Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

HIPLZ: Enabling Performance Portability for Exascale Systems

Jisheng Zhao*[1] | Colleen Bertoni[2] | Jeffrey Young[1] | Kevin Harms[2] | Vivek Sarkar[1] | Brice Videau[2]

[1]Computer School, Georgia Institute of Technology,, Atlanta, GA, United States
[2], Argonne National Laboratory, Lemont, IL, United States

Correspondence
Jisheng Zhao. Email: jisheng.zhao@cc.gatech.edu, Colleen Bertoni Brice Videau Email: bertoni@anl.gov, bvideau@anl.gov

Present Address
Atlanta, GA, United States

Summary

While heterogeneous computing has emerged as a dominant trend in current and future High-Performance Computing (HPC) systems, it is also widely recognized that this shift has led to increased software complexity due to a proliferation of programming systems for different heterogeneous processors. One such example is the Heterogeneous-Compute Interface for Portability from AMD (`HIP`), which is composed of a C Runtime API and C++ Kernel Language. Many HPC applications will likely use `HIP` on future exascale systems (e.g., Frontier and El Capitan), but `HIP` currently only targets AMD and NVIDIA processors. This limitation creates challenges for users who would also like to run their applications on exascale systems based on other architectures (e.g., Aurora, which is based on Intel hardware) that are currently not targeted by `HIP`.

In this paper, we introduce the design and implementation of `HIPLZ`, a compiler and runtime system that uses the Intel Level Zero API to support `HIP` on Intel GPU architectures. We discuss the design of `HIPLZ`, derived from `HIPCL` (an implementation of `HIP` on top of `OpenCL`), and portability issues that occur from using the Level Zero runtime as a backend. We evaluate our implementation by running several performance benchmarks and mini-apps written in `HIP` on Intel architectures using `HIPLZ`. Our results show that this approach provides competitive performance relative to Intel's OpenCL implementations on Intel Gen9 and UHD Graphics 770 GPUs, while providing good coverage of features needed by HPC applications. Overall, this approach is a promising demonstration of enabling performance portability for exascale systems.

KEYWORDS:
HPC, HIP, Parallel Programming Model, Runtime

1 | INTRODUCTION

Modern High Performance Computing (HPC) has been defined as an era of extreme heterogeneity where an increasing number of accelerators support SIMD parallelism, spatial computing, or domain specific architectures. This is especially true as we move toward exascale, where the majority of pre-exascale and exascale systems are accelerator-based. For example, 7 of the top 10 systems in the Top 500 for November 2021 are GPU-based [1]. Recently, NVIDIA systems were the dominant accelerator which applications would target, but several next-generation systems will be based on accelerators from different vendors: Aurora and SuperMUC-NG Phase II, with Intel GPUs [2,3] and Frontier, El Capitan, and LUMI with AMD GPUs [4,5,6,7,8]. Each vendor generally develops its own programming model and implementation which is optimized for its hardware. This design poses a challenge for application developers who wish to create portable code for multiple systems. Often this programming model heterogeneity results in application developers or library (e.g. Tensorflow [9], PyTorch [10])developers maintaining multiple branches of code in each different vendor-specific programming model, which increases code complexity and developer time requirements.

Heterogeneous-Compute Interface for Portability (HIP) from AMD is one example of such a programming system that targets AMD and NVIDIA architectures. In this paper, we introduce HIPLZ: a compilation and runtime system that supports HIP via Intel's Level Zero (L0) runtime [11] using the fat binary model for supporting multiple architectures and SPIR-V as an intermediate language (IL). To the best of our knowledge, HIPLZ is the first effort that bridges HIP to L0 which is the primary low level application programming interface (API) for Intel hardware. Since L0 is the fundamental software interface to Intel GPU and other hardware acceleration facilities, HIPLZ aims at evaluating a L0 based HIP implementation regarding performance and productivity.

In this paper, we present the following contributions:

1. The prototype of HIPLZ, a library that allows applications using the HIP API to run on devices that support Intel Level Zero and OpenCL drivers. The source code is located at: https://github.com/jz10/anl-gt-gpu .
2. A test suite that covers the major functionality of HIP and that uses it as the validation of HIPLZ.
3. An evaluation of test coverage and code performance of HIPLZ on two Intel GPUs: Gen9 and UHD Graphics 770. Our results show that HIPLZ supports the complete execution of 88% of tested applications and demonstrates performance parity with HIPCL and OpenCL for memory- and FLOP-focused benchmarks.

The paper is organized as follows: Section 2 gives background information about the HIP programming model, intermediate representation and the Intel L0 runtime. The details of the design and implementation are presented in Section 3. Section 4 discusses testing HIPLZ and evaluates the performance of HIPLZ. The related works are discussed in Section 5, and Section 6 concludes this paper.

2 | BACKGROUND

2.1 | Heterogeneous-compute Interface for Portability (HIP)

HIP [12] is a C++ 14 Runtime API and kernel language that is derived from CUDA [13] and that allows developers to create portable applications for AMD and NVIDIA GPUs from a single source code. It supports advanced C++ programming language features including templates, C++11 lambdas, and many other features. HIP is designed for portability with direct CUDA mode, so it supports automatically converting HIP API calls to CUDA. AMD also provides a tool called *hipify* [14] that can automatically convert CUDA codes to HIP with limited programmer effort.

2.2 | Standard Portable Intermediate Representation (SPIR-V) and Fat Binary

SPIR-V [15] is an industry open standard intermediate language (IL) for shader and kernel language compilers used for expressing parallel computation and GPU-based graphics. SPIR-V provides a common IL to developers for building computing kernels without needing to directly expose source code. This IL allows shipping compiled kernels in binary format while remaining portable on multiple hardware implementations.

The fat binary can be either IL or machine code binary. In the IL case, the fat binary model integrates base device code (kernel functions) into the host side executable binary via intermediate languages, and uses vendor APIs (driver compiler) to apply just-in-time compilation on kernel functions during runtime. SPIR-V and NVIDIA PTX are typical examples of IL used in fat binary.

2.3 | OpenCL and HIPCL

OpenCL [16] is a widely used, open standard for programming heterogeneous platforms, and is supported by most of the major accelerator vendors, including NVIDIA, AMD, Xilinx, ARM, and Intel. OpenCL is often used as a backend for higher level programming languages and APIs such as OpenMP, but is also used directly by some applications.

