2017 46th International Conference on Parallel Processing Workshops

OpenCL 2.0 Compiler Adaptation on LLVM for PTX Simulators

Chun-Chieh Yang[1], Shao-Chung Wang[2], Min-Yi Hsu[3], Yuan-Ming Chang[4], Yuan-Shin Hwang[5], and Jenq-Kuen Lee[6]
[12346]Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan
[5]National Taiwan University of Science and Technology, Taipei, Taiwan
{jet[1], scwang[2], myhsu[3], ymchang[4]}@pllab.cs.nthu.edu.tw, shin@csie.ntust.edu.tw[5], jklee@cs.nthu.edu.tw[6]

***Abstract*—OpenCL continues to gather momentum on both desktop and mobile devices. The new features of OpenCL 2.0 provides developers better expressive power in programming heterogeneous computing environments. Currently in the experimental simulation environment, gem5-gpu only supports CUDA, but GPGPU-Sim can support OpenCL by compiling OpenCL kernel code to PTX using real GPU driver. However, this driver compilation in GPGPU-Sim only can support up to OpenCL 1.2. To support OpenCL 2.0, it is necessary to extend the compiler to enable the compilation of OpenCL 2.0 kernel code to PTX. In this paper, our experience in enabling the compiler flow is reported. In OpenCL 2.0, it provides new features such as dynamic parallelism, work-group built-in functions, extend atomic built-in functions, and so on. The proposed compiler that is modified from Low Level Virtual Machine (LLVM) extends such features for enhancing the emulator to support OpenCL 2.0. After the compiler is modified, it can support dynamic parallelism, work-group built-in functions and extend atomic built-in functions. Using existing dynamic parallelism APIs in CUDA to implement OpenCL 2.0 enqueue kernel and revise compilation scheme in *clang*. Furthermore, the proposed compiler also creates local buffers for each work group to use for work-group built-in functions, and adds atomic built-in functions with memory order and memory scope for OpenCL 2.0 in NVPTX. From benchmarks, the proposed compiler can support the claim target.**

***Index Terms*—LLVM, OpenCL, PTX, Libclc, GPGPU-Sim**

I. Introduction

In recent years, a heterogeneous system consisting of multiple CPUs and GPUs is becoming increasingly attractive as a platform for high performance computing. Therefore, how to combine CPUs, GPUs, DSPs, and other devices to do parallel processing becomes an important issue. To address this issue, OpenCL is proposed [6], [7]. OpenCL continues to gather momentum on both desktop and mobile devices. Not only OpenCL is an open standard promoted by the Khronos Group, but also it can be used to program CPUs, GPUs, DSPs and other devices from different vendors.

Compared to OpenCL 1.2, OpenCL 2.0 offers the following new features: (1) Shared virtual memory: through OpenCL 2.0 APIs, OpenCL 2.0 can allocate a memory buffer to eliminate the memory copy between the CPU and GPU by sharing data in this memory. (2) Dynamic parallelism: OpenCL 2.0 allows kernel can launch kernels by itself to increase the flexibility of GPU scheduling. (3) Platform atomic: OpenCL 2.0 enhances atomic functions between the CPU and GPU to make the fine-grained data sharing possible. (4) Workgroup built-in functions: In order to improve performance and programmability, OpenCL 2.0 can effectively perform the restore operation in the GPU kernel.

Research focus is now on adding efficient APIs and hardware instructions to efficiently support OpenCL 2.0 applications. How to sufficiently utilize the new features of OpenCL 2.0 is a challenge and an opportunity for application developers. For research reason, there are several tools are currently available for simulating heterogeneous CPU-GPU system [3], [5], [16]. The proposed compiler is modified for the heterogeneous system which is based on the integrated CPU-GPU simulator, gem5-gpu [12]. This emulator consists of two parts: a CPU cluster with any number of CPUs modeled by gem5 [3], and a GPU modeled by GPGPU-Sim [5]. Currently, gem5-gpu only supports CUDA [4], but GPGPU-Sim can support OpenCL by compiling OpenCL kernel code to PTX using real GPU driver. However, this driver compilation in GPGPU-Sim only can support up to OpenCL 1.2. To support OpenCL 2.0, it is necessary to extend the compiler to enable the compilation of OpenCL 2.0 kernel code to PTX.

