HSA agents should be described as AQLs so that the agent can directly dispatch tasks into different HSA component hardware queues without software translation.

To obtain the benefit from the HSA features, HSA also provides a middle layer with low level runtime APIs and HSAIL to make high level heterogeneous language to lower into this layer. Therefore, high level heterogeneous language should have its compiler to translate the program into HSAIL and its runtime should be able to interact with HSA runtime. At execution time, the HSA runtime uses these APIs to dispatch tasks and manage resources. HSA runtime also invokes HSA Finalizer to translate the HSAIL into target machine code.

2.2 OpenCL Overview

OpenCL provides a programming framework to accelerate applications with tasks and data parallelism. OpenCL has C99-extended programming language, named as OpenCL C, for writing the kernel to run in parallel. OpenCL C involves some qualifier and vector operations to extend and built-in functions. Figure 2 is the simple OpenCL Kernel code. The __kernel qualifier identifies this kernel code that runs on OpenCL computing devices and is identified as an entry point. The memory qualifier __global means to place variables on global memory. The built-in function get_global_id() returns the thread id and the array uses this id as an index to access the data in the sample program.

In addition to the kernel code, programmers also must write the host program with OpenCL runtime APIs to control the task how to run on computing devices. For example, the kernel code is executed by many parallel threads on GPU or working-items in OpenCL specifications. Programmers must decide how these threads are divided into groups in host program. The threads in different groups can run independently and the threads in the same group can be synchronized during execution. Programmers also should decide the kernel code execution order, synchronization between kernels, memory allocations, and kernel code compilations. All of the above actions should be interacted with HSA.
3. The Proposed Framework with Compiler and Runtime

In this section, the proposed framework with compiler and runtime will be explained. Figure 3 shows our OpenCL flow on HSA. First, OpenCL host program runs on our PoCL-based OpenCL runtime. Runtime will invoke our LLVM-Based HSAIL compiler to compile OpenCL kernel into HSAIL and also invoke HSAIL assembler to translate the HSAIL into BRIG binary file. Our OpenCL runtime is built on the HSA runtime and driver released by HSA Foundation. The detailed compiler and runtime flow is described in the following.

3.1 The Prototype Compiler for OpenCL to HSAIL

Currently, most of OpenCL compilers are implemented by LLVM[19]. Khoronos also define OpenCL portable IR, Standard Portable Intermediate Representation (SPIR), which is based on LLVM IR with some specific annotations for OpenCL C extension. Therefore, our compiler uses Clang as front end and LLVM llc as backend. Figure 4 is the proposed OpenCL to HSAIL compilation flow. The OpenCL kernel code is converted to SPIR by Clang and then uses LLVM llc to compile bitcode to HSAIL. Finally, the hsailasm tool which is released by HSA Foundation is utilized to convert HSAIL code into BRIG format.

Our compiler has already supported some OpenCL features, such as kernel/memory qualifiers and some vector operations. Figure 5 is the simple HSAIL code converted from OpenCL codes shown in Fig. 2 by using our compiler. The __kernel qualifier for function vec_add is annotated in LLVM IR which is used to annotate kernel for function vec_add in HSAIL and kernarg for vec_add parameters. For the same reason, ld_global_xx and st_global_xx instructions are generated because the loaded and stored data is annotation with __global memory qualifier in OpenCL kernel. The built-in functions in OpenCL are mostly supported by intrinsic functions.

For example, the built-in function get_global_id(0) in Fig. 2 is directly translated to workitemabsid_u32 in Fig. 5. Vector accesses and some basic vector arithmetic instructions are also supported that makes the generated code more efficiently. Furthermore, if vector load instructions are supported, the vector data can be loaded directly instead of generating four scalar load instructions. As shown in Fig. 6, in the dark and light gray parts, four ld_global_u32 instructions can be merged into one ld_v4_global_u32. In this situation, the hardware does not load 32-bit data four times but directly load 128-bit data once.