HIPCL [17] is an open-source compilation and runtime system that allows running HIP programs on OpenCL platforms with sufficient capabilities. HIPCL relies on SPIR-V as a target IL (i.e. fat binary embedded in ELF binary) and implements the HIP API on top of OpenCL calls.

2.4 | Level Zero Runtime

Intel Level Zero (L0) [11] is a specification which is part of the Intel oneAPI suite which is composed of SYCL-based specification [1] and set of APIs and tools targeting CPU, GPU and FPGA devices (see Figure 1). The Intel L0 implementation provides a low-level library for interacting with accelerator

[1] SYCL [18,19] provides heterogeneous programming related classes and lambdas in C++ language.

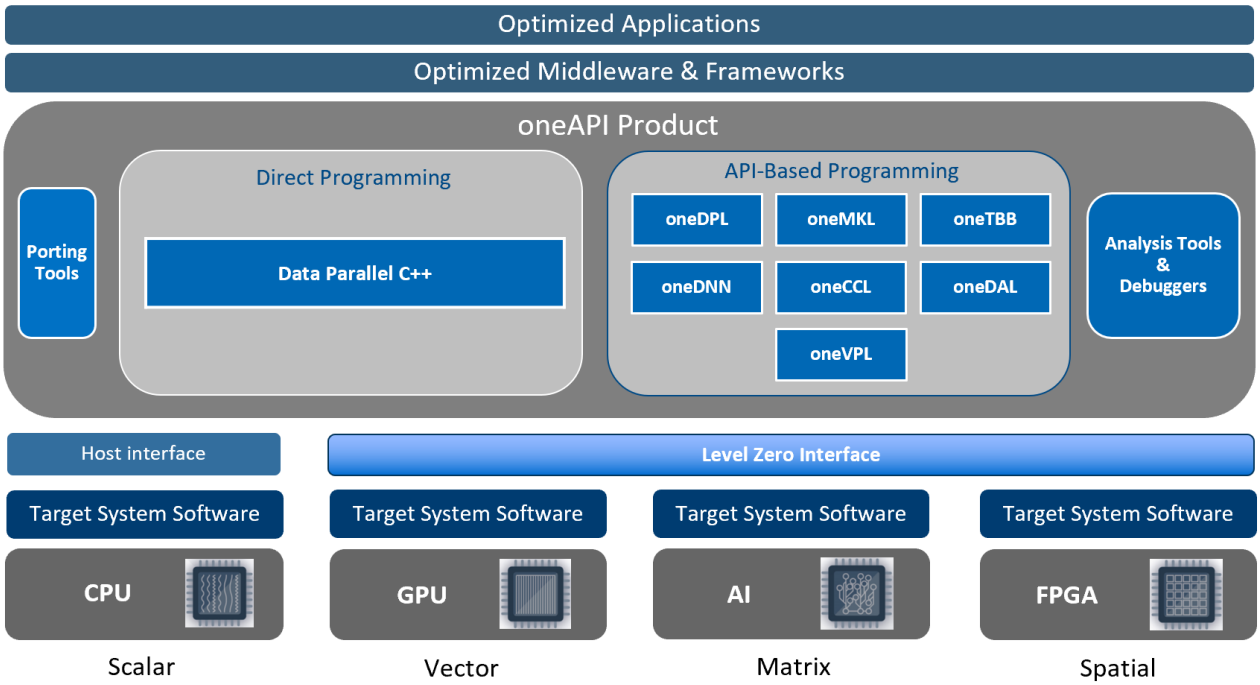

Figure 1 The Level Zero Runtime in Intel's oneAPI Framework (image source: https://spec.oneapi.io/level-zero/latest/core/INTRO.html , Graphic courtesy of Intel Corporation).

devices and brings flexibility through the support of a broad set of language features, e.g. unified shared memory, synchronization primitives, and device function pointers. The aim of the `L0` API is to provide a system level programming interface that easily allows higher level runtime APIs and libraries to target heterogeneous hardware. This is why we selected it for `HIPLZ`. The features of the `L0` API include, but are not limited to: device partitioning, instrumentation, debugging, power managements, frequency control, and hardware diagnostics. The `L0` specification does not define an intermediate representation for kernel language, but relies on SPIR-V as an IL.

3 | DESIGN AND IMPLEMENTATION

3.1 | Design Goal

The main design goal of `HIPLZ` is to connect the Intel `L0` runtime to the `HIP` programming model, thus enabling applications written using `HIP` to run on GPU devices driven by `L0`. Based on a survey of HPC application needs, we focused on supporting the following `HIP` features in `HIPLZ`:

- streams, including the command execution and callbacks (Section 3.4);
- memory management, including host, device, shared memory, and texture memory (Section 3.5);
- kernel and module management (Section 3.6);
- device management (Section 3.7);
- inter-operation with other parallel programming systems like Intel's DPC++ (Section 3.8);
- hipGraph related functionalities, including graph construction, launch, destroy and graph node creation (Section 3.10).

We ended up implementing 142 functions in `HIPLZ` out of 153 total `HIP` functions at the time he first version of `HIPLZ` was released. `HIP` now has 343 functions and the unincluded functions are mainly for graph operations.

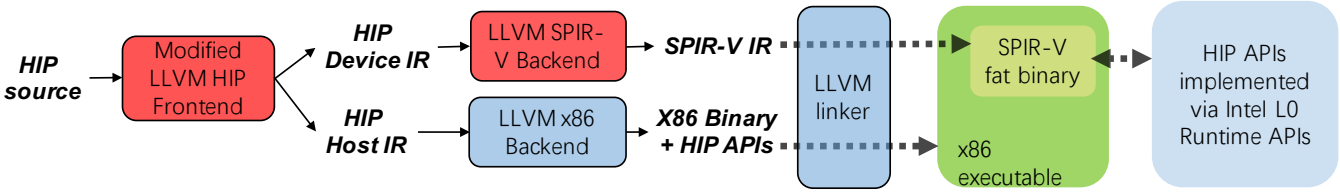

Figure 2 The compilation workflow for `HIPLZ`.