In this research work, this missing flow is addressed. The authors focus on the needed support in Low Level Virtual Machine (LLVM) compiler for a such OpenCL 2.0 framework [9], [15]. The proposed OpenCL 2.0 compiler is implemented from LLVM extends such features for enhancing the emulator to support OpenCL 2.0 that uses *Clang* as the front end to compile OpenCL code to LLVM bitcode and LLVM *llc* as the back-end to support the simulator [13]. To support OpenCL 2.0, *Clang* and *llc* for new language features are modified and *libclc* [10] for new built-in functions is extended, as marked in gray in Fig. 1.

First, *Clang* is modified to support program scope global variables that allow the programmers to declare and initialize the variables in the global address space at program scope. Second, compiler intrinsic functions are used to support new atomic built-in functions. The proposed compiler can automatically map the atomic built-in functions into our extended atomic PTX ISA. Third, *libclc* is revised to support work-group built-in functions. Most of work-group built-in functions are implemented by the parallel tree reduction algorithm.

Finally, the generic address space is supported by using the CUDA implementation in *Clang*. Pointers are declared to the generic address space when the default address space is not specified. If the compiler can infer the address space after analysis, compiler will annotate the address space for these pointers. Otherwise, in code generation pass, the load and store instructions are printed without address space syntax.

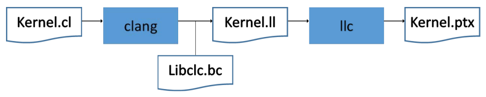


Fig. 1. OpenCL 2.0 Compilation Flow

The remainder of this paper is organized as follows. Section II presents some background. Section III describes the system model. In Section IV shows some simulation results. Finally, conclusions and future work are presented in Section V.

II. BACKGROUNDS

This section will describe some backgrounds such as OpenCL 2.0 and LLVM. OpenCL 2.0 provides some new features with a heterogeneous platform for CPUs, GPUs and DSPs. LLVM provides a compiler infrastructure with complete compiler flow.

A. OpenCL 2.0

OpenCL provides a programming framework to accelerate applications with parallelism by Khronos groups. The standard can be divided into two parts. The one part is the device-side kernel code that uses programming language specification based on C99 standard. OpenCL leverages C99-extended programming language, named as OpenCL C, for writing the kernel to run in parallel. OpenCL C also provides vector operations and built-in functions. There are three levels of parallelisms in OpenCL including work group parallelism, thread parallelism, and vector parallelism. Another part is the host-side code that is used to interact with the device. In addition to writing the kernel code, the programmers also must write the host code with OpenCL runtime APIs to control the task how to run on computing devices.

The OpenCL 2.0 API specification was devised around July 2015 by Khronos groups. It offers the following new features such as shared virtual memory, pipe linguistics, dynamic parallelism, workgroup built-in functions, and atomic operations. The OpenCL 2.0 specification has made significant improvements over the version 1.2, resulting in enhanced communication and collaboration among the hardware in a heterogeneous system. In OpenCL 1.2, the same virtual address space is not shared between the host and the OpenCL device, so the host memory and device memory must be managed, and then the communication between the two memories is needed to handle. For example, OpenCL device cannot use the pointer in the memory of the host side. OpenCL 2.0 breaks through this limit and it lets hosts and OpenCL devices can share the same range of virtual addresses. In OpenCL 1.2, GPUs need to spend time on copying and dealing with the buffer, but GPUs only need to deal with kernels program in OpenCL 2.0.

Besides, OpenCL 2.0 also introduces a new working mechanism for passing data between different kernel programs, with the name "pipes". A pipe is essentially a structured buffer, which contains "packets" of the collection space. The packet is the kernel program type object. These packets are placed in an orderly row. Pipes can only be accessed by kernel program functions and cannot be accessed by the host.

B. Low Level Virtual Machine

LLVM is a compiler framework for providing different compilation stage optimizations of arbitrary programming languages that include front end to compile user's code to LLVM bitcode and back-end to compile the bitcode to the machine code [8], [9]. LLVM also provides a powerful intermediate representation for efficient compiler transformations and analysis. LLVM intermediate representation (IR) can be represented in three different forms: as an in-memory compiler IR, as an on-disk bitcode representation, and as a human readable assembly language representation. There are many target architectures through different back-ends supported by LLVM such as X86, PowerPC, ARM, Thumb, SPARC, Alpha, PTX and CellSPU. LLVM IR provides intrinsic functions to support target-architecture specific functionality. Unknown intrinsics without specific handling are handled like external function calls during all LLVM passes.

