AMD ROCm™

HIP Programming Guide

Publication #	**1.0**	Revision:	**0802**
Issue Date:	**August 2021**		

[This page left blank intentionally]

Table of Contents

Chapter 1 Introduction

HIP is a C++ Runtime API and kernel language that allows developers to create portable applications for AMD and NVIDIA GPUs from a single source code.

1.1 Features

The key features are:

- HIP has little or no performance impact over coding directly in the CUDA mode.
- HIP allows coding in a single-source C++ programming language, including features such as
    - Templates
    - C++11 lambdas
    - Classes
    - namespaces.
- HIP allows developers to use the development environment and tools on each target platform.
- The HIPify tool to automatically convert sources from CUDA to HIP.
- Developers can specialize in the platform (CUDA or AMD) to tune for performance.

New projects can be developed directly in the portable HIP C++ language and run on either NVIDIA or AMD platforms. Additionally, HIP provides porting tools, making it easy to port existing CUDA codes to the HIP layer, with no loss of performance compared to the original CUDA application. Thus, you can compile the HIP source code to run on either platform and isolate some features to a specific platform using conditional compilation.

NOTE: HIP is not intended to be a drop-in replacement for CUDA, and developers should expect to do some manual coding and performance tuning work to complete the port.

1.2 Accessing HIP

HIP is open source in GitHub and the repository maintains the following branches.

- Main branch: This is the stable branch and is up to date with the latest release branch. For example, if the latest HIP release is rocm-4.1.x, the main repository is based on this release.

- Release branch: The release branch corresponds to each ROCM release listed with release tags, such as rocm-4.0.x, rocm-4.1.x, and others.

For more information, refer to *https://github.com/ROCm-Developer-Tools/HIP*

1.2.1 Release Tagging

HIP releases consist of naming conventions for each ROCM release to help differentiate them. For example, **rocm *x.yy***, where *x.yy* reflects the ROCm release number.

1.3 HIP Portability and Compiler Technology

HIP C++ code can be compiled with either AMD or NVIDIA GPUs. On the AMD ROCm platform, HIP provides a header and runtime library built on top of the HIP-Clang compiler. The HIP runtime implements HIP streams, events, and memory APIs, and is an object library that is linked with the application.

On the NVIDIA CUDA platform, HIP provides a header file, which translates from the HIP runtime APIs to CUDA runtime APIs. The header file contains mostly inline functions and, thus, has a very low overhead developers coding in HIP should expect the same performance as coding in native CUDA. The code is then compiled with nvcc, the standard C++ compiler provided with the CUDA SDK. Developers can use any tools supported by the CUDA SDK including the CUDA profiler and debugger.

Thus, HIP provides source portability to either platform. HIP provides the hipcc compiler driver which will call the appropriate toolchain depending on the desired platform. The source code for all headers and the library implementation is available on GitHub.

Chapter 2 Installing HIP

2.1 Installing Pre-built Packages

You can install HIP with the package manager and the pre-built binary packages for your platform.

2.2 Prerequisites

You can develop HIP code on the AMD ROCm platform using the HIP-Clang compiler and on a CUDA platform with nvcc.

2.3 AMD Platform

HIP is installed with the ROCm driver package. For more information on HIP installation instructions, refer to the ROCm Installation Guide at

https://rocmdocs.amd.com/en/latest/Installation_Guide/Installation-Guide.html

Note, HIP-Clang is the compiler for compiling HIP programs on the AMD platform.

2.4 NVIDIA Platform

HIP-nvcc is the compiler for HIP program compilation on the NVIDIA platform.

- Add the ROCm package server to your system as per the OS-specific guide available *here*.
- Install the "hip-nvcc" package. This will install CUDA SDK and the HIP porting layer.

```
apt-get install hip-nvcc
```

Default Paths and Environment Variables

- By default, HIP looks for CUDA SDK in /usr/local/cuda (can be overriden by setting CUDA_PATH env variable).
- By default, HIP is installed into /opt/rocm/hip (can be overridden by setting HIP_PATH environment variable).
- Optionally, consider adding /opt/rocm/bin to your path to make it easier to use the tools.