3.2 PoCL-Based OpenCL Runtime for HSA

PoCL is used to build our OpenCL runtime for HSA. PoCL is one of the open sources implemented for OpenCL standard that has already supported different hardware such as homogeneous muti-core X86/ARM and VLIW-style TTA processors. PoCL has already passed some OpenCL benchmarks and been designed easily to adapt to new targets and
ld_kernarg_u64 $d0, [%b];
ld_global_u32 $s0, [$d0];
ld_global_u32 $s1, [$d0+4];
ld_global_u32 $s2, [$d0+8];
ld_global_u32 $s3, [$d0+12];
ld_kernarg_u64 $d1, [%a];
ld_global_u32 $s4, [$d1];
ld_global_u32 $s5, [$d1+4];
ld_global_u32 $s6, [$d1+8];
ld_global_u32 $s7, [$d1+12];
ld_kernarg_u64 $d0, [%b];
ld_kernarg_u64 $d1, [%a];
ld_v4_global_u32 ($s0, $s1, $s2, $s3), [$d0];
ld_v4_global_u32 ($s4, $s5, $s6, $s7), [$d1];

Fig. 6: A Simple HSAI of Vector Load

Fig. 7: The Runtime Flow between OpenCL and HSA devices. PoCL also uses the LLVM as a kernel compiler. Therefore, it is a good choice for building our OpenCL runtime for HSA by using PoCL.

Runtime flows of both OpenCL and HSA are similar. They need to request equipment, set work queues, load in memory addresses, compile kernel codes, forward parameters, and execute kernel codes. After executing the kernel codes, host programs release resources used. Figure 7 shows the relationship of runtime behaviors between the OpenCL and HSA.

When running an OpenCL host program, it should initialize an environment. The environment includes a platform that provides devices, which are supplied by hardware vendors or some open source projects, for kernels and OpenCL contexts to run. In a HSA host program, users also need to initialize HSA runtime first and get a HSA device by using HSA agent iteration functions. According to the user assignment in the host program, runtime can determine which devices are used. In our implementation, if the device type is CPU, a PoCL runtime API initializes device pthread that is the original device of PoCl and the kernel will be translated to pthread model in the build steps. If the device type is GPU or default, a Pocl runtime API initializes the HSA device. If the device type is ALL, both GPU and CPU devices are initialized. Besides, HSA runtime APIs also are invoked to obtain the device feature and setup some OpenCL device information.

Then, corresponding to OpenCL standards, users need to create a queue and program object for communication between hosts and devices. If the command queue is for HSA GPU, it also needs to invoke HSA runtime APIs to create HSA queue.

The OpenCL kernel code object is created by using opencl creating APIs, and different OpenCL APIs are used to store the OpenCL kernel source file or BRIG files. If the input file is OpenCL kernel code, kernel files are compiled to BRIG files as introduced in the compiler flow. After compiling, the BRIG file is loaded as a BRIG module and the module becomes the member of both HSA program object and OpenCL program object. Finally, the BRIG module is finalized at OpenCL kernel creating APIs. In this step, the kernel name can be obtained from the parameters with API and is used to find the correspondent symbol offset which must be filled in a finalization list. The finalizer finalizes the kernel object in the finalization list one by one.

At OpenCL kernel argument setting and kernel execution stage, the kernel argument for HSA should be initialized and the HSA agent packet should be filled in AQL prepared for dispatching kernel to execute. Table 1 illustrates the HSA dispatch packet members and the source. Header identifies the packet type. There are three types of the packet type, DISPATCH, AGENT_DISPATCH, and BARRIER. DISPATCH is used to dispatch the kernel from a host to a device,
and \texttt{AGENT\_DISPATCH} is used to dispatch kernel from a device to a device. This feature can be used to support the OpenCL 2.0 device feature. \texttt{BARRIER} is used to delay packets for describing packet dependencies.

Dimensions, grid\_size and workgroup\_size correspond to work\_dim, global\_work\_size and local\_work\_size which can be obtained from OpenCL APIs used by defining the number of thread creation. Private and Group segment size can be queried from HSA code descriptor acquired from finalizing the BRIG module. The group segment size only indicates static group size which is the total variable declaration size with \texttt{__local} qualifier in the OpenCL kernel code. The dynamic group size will be acquired from the OpenCL kernel argument setting API. Kernel\_object\_address is used to point to the address where the function kernel resides, and it can be queried from the HSA code descriptor. In the HSA runtime specification, a user should find a block of memory called as region that can be used in kernel arguments, and allocate the region as the place to store kernel arguments by using the HSA memory allocation API. The API would output the start address of the block of memory, and the address should be passed to Kernelfarg\_address. Complete\_signal is used to identify whether the kernel dispatched finishes executing or not. A HSA signal object is created by using the HSA signal API and is passed to complete\_signal.