3.2 | The compilation system

The workflow for the compilation of a `HIP` program by `HIPLZ` is shown in Fig. 2. The `HIPLZ` compilation workflow is based on that of `HIPCL`, which is a `HIP`-compatible compiler frontend based on the LLVM/Clang compiler.

The `HIPLZ` compiler translates `HIP` source code to two parts of LLVM intermediate representation : host IR and device IR. The host part is processed via the legacy LLVM x86 backend to produce an x86 binary, and the device part is processed via the LLVM SPIR-V backend to produce SPIR-V IR. The x86 binary and the SPIR-V IR are then linked together to make an x86 executable binary (or shared library) that is embedded with SPIR-V (a fat binary).

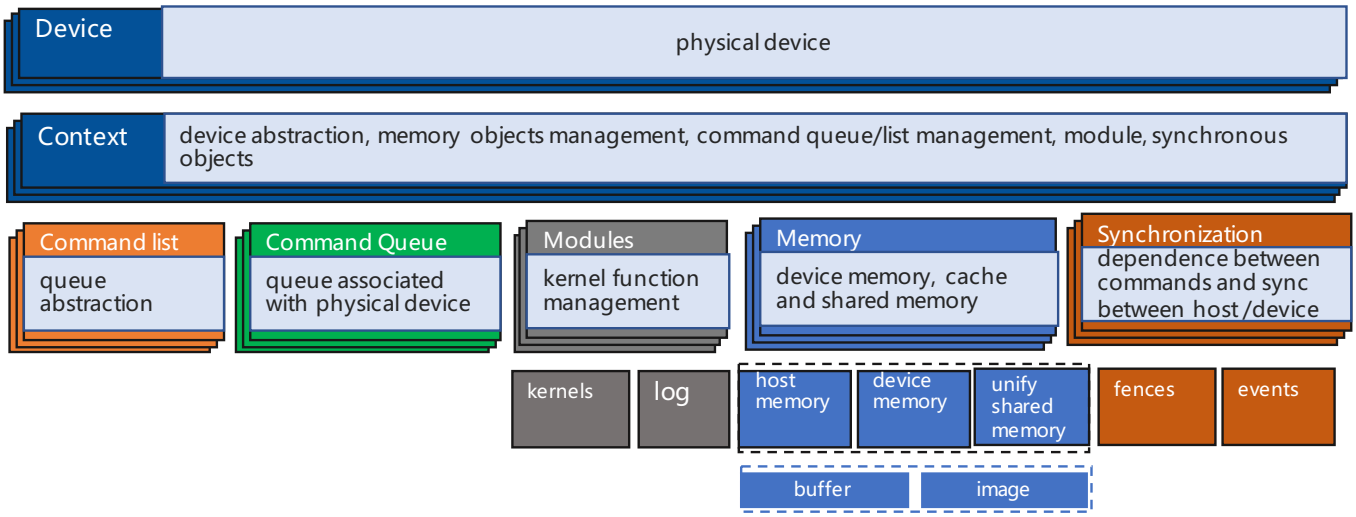

Figure 3 The organization of Intel Level Zero runtime.

3.3 | Runtime System

Before getting into the details of `HIP` feature support, here we introduce the basic structure of `L0`. Figure 3 presents the organization of `L0` APIs and objects in a top-down manner. On the top level, each `Driver` interacts with a collection of heterogeneous computing devices that share a given software stack. A physical device is presented as a `Device` that is associated with a `Context` that provides an interface for managing memory, modules, synchronization objects, command lists and queues. `L0`'s memory management covers hosts, devices, shared memory, and image samplers.

The `L0` API is very similar to `OpenCL`'s, especially in terms of the device data abstraction, execution model, and event driven synchronization. However, `L0` is at lower level and many features that are available in `OpenCL` are left to the application developer to implement. Such features include (but are not limited to) reference counting to handle object lifetime, callbacks on events' state change, or host kernel enqueuing. `HIPLZ` wraps `L0` data structures in C++ classes in an object-oriented manner, similar to `OpenCL`'s C++ bindings. The reference relation among the classes (the same as in `L0`) is shown in Figure 4.

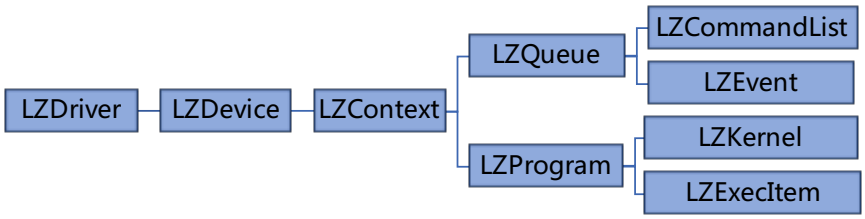

Figure 4 The HIPLZ class organization.

Table 1 gives some details about the functional correspondence of similar objects for the different programming models we will use in the next sections HIP, HIPLZ, L0, OpenCL, and SYCL. The HIPLZ compiler translates a HIP object to its corresponding data structure in HIPLZ as in the first two columns in Table 1.

Table 1 Functional correspondence among HIP, HIPLZ, L0, OpenCL, and SYCL objects.

HIP	HIPLZ	L0	OpenCL	SYCL
hipDevice	LZDevice	ze_device_t	cl_device_id	sycl::device
hipContext	LZContext	ze_context_t	cl_context	sycl::context
hipStream	LZQueue	ze_queue_t	cl_command_queue	sycl::queue
		ze_commandlist_t		
hipModule	LZModule	ze_module_t	cl_program	sycl::program
hipFunction	LZKernel	ze_kernel_t	cl_kernel	sycl::kernel
hipTextureObject	LZTexture	ze_image_t	cl_image	sycl::image
		ze_sampler_t	cl_sampler	sycl::sampler
hipGraph_t	LZGraph	ze_command_list	cl_command_buffer	

3.4 | Streams

A stream in HIP is presented as a sequence of tasks (e.g. kernels, memory copies, events) that execute in FIFO order. The tasks being executed in different streams are allowed to overlap and share device resources. Three types of streams exist in HIP, the default stream (or NULL stream), blocking streams, and non-blocking streams. The default stream is used to execute tasks (kernel launching and data transfers) that are not explicitly associated with any other stream. A blocking stream is synchronous, meaning that all operations submitted to the stream are guaranteed to complete in the order they were submitted. Both blocking and non-blocking streams can be created by the application programmer explicitly, and each differs in how they synchronize with the default stream. Tasks in the default stream will wait for all tasks previously submitted to blocking streams to be completed before executing. Similarly, tasks in blocking streams will wait for all tasks previously submitted to the default stream to be completed before executing. Non-blocking streams do not synchronize with the default stream. Based on the specification introduced above, the tasks located in different streams may be executed out of order or concurrently.