2.5 Building HIP from Source

2.5.1 Build ROCclr

ROCclr is defined on AMD platform that HIP use Radeon Open Compute Common Language Runtime (ROCclr), which is a virtual device interface that HIP runtimes interact with different backends.

For more information, see *https://github.com/ROCm-Developer-Tools/ROCclr*

```
git clone -b rocm-4.3.x https://github.com/ROCm-Developer-Tools/ROCclr.git
export ROCclr_DIR="$(readlink -f ROCclr)"
git clone -b rocm-4.3.x https://github.com/RadeonOpenCompute/ROCm-OpenCL-Runtime.git
export OPENCL_DIR="$(readlink -f ROCm-OpenCL-Runtime)"
cd "$ROCclr_DIR"
mkdir -p build;cd build
cmake -DOPENCL_DIR="$OPENCL_DIR" -DCMAKE_INSTALL_PREFIX=/opt/rocm/rocclr ..
make -j
sudo make install
```

2.5.2 Build HIP

```
git clone -b rocm-4.3.x https://github.com/ROCm-Developer-Tools/HIP.git
export HIP_DIR="$(readlink -f HIP)"
cd "$HIP_DIR"
mkdir -p build; cd build
cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_PREFIX_PATH="$ROCclr_DIR/build;/opt/rocm/" -
DCMAKE_INSTALL_PREFIX=</where/to/install/hip> ..
make -j
sudo make install
```

2.5.3 Default paths and environment variables

- By default, HIP looks for HSA in /opt/rocm/hsa (can be overridden by setting HSA_PATH environment variable).
- By default, HIP is installed into /opt/rocm/hip (can be overridden by setting HIP_PATH environment variable).
- By default, HIP looks for clang in /opt/rocm/llvm/bin (can be overridden by setting HIP_CLANG_PATH environment variable)
- By default, HIP looks for device library in /opt/rocm/lib (can be overridden by setting DEVICE_LIB_PATH environment variable).
- Optionally, consider adding /opt/rocm/bin to your PATH to make it easier to use the tools.
- Optionally, set HIPCC_VERBOSE=7 to output the command line for compilation.

After installation, ensure *HIP_PATH* points to */where/to/install/hip.*

2.6 Verifying HIP Installation

1. Run hipconfig. Note, the instructions below assume a default installation path):

```
/opt/rocm/bin/hipconfig --full
```

1. Compile and run the square sample from:
   https://github.com/ROCm-Developer-Tools/HIP/tree/main/samples/0_Intro/square

Chapter 3 Programming with HIP

3.1 HIP Terminology

Term	Description
host, host cpu	Executes the HIP runtime API and is capable of initiating kernel launches to one or more devices.
default device	Each host thread maintains a "default device". Most HIP runtime APIs (including memory allocation, copy commands, kernel launches) do not use accept an explicit device argument but instead implicitly use the default device. The default device can be set with hipSetDevice.
active host thread	Thread running the HIP APIs.
HIP-Clang	Heterogeneous AMDGPU Compiler, with its capability to compile HIP programs on the AMD platform. *https://github.com/RadeonOpenCompute/llvm-project*
hipify tools	Tools to convert CUDA code to portable C++ code (https://github.com/ROCm-Developer-Tools/HIPIFY).
ROCclr	A virtual device interface that computes runtimes interact with different backends such as ROCr on Linux or PAL on Windows. The ROCclr is an abstraction layer allowing runtimes to work on both OSes without much effort. For more information, see *https://github.com/ROCm-Developer-Tools/ROCclr*
hipconfig	Tool to report various configuration properties of the target platform.
nvcc	nvcc compiler

3.2 Getting Started with HIP API

3.2.1 HIP API Overview

The HIP API includes functions such as hipMalloc, hipMemcpy, and hipFree. Programmers familiar with CUDA will also be able to quickly learn and start coding with the HIP API. Compute kernels are launched with the 'hipLaunchKernel's macro call.