Using LLVM to implement the OpenCL compilers is a trend currently. The Khronos Group also defines the standard portable intermediate representation (SPIR), which is based on the LLVM IR with specific annotations for OpenCL C extension. Therefore, the proposed compiler uses *Clang* as front end and *llc* as back-end.

C. The Gem5-GPU for OpenCL 2.0 Simulator

The gem5-gpu simulator for OpenCL 2.0 is proposed from NTU and it is the first CPU-GPU simulator that supports OpenCL 2.0 [16]. CPU and GPU in gem5-gpu share the same address space and the GPU architecture is capable of running OpenCL 1.2 kernels. Compared to OpenCL 1.2, OpenCL 2.0 provides the new features are widely adopted by industry and it better represents the future design of heterogeneous systems. So, this simulator extends to support OpenCL 2.0. The integrated CPU-GPU simulator consists of two parts: a CPU cluster with any number of CPUs modeled by gem5 [3], and a GPU modeled by GPGPU-Sim [5]. In this system, CPU cores and a GPU are integrated on the same chip. Using a single off-chip memory with a shared directory that manages a directory-based cache coherence protocol connected both CPU cluster and GPU.

III. SYSTEM MODEL

OpenCL 2.0 features supported in our LLVM compiler on GPGPU-Sim, such as *enqueue_kernel*, atomic built-in functions, work group functions, program scope variable and generic address space. In this section, how to build up the OpenCL compiler is explained.

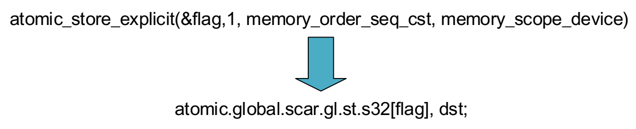
A. Platform Atomic

To achieve inter-thread communication, OpenCL 2.0 uses atomic functions to provide an efficient way. Although NVPTX back-end in LLVM already supports atomic functions, it adds *memory_order* and *memory_scope* to extern atomic functions in OpenCL 2.0. The *memory_order* specifies the detailed regular memory synchronization operations. The *memory_order* is the enumerated type and it contains the following enumerated options: *memory_order_relaxed*, *memory_order_acquire*, *memory_order_release*, *memory_order_acq_rel* and *memory_order_seq_cst*. The *memory_scope* specifies whether the memory ordering constraints given by *memory_order* apply to work-items in a work-group or work-items of a kernel(s) executing on the device or across devices. The *memory_scope* is also the enumerated type and it contains the following enumerated options: *memory_scope_work_item*, *memory_scope_work_group*, *memory_scope_device* and *memory_scope_all_svm_devices*. These options are specified three different degrees of the consistency in OpenCL: relaxed, acquire-release and sequential.

There are two parts are modified. One is adding new atomic instructions in NVPTX back-end, and the other one is creating the atomic built-in functions in the *libclc* based on these atomic instructions. These atomic instructions are virtual intrinsics code and only supported by the simulator. Table I shows the virtual intrinsics code in the proposed compiler and Fig. 2 shows the sample code after compilation. For supporting the atomic functions, some arguments such as *memory_scope* and *memory_order* are also added in the *libclc*.

TABLE I
THE BUILT-IN FUNCTION AND ITS VIRTUAL INTRINSICS CODE

OpenCL 2.0 built-in function	**PTX Instruction Extension in Simulation**
atomic_store	atom.space.order.scope.op.type space : global, share order : acq, rel, ar, rlx • *memory_order_relaxed* : *rlx* • *memory_order_acquire* : *scacq* • *memory_order_release* : *screl* • *memory_order_acq_rel* : *scar* *memory_order_seq_cst* : *scar* Scope: • *memory_scope_work_group* : *cta* • *memory_scope_device* : *gl* • *memory_scope_all_svm_devices* : *sys*
atomic_store_explicit	
atomic_load_explicit	
atomic_fetch_add _explicit	
atomic_compare_ exchange_strong	
atomic_work_item _fence	



```
atomic_store_explicit(&flag,1, memory_order_seq_cst, memory_scope_device)
atomic.global.scar.gl.st.s32[flag], dst;
```

Fig. 2. The Sample Code with Atomic Store Explicit

B. Workgroup Built-in Function and Program Scope Variable

OpenCL 2.0 implements the new workgroup built-in functions that operate on a work-group level. There are three types of work-group built-in functions:

- Scan: Return the result of sum, min, or max for all threads with thread ID smaller than current thread, optionally including current thread.
- Reduce: Return the result of sum, min, or max among all threads in a work-group.
- Broadcast: Return the data of target thread by specifying the target thread ID.