To be able to implement HIP streams with L0, L0 offers two possible modes of execution to dispatch tasks to a device.

- A command buffer abstraction (named command list), that will aggregate a series of tasks, and that can later be submitted to a command queue. The driver is free to optimize the execution of the command lists based on the synchronization expressed by the programmer;
- A low latency dispatch (named immediate command list) that will execute tasks as soon as they are ready (dependencies met) and able to be executed (available resources).

In HIPLZ, streams are implemented via LZQueue objects that wrap L0's immediate command lists (see Figure 5(a)). This mode of execution is better suited to implement the FIFO behavior of HIP streams. Synchronization considerations are still important to ensure barriers between tasks within streams as well as to correctly implement the HIP default stream semantics and synchronization. Nonetheless, using the immediate command list greatly reduces the overhead of managing individual command lists that would need to be submitted to command queues and which would need

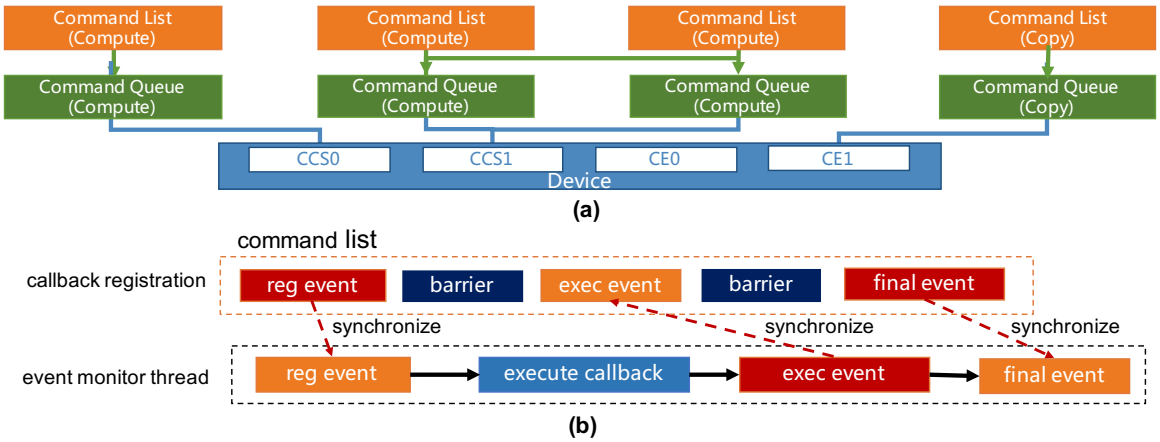

Figure 5 a. The basic HIPLZ Command List and Command Queue (image source: https://spec.oneapi.io/level-zero/latest/core/INTRO.html , Graphic courtesy of Intel Corporation, CCS refers to command streamer and CE refers to computing engine); b. The event order for executing callback

to be freed or recycled once the tasks they contain have finished executing. This technique eliminates the need for dedicated event tracking for each command list, irrespective of synchronization with other streams, and it also reduces the latency between task submission and execution.

The commands executed by the streams include: kernel functions, memory copy operations, host callbacks, and HIP event operations. The synchronization among different streams is supported via L0 events and their wait and signal APIs. The event object in a command list acts as either a barrier or signal, so two tasks running on different streams can use events to synchronize their executions.

Stream Synchronization Example: We use the host callback implementation as an example of how synchronization between and within streams in HIPLZ is implemented with L0. Figure 5(b) presents the workflow of host callback registration and invocation in HIPLZ. The callback function pointer is registered by the callback registration API, and a synchronization scheme is set up to program the callback using L0 events. This implementation of host callbacks requires a three point synchronization scheme. For each callback three L0 events are created, here called `reg event`, `exec event` and `final event`. Three synchronization primitives are added to the L0 immediate command list: a barrier that will signal the `reg event` once it is reached, a barrier that will wait for the `exec event` to be signaled by the host, and lastly a signal to `final event` signifying that the synchronization is complete and that the events can be freed (or recycled). In parallel, the event (host) monitor thread waits on `reg event` to be signaled, executes the callback, signals callback termination via `exec event` and waits on `final event` before releasing the resources.

3.5 | Memory management

HIPLZ supports several HIP memory management APIs, including `hipMalloc`, `hipMemcpy`, `hipMemcpyAsync`, and `hipFree`. Users can specify the allocation site, i.e. host memory, device memory or shared memory. Shared memory is based on the underlying GPU's support, and its reference is presented as a raw pointer that can be referred on both the host and device side. As mentioned in Section 3.4, in L0 the memory copy operation is implemented as a command that is queued on the command list and is executed via command queue.

HIP texture objects are special memory objects, and their support is similar to texture objects in CUDA; that is, the texture object is a first-class C++ object and can be passed as an argument just as if it is a pointer. HIPLZ provides `hipCreateTextureObject` and `hipDestroyTextureObject` to allocate and free texture objects.

The texture object is composed as an image buffer and a sampler object that operates on an image buffer. Since the image and sampler are defined as separate objects in L0 (i.e. `ze_image_t` and `ze_sampler_t`), we create the texture object as a C struct, as shown in Listing 1.

The `ze_image_t` and `ze_sampler_t` created via the L0 API are raw pointer values, thus they can be stored as `intptr_t` values. The actual texture operations are performed on reinterpreted structure fields, as shown in lines 6–9 of Listing 1, where a 2 dimensional texture of floating point values is sampled at coordinates `x` and `y`. This scheme relies on implementation specific behaviors of the Intel driver compiler.