For more information, refer to the HIP API Guide at

https://github.com/RadeonOpenCompute/ROCm/blob/master/AMD_HIP_API_Guide_v4.3.pdf

3.2.2 HIP API Examples

3.2.2.1 Example 1

Here is an example showing a snippet of the HIP API code:

```
hipMalloc(&A_d, Nbytes));
hipMalloc(&C_d, Nbytes));
hipMemcpy(A_d, A_h, Nbytes, hipMemcpyHostToDevice);
const unsigned blocks = 512;
const unsigned threadsPerBlock = 256;
hipLaunchKernel(vector_square,   /* compute kernel*/
                dim3(blocks), dim3(threadsPerBlock), 0/*dynamic shared*/, 0/*stream*/,
/* launch config*/
                C_d, A_d, N);  /* arguments to the compute kernel */
hipMemcpy(C_h, C_d, Nbytes, hipMemcpyDeviceToHost);
```

The HIP kernel language defines builtins for determining grid and block coordinates, math functions, short vectors, atomics, and timer functions. It also specifies additional defines and keywords for function types, address spaces, and optimization controls. For a detailed description, see *Section 3.4* in this document.

3.2.2.2 Example 2

Here's an example of defining a simple 'vector_square' kernel.

```
template <typename T>
__global__ void
vector_square(T *C_d, const T *A_d, size_t N)
{
    size_t offset = (blockIdx.x * blockDim.x + threadIdx.x);
    size_t stride = blockDim.x * gridDim.x;
    for (size_t i=offset; i<N; i+=stride) {
        C_d[i] = A_d[i] * A_d[i];
    }
}
```

The HIP Runtime API code and compute kernel definition can exist in the same source file - HIP takes care of generating host and device code appropriately.

3.2.2.3 More HIP Examples

For more examples to learn and use HIP, see

https://github.com/ROCm-Developer-Tools/HIP/tree/main/samples

3.3 Introduction to Memory Allocation

3.3.1 Host Memory

hipHostMalloc allocates pinned host memory which is mapped into the address space of all GPUs in the system. There are two use cases for this host memory:

- Faster HostToDevice and DeviceToHost Data Transfers: The runtime tracks the hipHostMalloc allocations and can avoid some of the setup required for regular unpinned memory. For exact measurements on a specific system, experiment with --unpinned and --pinned switches for the hipBusBandwidth tool.

- Zero-Copy GPU Access: GPU can directly access the host memory over the CPU/GPU interconnect, without need to copy the data. This avoids the need for the copy, but during the kernel access each memory access must traverse the interconnect, which can be tens of times slower than accessing the GPU's local device memory. Zero-copy memory can be a good choice when the memory accesses are infrequent (perhaps only once). Zero-copy memory is typically "Coherent" and thus not cached by the GPU but this can be overridden if desired and is explained in more detail below.

3.3.2 Memory allocation flags

hipHostMalloc always sets the hipHostMallocPortable and hipHostMallocMapped flags. Both usage models described above use the same allocation flags, and the difference is in how the surrounding code uses the host memory. See the hipHostMalloc API for more information.

hipHostMallocNumaUser is the flag to allow host memory allocation to follow NUMA policy set by the user.

See hipHostMalloc API in the HIP API guide for more information.

https://github.com/RadeonOpenCompute/ROCm/blob/master/AMD_HIP_API_Guide_v4.3.pdf

3.3.3 NUMA-aware host memory allocation

The Non-Uniform Memory Architecture (NUMA) policy determines how memory is allocated and selects a CPU closest to each GPU.