To support work-group built-in functions in OpenCL 2.0, the source code of the *libclc* is modified. By creating a scratch memory in the local memory, when users setup the pragma, the pragma is enabled to support the new work-group built-in functions. Using this scratch memory, let the kernels that use this feature can exchange data, such as *work_group_broadcast*. The *work_group_barrier* based on the system barrier call is created to guarantee the exchange data is correct. Listing 1 shows the sample code of the *work_group_broadcast*.

Listing 1. The Sample Code of the work_group_broadcast

```
__global ulong__wg_scratch[
        MAX_WORK_GROUP_NUM_PER_GPU][
        MAX_WAVES_PER_SIMD];
#define GEN_BROADCAST(TYPE)
__attribute__((overloadable, weak,
        always_inline))TYPE
Work_group_broadcast(TYPE a, size_t local_id_x)
{
 __global TYPE *p = (__local TYPE *)__wg_scratch;
 if (get_local_id() == local_id_x)
 ......

}
```

C. Dynamic Parallelism

The Dynamic parallelism feature is that GPU can self-enqueue kernels to reduce CPU intervention. The proposed compiler also supports dynamic parallelism that allows the running kernel to invoke other child kernels in the same device without communicating to the host. Listing 2 shows the OpenCL sample code for supporting dynamic parallelism.

First, a built-in function gets the default queue which is used to get the default device queue (Q) at line 3. At line 4, a builtin function *ndrange_1D* is used to set the number of invoked child kernels (NDR). Each invoked child kernel (myblock) is represented by block syntax, which is similar to a function type, as shown at line 5 of Listing 2. Finally, the built-in function *enqueue kernel* enqueues the block to the device queue.

In NVIDIA GPUs, the dynamic parallelism at the PTX level is supported by three device-side launch APIs, including *cudaStreamCreateWithFlags*, *cudaGetParameterBuffer*, and *cudaLaunchDevice* [2]. These three device-launch APIs can be leveraged to implement the sample code in Listing 3 at PTX level. *cudaStreamCreateWithFlags* returns the device queue and can be used to implement OpenCL built-in function to get default queue.

Listing 2. Dynamic Parallelism Sample Code

```
……
int tid = get_global_id (0);
Queue_t Q = get_default_queue ();
ndrange_t NDR = ndrange_1D (32);
void (^myblock) ( void)=^{
        VectorAdd_child (a, b, c, tid *32);
 };
 enqueue_kernel (Q, …, NDR, myblock);
……
```

Additionally, *cudaGetParameterBuffer* and *cudaLaunchDevice* are used to implement the built-in function enqueue kernel. *cudaGetParameterBuffer* returns the parameter buffer for filling the parameters and *cudaLaunchDevice* launches the block with the parameter buffer. Because the invoked blocked pointer and its parameters are stored at global memory, they can be directly accessed and copied into the parameter buffer.

The PTX-level dynamic parallelism code generated by the proposed compiler consists of three built-in functions, which act like runtime APIs to abstract away the detailed implementation. The three built-in functions are:

- *cudaStreamCreateWithFlags*: This function creates a device command queue to push child kernels in. Threads in the same work-group will get the same command queue when this function is called.
- *cudaGetParameterBuffer*: This function takes the function pointer and metadata of the child kernel (i.e. the number of work-groups in this kernel and the size of a workgroup) as inputs and allocates a parameter buffer for the parent thread to write parameters in.
- *cudaLaunchDevice*: This function takes a parameter buffer and a stream as inputs, and it push the kernel associated with the input parameter buffer into the input stream.

To support OpenCL C block language extension, the block implementation is leveraged in *Clang*. Existing dynamic parallelism APIs in CUDA are used to implement the sample code in Listing 3. The authors let the *Clang* treat child kernel block as *ObjC* block in Listing 4a and it will capture two variables. The sample code in Listing 4b and 4c are the original *ObjC* block representation in *Clang*. In Listing 4b, the *Clang* will create the *block_literal* structure and this structure will include the captured variables. But in original *invoke_function*, it will implicit the argument name and the captured variables will be extracted from code during codegen. Therefore, the compilation scheme in Listing 5 is revised for *ObjC* block statement in *Clang* to handle captured variables.