Listing 1: HIP Texture Object Examples

```
1 typedef struct hipTextureObject_s {
2   intptr_t image;
3   intptr_t sampler;
4 } hipTextureObject_st, *hipTextureObject_t;
5
6 return read_imagef(
```

```
7   __builtin_astype(texObj->image, read_only image2d_t),
8   __builtin_astype(texObj->sampler, sampler_t),
9   (float2)(x, y)).x;
```

3.6 | Kernel and module management

HIP defines three different attributes for functions: __host__, __device__, and __global__. A __host__ decorated function is a function that is to be executed on the host, and functions without decorators will be considered host functions. A __device__ function will be callable from the device, and this decorator can be combined with __host__ to obtain a function that can execute on both the device and the host. A __global__ decorated function or kernel is callable from the host. The HIPLZ compiler translates the kernel and device functions to SPIR-V IL, and they are translated to device binary via vendor compiler during runtime. Each kernel function is wrapped into a LZKernel object and managed by a LZProgram object that presents the L0 module. The kernel launch is based on the L0 API and issues a command to the immediate command list.

HIPLZ also supports device global variables that are used for exchanging values between kernels and host code. Device global variables are supported in SPIR-V, and they can be interacted with from the host using L0. They can also be supported in OpenCL using Intel extensions.

3.7 | Device management

The device management in HIPLZ focuses device selection (hipSetDevice and hipGetDevice) and device property queries (i.e. hipGetDeviceProperties). From L0 standpoint, this means creating a L0 context containing all the devices, and exposing those devices through the hipGetDeviceCount. This allows sharing memory between devices using USM, without needing to register USM allocations between different contexts. Setting the current active device in HIPLZ changes the values for the default devices and default stream. HIP device properties are derived from the different device properties available in L0.

3.8 | DPC++ Interoperability

Interoperability between SYCL and HIP helps users maintain large heterogeneous code bases, and it also leverages the advantages of high performance libraries built by vendors (e.g. Intel oneMKL [20]). This has to solve two issues: 1) Execution context switching to ensure the same GPU device management and execution environment exists for both programming environments and 2) Data transfer between the HIPLZ and DPC++ programming environments. Both HIPLZ and DPC++ use L0 as the runtime driver for executing kernel functions on Intel GPUs, and use L0's driver object handles to maintain and exchange GPU device information, e.g. to pass an execution context object from HIPLZ to DPC++ and vise versa. To support data exchange, the unified shared memory (USM) mechanism is employed. Both HIPLZ and DPC++ use raw pointers to maintain the reference of the allocated memory from USM, and this simplifies memory reference passing between objects in each execution context.

3.9 | Kernel library

The implementation of the HIP math API in HIPLZ is based on OCML [21], which is a thin layer wrapping the OpenCL builtin math functions. We used this since many HIP math API have direct equivalent in OpenCL. The HIPLZ kernel library is an implementation of the HIP math API based on OCML [21].

3.10 | Graph Support

The functionality of HIP graph is similar to CUDA graph. It organizes computing tasks (i.e. kernel function and memory copy) as a workflow graph (hipGraph) [2]. The hipGraph can be launched via the hipGraphLaunch API, which means that multiple graph nodes (i.e. kernel function) can be launched via a single host side operation. Since L0 does not provide graph specific support, HIPLZ has to emulate graph operations.

The hipGraph is presented as a directed acyclic graph (DAG) (see Figure 6 (a)) in HIPLZ. It can be explicitly constructed via creating and adding graph nodes into it. The graph node can represent two kinds of tasks: computing kernel and memory copy. As shown in Figure 7, the two subclasses of hipGraphNode carry the properties for executing kernel functions and memory copies. The dependencies among graph nodes are also specified explicitly during node creation. Another way to construct hipGraph is to capture kernel invocations in an user-specified execution region,

[2]The implementation of HIP graph support is located at https://github.com/jz10/anl-gt-gpu wip branch

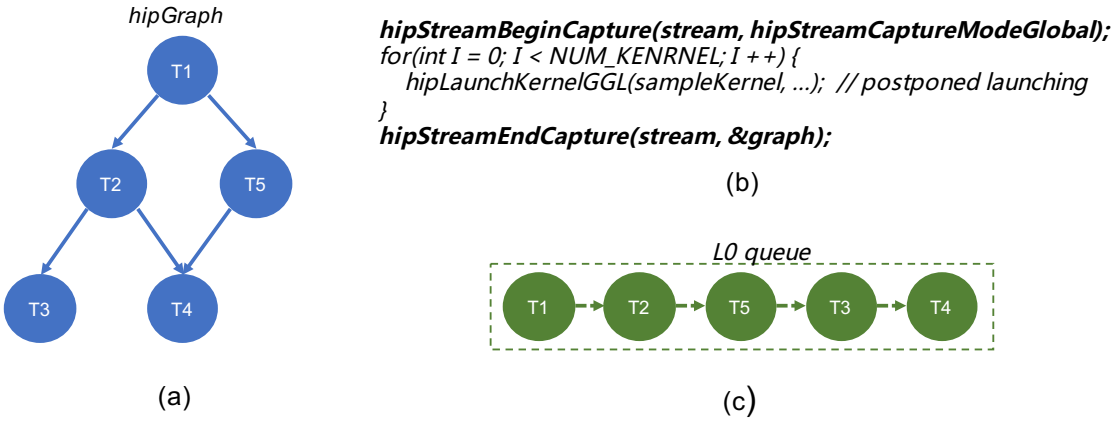

Figure 6 The hipGraph representation, construction and execution.

translate them to a graph node implicitly and add them into graph with sequential dependency. The execution region is defined via invocation of `hipStreamBeginCapture` and hipStreamEndCapture that captures the kernel functions scheduled on the given stream (see Figure 6(b)).

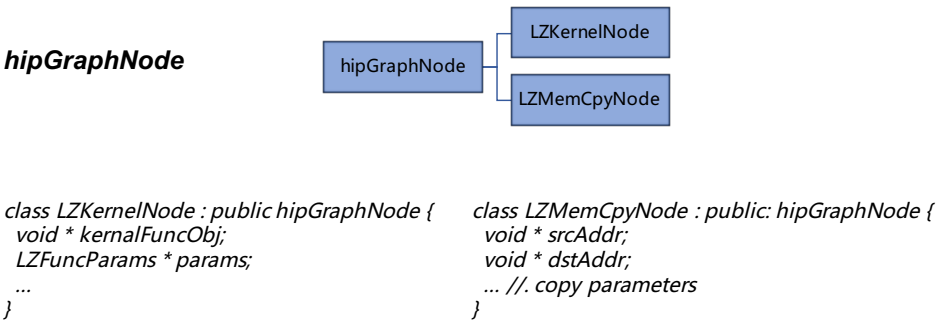

Figure 7 The `hipGraphNode` and its sub classes that represent kernel function and memory copy operation.