NUMA also measures the distance between the GPU and CPU devices. By default, each GPU selects a Numa CPU node that has the least NUMA distance between them; the host memory is automatically allocated closest to the memory pool of the NUMA node of the current GPU device.

Note, using the *hipSetDevice* API with a different GPU provides access to the host allocation. However, it may have a longer NUMA distance.

3.3.4 Managed memory allocation

HIP now supports and automatically manages Heterogeneous Memory Management (HMM) allocation. The HIP application performs a capability check before making the managed memory API call *hipMallocManaged*.

NOTE: The _*managed*_ keyword is unsupported currently.

For example,

```
int managed_memory = 0;
HIPCHECK(hipDeviceGetAttribute(&managed_memory,
  hipDeviceAttributeManagedMemory,p_gpuDevice));

if (!managed_memory ) {
  printf ("info: managed memory access not supported on the device %d\n
Skipped\n", p_gpuDevice);
}
else {
  HIPCHECK(hipSetDevice(p_gpuDevice));
  HIPCHECK(hipMallocManaged(&Hmm, N * sizeof(T)));
. . .
}
```

The managed memory capability check may not be necessary; however, if HMM is not supported, then managed malloc will fall back to using system memory. Other managed memory API calls will, then, have undefined behavior.

For more details on managed memory APIs, refer to the HIP API Guide at,

https://github.com/RadeonOpenCompute/ROCm/blob/master/AMD_HIP_API_Guide_v4.3.pdf

3.3.5 HIP Stream Memory Operations

HIP supports Stream Memory Operations to enable direct synchronization between Network Nodes and GPU. The following APIs are added:

- hipStreamWaitValue32
- hipStreamWaitValue64
- hipStreamWriteValue32
- hipStreamWriteValue64

For more details, refer to the HIP API Guide at,

https://github.com/RadeonOpenCompute/ROCm/blob/master/AMD_HIP_API_Guide_v4.3.pdf

3.3.6 Coherency Controls

ROCm defines two coherency options for host memory:

- Coherent memory: Supports fine-grain synchronization while the kernel is running. For example, a kernel can perform atomic operations that are visible to the host CPU or to other (peer) GPUs. Synchronization instructions include threadfence_system and C++11-style atomic operations. However, coherent memory cannot be cached by the GPU and thus may have lower performance.

- Non-coherent memory: Can be cached by GPU but cannot support synchronization while the kernel is running. Non-coherent memory can be optionally synchronized only at command (end-of-kernel or copy command) boundaries. This memory is appropriate for high-performance access when fine-grain synchronization is not required.

HIP provides the developer with controls to select which type of memory is used via allocation flags passed to hipHostMalloc and the HIP_HOST_COHERENT environment variable. By default, the environment variable HIP_HOST_COHERENT is set to 0 in HIP.

- hipHostMallocCoherent=0, hipHostMallocNonCoherent=0: Use HIP_HOST_COHERENT environment variable:
    - If HIP_HOST_COHERENT is defined as 1, the host memory allocation is coherent.
    - If HIP_HOST_COHERENT is not defined, or defined as 0, the host memory allocation is non-coherent.
- hipHostMallocCoherent=1, hipHostMallocNonCoherent=0: The host memory allocation will be coherent. HIP_HOST_COHERENT env variable is ignored.
- hipHostMallocCoherent=0, hipHostMallocNonCoherent=1: The host memory allocation will be non-coherent. HIP_HOST_COHERENT env variable is ignored.
- hipHostMallocCoherent=1, hipHostMallocNonCoherent=1: Illegal.

3.3.7 Visibility of Zero-Copy Host Memory

The coherent and non-coherent host memory visibility is described in the table below. Note, the coherent host memory is automatically visible at synchronization points.