Some block information also are stored, such as the kernel function pointer and its parameters, in global memory to support OpenCL device enqueue APIs.

Listing 3. Pseudo Code of Current Approach

```
void enqueue_kernel(… ,  void (^childBlk)(void)){
        void* buffer = cudaGetParameterBuffer(
                childBlk->invoke_func,⋯
        };
        //Try to get captured variables from childBlk

        foreach captures as capture
                memcpy(buffer+offset, capture);

        cudaLaunchDevice(buffer,⋯);
}
```

Listing 4a. The Sample Code of the Dynamic Parallelism

```
__kernel void foo(__global float* argA){
        int x = 100;
        enqueue _kernel(queue, flags, ndrange,
        ^{
                *argA = (float)x + 3.0f;
        });
}
```

Child Kernel Block (ObjC block)

Captured Variables

Listing 4b. The Structure of the *block_literal*

```
typedef struct {

        void* isa;  //"Is a (something)"
        int reserved;
        void* invoke_function;
        block_descriptor descriptor;

        int capturedA;
        float* capturedB;

}block_literal;
```

The "real worker"

Captured variables

Listing 4c. Original *invoke_function*

```
//Implicit Argument (inserted during codegen)
define void @invoke_func(i8* block, i32 arg 1, ⋯){

        %capturedA = getelementptr block, offsetA
        %capturedB = getelementptr block, offsetB
        ……
}
```

Captured variables extraction code (inserted during codegen)

IV. Experiments

This section presents some experiments aimed at validating and demonstrating the features available with our OpenCL 2.0 compiler.

A. *Experimental Platform*

The experiments were performed on Ubuntu 14.04.4 64-bit with LLVM version 3.8. The CPU in the platform is Intel Core

Listing 5. Revise *invoke_function* in *Clang*

```
define void @invoke_func(i8* block, i32 arg1, ...){

        %capturedA = getelementptr block, offsetA
        %capturedB = getelementptr block, offsetB
        ......
}

define void @invoke_func(
        i32 capturedA,
        i32 capturedB,
        i32 arg1,...){
        ......
}
```

CPU i7-6700 @ 3.4GHz and the GPU is NVIDIA GeForce GTX 980. The GPGPU-Sim version is 3.2.2. Two benchmarks were used: one is the AMDAPPSDK 3.0 and the other one is the NTU OpenCL benchmark [1], [11].

B. Experimental Results

Table II shows the experimental results with the proposed compiler on AMDAPPSDK 3.0 benchmark. There are 9 benchmarks can verify and pass on GPGPU-Sim, and there is one benchmark passed on the proposed compiler but it still is verified for dynamic parallelism on simulator. The scale-invariant feature transform (SIFT) also is passed in the NTU OpenCL benchmark [14]. The SIFT is a extraction algorithm with object recognition. Figure 3 shows the execution time of some benchmarks that already can execute on the simulator gem5-gpu.

TABLE II
AMDAPPSDK BENCHMARK EXPERIMENT FOR THE PROPOSED OPENCL 2.0 COMPILER VALIDATION

AMDAPPSDK 3.0 Sample	**OpenCL 2.0 features**	**Verified on GPGPU-Sim**
SimpleGenericAddressSpace	Generic Address Space	Pass
RangeMinimumQuery	Shared Virtual Memory pointer with offset	Pass
BuiltInScan	New Workgroup Built-in APIs	Pass
SVMAtomicsBinaryTreeInsert	SVM Fine Grain Buffer + Platform Atomics	Pass
CalcPie	Platform atomic	Pass
FineGrainSVMCAS	SVM Fine Grain Buffer + C++ 11 Atomics	Pass
FineGrainSVM	SVM Fine Grain Buffer + C++ 11 Atomics	Pass
SVMBinaryTreeSearch	SVM Coarse Grain	Pass
RecursiveGaussian _ProgramScope	Program scope variable	Pass

Table III shows the petajoule (PJ) that each stage SIFT consumes for using memory on OpenCL 1.2 and OpenCL 2.0. As shown in Table III, the consuming energy for accessing memory can be divided into following parts: active memory block, read, write and refresh memory block, and close memory block. From the table, it can be found that the PJ is decreased during each stage on OpenCL 2.0 when SIFT uses memory.