Graph launching executes/schedules a graph node on a given stream. To ensure the correctness of data dependencies, `HIPLZ` schedules graph nodes by breadth-first traversal on the graph(see Figure 6(c)).

3.11 | Discussion

The `L0` API presents a unique set of challenges that must be overcame to implement `HIP`.

`Program interface`: The `L0` API organization is very similar to `OpenCL`, especially for the objects that abstract the GPU device. However, `L0` is a lower level API than `OpenCL`, as it lacks a kernel language, object lifetime management, and also requires finer grained control on tasks using queues and command lists. This requires careful management of object lifetime in `HIPLZ`, and more involved synchronization schemes than in `HIPCL`.

`Capacity of conversion`: Users could benefit from a conversion guide that would describe the potential pitfalls that can arise from migrating to the `L0` API from other heterogeneous programming models.

`Lack of memory management mechanism`: `L0` provides memory allocation support for host, device and shared memory, but there is not an uniform way to manage the allocated memory object, so it is very easy to introduce a scenario that mistakenly uses the memory objects.

`Lack of thread safety`: There are many runtime objects and APIs that are not thread-safe in the `L0` specification, so mutual exclusion is employed for all relevant API call sites in `HIPLZ` using mutexes.

4 | EVALUATION

In this section we introduce the hardware we evaluated HIPLZ on, an overview of the tests, and then the results.

4.1 | Employed GPU Systems

In this study we evaluated HIPLZ on two Intel GPUs:

- Intel Gen9[22] on the JLSE cluster[23]. The Gen9 is an integrated GPU which is available in commercial Intel products.The Gen9 nodes are composed of a Intel Xeon Processor E3-1585 v5 CPU and Iris Pro Graphics P580 (GT4e) Gen9 GPU[22]. With a peak clock rate of 1.15 GHz and 72 execution units which can each perform 2 double precision or 8 single precision FMAs per clock. The Gen9 GPU has a peak theoretical double (single) precision performance of 331.2 GFlop/s (1324.8 GFlop/s). With 2 channels of DDR4-2133, the peak theoretical DRAM bandwidth is 34.1 GB/s (2.133 Ghz * 8 bytes/clock * 2).
- Intel UHD Graphics 770, which is integrated into the Intel Core i9-12900K CPU, has a peak clock rate of 1.45GHz and 32 execution units. Its peak theoretical single precision performance is 742.4 GFlop/s. With 4 channels of DDR4 shared with i9 processor, the peak theoretical DRAM bandwith is 76.8 GB/s.

4.2 | Overview of Tests

To evaluate HIPLZ we collected a repository of HPC-relevant benchmarks, mini-apps, frameworks, and applications hosted on GitHub[24]. The 50 selected codes include 2 benchmarks, 7 mini-apps (2 for BerkeleyGW), 1 application, and 40 HIP examples. The codes are listed in Table 4.

4.3 | Results

We first discuss the performance results of the benchmarks and then the overall build/run/pass rate for the tests. For the measurements presented here we used:

- HIPLZ version: From HIPLZ, branch launch_bounds, commit cbf2260
- HIPCL version: From a fork of HIPCL, https://github.com/Kerilk/hipcl, in branch fence, commit dd39656
- OpenCL version: Intel OpenCL 3.0 NEO, driver version 22.02 for Gen9 and 23.05 for UHD Graphics 770;
- Intel Compute Runtime NEO version: driver version 22.02 for Gen9 and 23.05 for UHD Graphics 770;
- hip-test-suite[24], commit 3b19290 .

We note that HIPLZ has a differently named compiler driver than AMD HIP. HIPLZ uses clang++, while HIP uses hipcc.

4.3.1 | Benchmark and Performance Results

To evaluate the performance of the HIPLZ implementation, we consider the tests in the hip-test-suite benchmarks subdirectory. The two tests in this subdirectory (ERT and BabelStream) measure the memory bandwidth and/or the peak performance of the system. The results are summarized in Table 2. By comparing the memory bandwidth and floating point performance, HIPLZ performs similarly to the OpenCL port, near the theoretical peaks of the Gen9 device.

For the memory bandwidth measurements, we expect the code to be able to reach 80% of the theoretical memory bandwidth of the hardware. As shown in Table 2, with the HIPLZ implementation, the HIP BabelStream port measures a bandwidth of 27.76 GB/s on Gen9 and 63.17GB/s on UHD Graphics 770, the HIP ERT port measures 25.84 GB/s on Gen9 and 63.22GB/s on UHD Graphics 770. These are both near 80% of the theoretical bandwidth of the employed hardware.

For the floating point performance measurements, with our HIPLZ implementation, ERT measured 303.22 Gflop/s double precision peak performance and 1240.69 Gflop/s single precision peak performance on Gen9. The measured double precision peak performance is about 91% of the theoretical value, and the measured single precision peak performance is about 94% of the theoretical value. Since UHD Graphics 770 does not have double precison support at the hardware level, we only measured performance for single precision, The single precision peak performance measured with ERT measured 715.3Gflop/s peak performance, that is about 96% of the theoretical value.

Table 2 Efficiency Evaluation of HIPLZ with Comparable APIs.

Test (measurement)	Gen9			UHD Graphics 770		
	HIPCL	OpenCL	HIPLZ	HIPCL	OpenCL	HIPLZ
DRAM Bandwidth (GB/s) (from Triad BabelStream)	26.13	26.07	27.42	62.17	62.2	63.17
DRAM Bandwidth (GB/s) (from ERT)	25.48	25.77	25.84	62.35	63.16	63.22
FP64 peak (Gflop/s) (from ERT)	301.66	299.12	303.22	N/A	N/A	N/A
FP32 peak (Gflop/s) (from ERT)	1235.39	1184.91	1240.69	713.9	702.5	715.3

Details about how this test was compiled and run can be found in *https://github.com/jz10/hip-test_suite*.