HIP API	Synchronization Effect	Fence	Coherent Host Memory Visibility	Non-Coherent Host Memory Visibility
hipStreamSynchronize	host waits for all commands in the specified stream to complete	system-scope release	yes	yes
hipDeviceSynchronize	host waits for all commands in all streams on the specified device to complete	system-scope release	yes	yes
hipEventSynchronize	host waits for the specified event to complete	device-scope release	yes	depends - see the description below
hipStreamWaitEvent	stream waits for the specified event to complete	none	yes	no

3.3.7.1 hipEventSynchronize

Developers can control the release scope for hipEvents. By default, the GPU performs a device-scope acquire and release operation with each recorded event. This will make host and device memory visible to other commands executing on the same device.

A stronger system-level fence can be specified when the event is created with hipEventCreateWithFlags.

hipEventReleaseToSystem: Perform a system-scope release operation when the event is recorded. This will make both Coherent and Non-Coherent host memory visible to other agents in the system but may involve heavyweight operations such as cache flushing. Coherent memory will typically use lighter-weight in-kernel synchronization mechanisms, such as an atomic operation, and, thus, do not need to use **hipEventReleaseToSystem**.

hipEventDisableTiming: Events created with this flag do not record profiling data, thus, providing optimal performance if used for synchronization.

NOTE: For HIP events in kernel dispatch using *hipExtLaunchKernelGGL/hipExtLaunchKernel*, events passed in the API are not explicitly recorded and should only be used to get elapsed time for that specific launch.

In case events are used across multiple dispatches, for example, *start* and *stop* events from different *hipExtLaunchKernelGGL/hipExtLaunchKernel* calls, they will be treated as invalid unrecorded events, and HIP will display an error *"hipErrorInvalidHandle"* from *hipEventElapsedTime*.

Summary and Recommendations

- Coherent host memory is the default and is the easiest to use since the memory is visible to the CPU at typical synchronization points. This memory allows in-kernel synchronization commands such as threadfence_system to work transparently.
- HIP/ROCm also supports the ability to cache host memory in the GPU using the "Non-Coherent" host memory allocations. This can provide a performance benefit, but care must be taken to use the correct synchronization.

3.3.7.2 Device-Side Malloc

HIP-Clang currently does not support device-side malloc and free.

3.3.7.3 Use of Long Double Type

In HIP-Clang, the long double type is an 80-bit extended precision format for x86_64, which is not supported by AMDGPU. HIP-Clang treats long double type as IEEE double type for AMDGPU. Using long double type in HIP source code will not cause an issue as long as data of long double type is not transferred between host and device. However, the long double type should not be used as kernel argument type.

3.3.7.4 FMA and Contractions

By default, HIP-Clang assumes -ffp-contract=fast. For x86_64, FMA is off by default since the generic x86_64 target does not support FMA by default. To turn on FMA on x86_64, either use -mfma or -march=native on CPU's supporting FMA.

When contractions are enabled and the CPU has not enabled FMA instructions, the GPU can produce different numerical results than the CPU for expressions that can be contracted.

3.3.7.5 Creating Static Libraries

You can create the following types of static libraries with HIP-Clang:

- Static libraries that do not export device functions, and, therefore, can only be launched through host functions within the same library. The advantage of this type of library is that it can be linked with a non-hipcc compiler such as GCC.

These libraries contain host objects with the device code embedded as fat binaries. It can only be generated using the flag *--emit-static-lib*.

For example,

```
hipcc hipOptLibrary.cpp --emit-static-lib -fPIC -o libHipOptLibrary.a
```

- Static libraries that export device functions within the library to be linked by other code objects or libraries. However, this requires using hipcc as the linker. These libraries contain relocatable device objects and are created using ar.