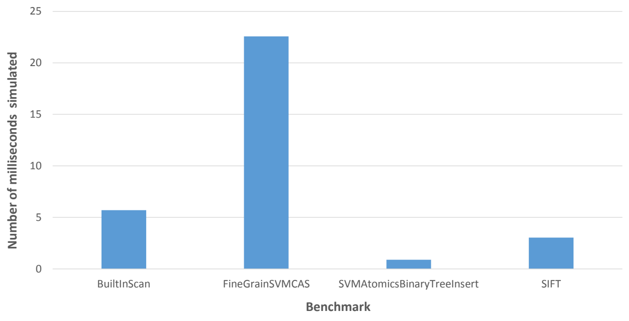


Fig. 3. The Execution Time of Some Benchmarks

TABLE III
THE DETAIL ENERGY USED ON MEMORY OF SIFT (PJ)

	SIFT 1.2	**SIFT 2.0**
Energy for active and precharge	15634530	15225555
Energy for read, write and refresh	480997920	478998000
Energy for activate and precharge background	4026971625	4009942125
Total energy	4523604075	4504165680

V. CONCLUSIONS AND FUTURE WORK

In this paper, a compiler is proposed based on LLVM, and the compiler can support OpenCL 2.0 features for PTX. This compiler implements several features such as program scope variable, work-group built-in function, platform atomic, and dynamic parallelism. Virtual intrinsics code is added to support atomic extenuation in the proposed compiler and the code only supports this simulator. By creating a scratch memory in the local memory, when users setup the pragma, the pragma is enabled to support the new work-group built-in functions. Existing dynamic parallelism APIs in CUDA are used to implement the dynamic parallelism feature.

Although most features of OpenCL 2.0 is verified, the proposed compiler is still continuing modified to support new features in OpenCL 2.0. Currently, the pipe feature of OpenCL 2.0 is still under verified. In the future, the proposed compiler is hoped can fully satisfy all features of OpenCL 2.0.

ACKNOWLEDGMENT

This research is supported in part by the Ministry of Science and Technology of Taiwan under Grant: MOST 105-2622-8-002-002, and MediaTek Inc.

References

[1] AMD OpenCL Accelerated Parallel Processing (APP). http://developer.amd.com/tools-and-sdks/.

[2] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174. IEEE, 2009.

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[4] CUDA Zone. https://developer.nvidia.com/cuda-zone.

[5] GPGPU-Sim. http://www.gpgpu-sim.org/.

[6] Khronos. https://www.khronos.org/.

[7] Khronos OpenCL Resources. https://www.khronos.org/opencl/resources.

[8] Chris Lattner and Vikram Adve. The llvm instruction set and compilation strategy. *CS Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS*, 2002.

[9] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[10] libclc. http://libclc.llvm.org/.

[11] opencl2.0-sim. https://github.com/ntueclab/opencl2.0-sim.

[12] Jason Power, Joel Hestness, Marc S Orr, Mark D Hill, and David A Wood. gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters*, 14(1):34–36, 2015.

[13] Dillon Sharlet, Aaron Kunze, Stephen Junkins, and Deepti Joshi. Shevlin park: Implementing c++ amp with clang/llvm and opencl. In *General Meeting of LLVM Developers and Users*, 2012.

[14] Seven OpenCL Benchmarks for Heterogeneous System Architecture Evaluation. http://mtkntu.ntu.edu.tw/upload/edmfs150404031052772.pdf.

[15] The LLVM Compiler Infrastructure. http://llvm.org/.

[16] Li Wang, Ren-Wei Tsai, Shao-Chung Wang, Kun-Chih Chen, Po-Han Wang, Hsiang-Yun Cheng, Yi-Chung Lee, Sheng-Jie Shu, Chun-Chieh Yang, Min-Yih Hsu, Li-Chen Kan, Chao-Lin Lee, Tzu-Chieh Yu, Rih-Ding Peng, Chia-Lin Yang, Yuan-Shin Hwang, Jenq-Kuen Lee, Shiao-Li Tsao, and Ming Ouhyoun. Analyzing opencl 2.0 workloads using a heterogeneous cpu-gpu simulator. In *accepter by ISPASS 2017 poster*. IEEE, 2017.