Several of the tests in the proxies and HIP-Examples subfolders also have HIP and OpenCL ports and measure performance metrics. We also compare several of these performance metrics in Table 3. As shown in Table 3, the performance achieved by HIPLZ on Intel Gen9 GPUs and UHD Graphics 770 is similar to that achieved by the OpenCL port for additional tests.

Table 3 Performance metrics from additional tests

Test (measurement)	Gen9		UHD Graphics 770	
	HIPLZ	OpenCL	HIPLZ	OpenCL
su3_bench (Total GFLOP/s)	28.813	28.6	15.35	15.29
strided-access (Stride 2 bandwidth, GB/s)	21.573	22.0791	50.77	53.12
GPU-STREAM (Triad bandwidth GB/s)	27.8	26.5	67.38	66.17
mixbench (Compute iter 256, Read-only Iops/bytes)	413.55	414.55	239.87	239.62

We also note that although add4 and cuda-stream do not have OpenCL ports in the test suite, they measure memory bandwidth. The bandwidth reported is similar to that reported by the OpenCL and HIP ports of Babelstream in Table 2, so we can consider them achieving the expected performance.

4.3.2 | Overall Results

The results are shown in Table 4. Out of 50 tests, 48/50 (96 %) compile without errors, 44/50 (88%) compile and run without crashing, and 40/50 (80%) compile, run to completion, and give the correct answer.

4.3.3 | Discussion of Results

We categorize the failures (shown in Table 4) into 3 types:

- Building failures: this was due to dependence on external libraries that are not currently supported by HIPLZ (cholla (dependence on hipfft), KokkosDslash (dependence on kokkos)), unimplemented functions (adept-proxy (three-argument shuffles)), and compiler errors (BerkeleyGW-FF, GridMini);
- Runtime failures: this was due to acquireing device memory that exceeds the capacity of GPU memory (gpu-burn);
- Verification failures: the computing results were incorrect (BerkeleyGW-GPP and RSBench).

5 | RELATED WORK

Many of the programming language systems that support GPU offloading translate high-level programming language constructs to heterogeneous programming model APIs. Typical examples are OpenMP [25] and OpenACC [26]. Compilers which support OpenMP or OpenACC translate high-level pragma-based abstractions to lower-level (for example, CUDA driver or OpenCL) calls. This allows code using OpenMP or OpenACC to target a wide variety of hardware as long as the compiler lowers the abstractions into lower-level representations that the underlying runtime can ingest. This representation is bundled in a fat binary-based executable, in which the same binary embeds both host and device code. This allows the

Table 4 Detailed results of building, running, and checking correctness for the tests

Test	Build	Run	Correct	Test	Build	Run	Correct
BabelStream	Y	Y	Y	mixbench	Y	Y	Y
cs-roofline-toolkit	Y	Y	Y	BinomialOption	Y	Y	Y
cholla	N			BitonicSort	Y	Y	Y
KokkosDslash	N			FastWalshTransform	Y	Y	Y
su3_bench	Y	Y	Y	FloydWarshall	Y	Y	Y
BerkeleyGW-FF	N			HelloWorld	Y	Y	Y
BerkeleyGW-GPP	Y	N		Histogram	Y	Y	Y
add4	Y	Y	Y	MatrixMultiplication	Y	Y	Y
cuda-stream	Y	Y	Y	PrefixSum	Y	Y	Y
gpu-burn	Y	N		RecursiveGaussian	Y	Y	Y
mini-nbody	Y	Y	Y	SimpleConvolution	Y	Y	Y
reduction	Y	Y	Y	dct	Y	Y	Y
rodinia_3.0 (18 tests)	Y (18)	Y (18)	Y (16)	dwtHaar1D	Y	Y	Y
rtm8	Y	Y	Y	adept-proxy	N		
strided-access	Y	Y	Y	RSBench	Y	Y	N
vectorAdd	Y	Y	Y	GridMini	N		
GPU-STREAM	Y	Y	Y				

device code to be recompiled or optimized when the driver is updated, without having to rebuild the application. The usage of fat binaries brings the advantage for application deployment, i.e. no need to maintain separated binary or source code (host and device) and link them together for execution. LLVM/Clang[27] uses PTX as the intermediate language (IL) for the `CUDA` driver. Intel OpenMP compiler makes another choice and uses SPIR-V as IL in order to target their `OpenCL` or `L0` based GPU backends[28,29,30]. The approach in `HIPLZ` is similar, although we implement the `HIP` API and not pragma-based approaches, and we use SPIR-V as the intermediary representation.

Since the API pattern of `HIP` is highly similar to `CUDA`'s, there have been some pattern match based tools[14] that directly translate `CUDA` API calls to `HIP` API calls, thus enables the application porting from `CUDA` to `HIP`.

Different approaches exist to bridge programming models to `L0`: for example ZLUDA[31] is a demonstrator showcasing running unmodified `CUDA` applications on top of `L0` by providing a `L0` APIs based implemention of `CUDA` driver APIs, and converting NVIDIA PTX[32] to SPIR-V at runtime. Although ZLUDA only supports a limited subset of applications, but it does showcase promising performance on those applications.

Another well known project bridging several programming models to OpenCL is pocl[33]. pocl implements OpenCL for NVIDIA GPUs on top of `CUDA`, AMD GPUs on top of HSA and supports CPU devices as well through the Posix Threads programming API.

OpenSYCL[34?,35,36] is a `SYCL` implementation that leverages existing heterogeneous programming model such as `CUDA`, `HIP` to support different GPU architectures. It also provides a support for Intel GPUs via using `L0` as one of its modular runtime backend plugin.

In this paper, we present `HIPLZ` that bridges `HIP` to `L0` to supports Intel GPU. It directly uses `L0` APIs to implement `HIP` APIs' functionalities and `SPIR-V` as intermediate language (IL) to represent the kernel functions, and the host side executable binary invokes driver compiler to translate IL to device binary. Compared with HIPCL , the advantage of this approach is to leverage L0 APIs' flexibility to get direct control of accelerator device, e.g. immediate command list, unified shared memory. Compared with ZLUDA, HIPLZ provides better coverage of GPU programming model's features (e.g. asynchronous execution).