In our kernel argument initialization mechanism, the argument buffer is allocated and initialized based on different types. The type of kernel arguments is accessed by using the kernel argument descriptor which is built when a user invokes the kernel creating API, and an argument structure array is generated that includes size and value obtained from the kernel argument setting API. Then, a buffer size is created which is queried from kernelfarg\_segment\_size, and the required kernel argument data is copied to the buffer one by one. A pointer is used to point to the start of the current argument data that should be copied. If the type of an argument that gets from the kernel argument descriptor is a local type, the group\_segment\_size should be copied as a pointer to the buffer instead of a null pointer created by users.

Dynamic group size also is added into group\_segment\_size, because group memory is divided into two parts according to HSA runtime specifications, static group memory and dynamic group memory. Static group memory is allocated from address 0x0 to the size group\_segment\_size, and dynamic group memory starts from group\_segment\_size. The amount of group memory size is obtained when a local qualifier argument is accessed. In terms of structure types, OpenCL and HSA access structure arguments in different ways. In OpenCL, the structure argument is regarded as a block of continuous memory. However, in HSA, the structure argument is regarded as a pointer. Therefore, the pointer of structure argument is copied instead of the total structure argument such as scalar type. After finishing initializing the kernel argument, all of the buffer data is copied to the address kernarg\_address in the AQL packet for dispatching.

4. Evaluation

In this section, some experimental results are performed to show the performance of our proposed compiler, HSAIL-HLC-Stable compiler, and OpenCL runtime on HSA platform.

4.1 Environment

AMD Kaveri A10-7850K APU was used as our experimental platform which includes one 4-core CPU and GPU and is the first HSA-enabled processor. The platform ran Ubuntu 14.04.2 LTS and AMD HSA runtime and driver 1.0 version released by HSA Foundation were installed. Our OpenCL to HSAIL compiler is built on LLVM 3.3 and OpenCL runtime is built on PoCL 0.8. Moreover, our benchmarks are selected from AMD APP SDK 2.8.

4.2 Experimental Results

In our experiments, three different OpenCL frameworks were compared. One is current AMD official OpenCL framework from AMD GPU driver Catalyst Omega 14.12. Another one uses our PoCL-based OpenCL runtime but uses HSAIL-HLC-Stable compiler released by HSA Foundation. The other framework uses our OpenCL runtime and compiler. Figure 8 illustrates the comparison of execution time between three frameworks. As shown in Fig.8 the execution time of PoCl-based Runtime with the proposed compiler is faster than that of AMD official OpenCL framework in some benchmarks, such as BitonicSort and FloydWarshall.

However, most of the execution time of the proposed compiler is longer than AMD official OpenCL framework. It means that our framework still can be improved in the execution time. Although the execution time of our framework in most applications are more than OpenCL runtime, the execution time of our framework decreased more than 70% when comparing with HSAIL-HLC-Stable. This is because the compilation time of HSAIL-HLC-Stable is too long for performing compiler optimizations. Therefore, the execution time of the kernel compiled by HSAIL-HLC-Stable is faster in most of applications as shown in Fig.9 and Table 2.

Figure 9 shows the comparison of kernel execution time between three frameworks and Table 2 lists the kernel execution time. The execution time of the kernel generated by our proposed compiler is better than AMD official OpenCL framework but worse than HSAIL-HLC-Stable. From the above two experiments, our proposed compiler has ability to do advanced research about HSA compiler and runtime optimizations.
5. Conclusions and Future Work

In this paper, an OpenCL framework is built on the HSA Platform. Our OpenCL to HSAIL compiler is built based on LLVM and runtime is based on PoCL. Our proposed compiler supports some OpenCL feature extensions, such as memory qualifier and vector operation. Our proposed compiler translates the OpenCL kernel code into HSAIL and the HSAIL is encoded into BRIG binary format by using the tool released from HSA Foundation. Therefore, a HSA GPU device is added in the PoCL device list, and some special HSA features are handled for OpenCL running on HSA GPUs, such as filling agent packet with AQL for kernel dispatching.

From experimental results, our proposed framework can pass some benchmarks from AMD APP SDK. Our proposed framework supports enough OpenCL features for advanced research, and is planned to support for more OpenCL features for performing some optimizations on compiler and runtime, such as performing vector and memory layout compiler optimizations on OpenCL kernels, and HSA task scheduling between CPUs and GPUs.

Acknowledgment

This research is supported in part by the Ministry of Science and Technology of Taiwan, the Ministry of Economic Affairs of Taiwan, and MediaTek Inc.

References