6 | CONCLUSION

In this paper, we introduced the design and implementation of `HIPLZ`, a compilation and runtime system that allows `HIP` code to run on Intel GPUs. It uses the `L0` API to implement the `HIP` API's functionalities and `SPIR-V` as the IL to represent the kernel functions. To the best of our knowledge, `HIPLZ` is the first compiler and runtime system that allows `HIP` code to run on Intel GPUs by using `L0`.

`HIPLZ` successfully compiled and produced correct results on Intel Gen9 GPU and Intel UHD Graphics 770 for more than 35 `HIP` test cases and mini-apps. In terms of performance, we ran two performance benchmarks using `HIPLZ` and were able to achieve approximately the same peak values as `OpenCL`, demonstrating that `HIPLZ` produces code that can effectively use the Intel GPU hardware. Future work will focus on extending performance for more applications and interoperability with other programming models like OpenMP, Chapel and Python.

7 | ACKNOWLEDGEMENTS*

This work was supported by the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357, and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration). We also gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

References

1. Top 500 List. https://www.top500.org/lists/top500/list/2021/11/.
2. Aurora. https://www.alcf.anl.gov/aurora.
3. SuperMUC-NG. https://www.hpcwire.com/2021/05/05/lrz-announces-new-phase-of-supermuc-ng-supercomputer-with-intels-ponte-vecchio-gpu/.
4. Frontier. https://www.olcf.ornl.gov/frontier/.
5. Next-Generation AMD EPYC™ CPUs and Radeon™ Instinct GPUs Enable El Capitan Supercomputer at Lawrence Livermore National Laboratory to Break 2 Exaflops Barrier. https://www.amd.com/en/press-releases/2020-03-04-next-generation-amd-epyc-cpus-and-radeon-instinct-gpus-enable-el-capitan.
6. EL Captain Supercomputer. https://www.hpe.com/us/en/compute/hpc/cray/doe-el-capitan-press-release.html.
7. One of the world's mightiest supercomputers, LUMI, will lift European research and competitiveness to a new level and promotes green transition. hhttps://www.csc.fi/en/-/lumi-one-of-the-worlds-mightiest-supercomputers.
8. LUMI Supercomputer. hhttps://www.csc.fi/en/-/lumi-one-of-the-worlds-mightiest-supercomputers.
9. TensorFlow. https://www.tensorflow.org/.
10. PyTorch. https://pytorch.org/.
11. Parasyris K, Pennycook J, Hartley T. Intel Level Zero: An Open and Scalable Low-Level Interface for Modern Architectures. In: IEEE. ; 2018: 673–678.
12. Heterogeneous compute Interface for Portability (HIP). https://rocmdocs.amd.com/en/latest/Programming_Guides/Programming-Guides.html.
13. Nvidia CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit.
14. HIPFY. https://github.com/ROCm-Developer-Tools/HIPIFY.
15. The Industry Open Standard Intermediate Language for Parallel Compute and Graphics. https://www.khronos.org/spir/.
16. Gaster B, Howes L, Kaeli DR, Mistry P, Schaa D, McIntosh-Smith S. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Micro* 2011; 31(5): 28–40.
17. Babej M, Jääskeläinen P. HIPCL: Tool for Porting CUDA Applications to Advanced OpenCL Platforms Through HIP. In: IWOCL '20. ; 2020
18. SYCL. https://www.khronos.org/sycl/.
19. Wu P, Liu Y, Gao GR. Exploring the Performance Portability of SyCL for Heterogeneous Computing. In: IEEE. ; 2018: 444–455.
20. oneMKL. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.
21. OCML. https://github.com/RadeonOpenCompute/ROCm-Device-Libs/blob/amd-stg-open/doc/OCML.md.
22. The Compute Architecture of Intel Processor Graphics Gen9. https://software.intel.com/content/dam/develop/external/us/en/documents/the-compute-architecture-of-intel-processor-graphics-gen9-v1d0.pdf.

23. Argonne Joint Lab for System Evaluation (JLSE). https://www.jlse.anl.gov.
24. HIP Test Set. https://github.com/jz10/hip-test_suite.
25. Nugteren C, Diener M, Corporaal H. OpenMP Offloading on Heterogeneous Systems: A Comprehensive Survey. *arXiv preprint arXiv:2002.10333* 2020.
26. OpenACC. https://www.openacc.org/.
27. Clang Compiler. http://clang.llvm.org/.
28. Intel oneAPI. https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html.
29. DPC++. https://www.intel.com/content/www/us/en/developer/tools/oneapi/data-parallel-c-plus-plus.html.
30. Intel® oneAPI DPC/C++ Compiler and Intel® Fortran Compiler . https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-cpp-fortran-compiler-openmp/top.html.
31. Janik A. ZLUDA: CUDA on Intel GPUs. https://github.com/vosen/ZLUDA.
32. Parallel Thread Execution (PTX) and ISA. https://docs.nvidia.com/cuda/parallel-thread-execution/.
33. Jääskeläinen P, al e. Pocl: A Performance-Portable OpenCL Implementation. *Int. J. Parallel Program.* 2015; 43(5): 752–785. doi: 10.1007/s10766-014-0320-y
34. Gentilhomme RP, ElGindy H, Hästbacka M, et al. hipSYCL: A Comprehensive Foundation for Heterogeneous Programming. In: IEEE. ; 2019: 1–13.
35. Alpay A, Heuveline V. One Pass to Bind Them: The First Single-Pass SYCL Compiler with Unified Code Representation Across Backends. In: ACM; 2023: 7:1–7:12
36. Homerding B, Tramm JR. Evaluating the Performance of the hipSYCL Toolchain for HPC Kernels on NVIDIA V100 GPUs. In: ACM; 2020: 16:1–16:7
37. Intel Level Zero Spec. http://spec.oneapi.io/level-zero/latest/index.html.
38. Open Standard for Parallel Programming of Heterogeneous Systems(OpenCL). https://www.khronos.org/opencl/.
39. OpenMP. https://www.openmp.org/.
40. OpenSYCL. https://github.com/OpenSYCL/OpenSYCL.
41. Alpay A, Soproni B, Wünsche H, Heuveline V. Exploring the possibility of a hipSYCL-based implementation of oneAPI. In: ACM; 2022: 10:1–10:12