  For example,

```
hipcc hipDevice.cpp -c -fgpu-rdc -o hipDevice.o
ar rcsD libHipDevice.a hipDevice.o
```

For more information, see *samples/2_Cookbook/15_static_lib.*

3.4 HIP Kernel Language

HIP provides a C++ syntax that is suitable for compiling most code that commonly appears in compute kernels, including classes, namespaces, operator overloading, templates, and more. Additionally, it defines other language features designed specifically to target accelerators, such as the following:

- A kernel-launch syntax that uses standard C++, resembles a function call, and is portable to all HIP targets
- Short-vector headers that can serve on a host or a device
- Math functions resembling those in the "math.h" header included with standard C++ compilers
- Built-in functions for accessing specific GPU hardware capabilities

This section describes the built-in variables and functions accessible from the HIP kernel. It's intended for readers who are familiar with CUDA kernel syntax and want to understand how HIP is different.

The features are marked with one of the following keywords:

- Supported - HIP supports the feature with a CUDA-equivalent function
- Not supported - HIP does not support the feature
- Under development - the feature is under development but not yet available

3.4.1 Function-Type Qualifiers

3.4.1.1 __device__

The supported __device__ functions are:

- Executed on the device
- Called from the device only

The __device__ keyword can combine with the host keyword (see host).

3.4.1.2 __global__

The supported __global__ functions are:

- Executed on the device
- Called ("launched") from the host

HIP __global__ functions must have a void return type. See the *Kernel Launch example* for more information.

HIP lacks dynamic-parallelism support, so __global__ functions cannot be called from the device.

3.4.1.3 __host__

The supported __host__ functions are:

- Executed on the host
- Called from the host

__*host*__ can combine with __*device*__*,* in which case the function compiles for both the host and device. These functions cannot use the HIP grid coordinate functions. For example, "threadIdx_x". A possible workaround is to pass the necessary coordinate info as an argument to the function. _*host*__ cannot combine with __*global*__.

HIP parses the __*noinline*__ and __*forceinline*__ keywords and converts them to the appropriate Clang attributes.

3.4.1.4 Calling __global__ Functions

__*global*__ functions are often referred to as kernels, and calling one is termed launching the kernel. These functions require the caller to specify an "execution configuration" that includes the grid and block dimensions. The execution configuration can also include other information for the launch, such as the amount of additional shared memory to allocate and the stream where the kernel should execute. HIP introduces a standard C++ calling convention to pass the execution configuration to the kernel in addition to the Cuda <<< >>> syntax.

- In HIP, kernels launch with either the <<< >>> syntax or the "hipLaunchKernel" function.

- The first five parameters to hipLaunchKernel are the following:
  - symbol kernelName: the name of the kernel to launch. To support template kernels that contain "," use the HIP_KERNEL_NAME macro. The hipify tools insert this automatically.
  - dim3 gridDim: 3D-grid dimensions specifying the number of blocks to launch.
  - dim3 blockDim: 3D-block dimensions specifying the number of threads in each block.
  - size_t dynamicShared: the amount of additional shared memory to allocate when launching the kernel (see shared)
  - hipStream_t: stream where the kernel should execute. A value of 0 corresponds to the NULL stream (see Synchronization Functions).
- Kernel arguments must follow the five parameters

```
// Example pseudo code introducing hipLaunchKernel:
__global__ MyKernel(hipLaunchParm lp, float *A, float *B, float *C, size_t N)
{
...
}
MyKernel<<<dim3(gridDim), dim3(groupDim), 0, 0>>> (a,b,c,n);
// Alternatively, kernel can be launched by
// hipLaunchKernel(MyKernel, dim3(gridDim), dim3(groupDim), 0/*dynamicShared*/,
0/*stream), a, b, c, n);
```

The hipLaunchKernel macro always starts with the five parameters specified above, followed by the kernel arguments. HIPIFY tools optionally convert CUDA launch syntax to hipLaunchKernel, including conversion of optional arguments in <<< >>> to the five required hipLaunchKernel parameters. The dim3 constructor accepts zero to three arguments and will by default initialize unspecified dimensions to 1. See dim3. The kernel uses coordinate built-ins (thread*, block*, grid*) to determine the coordinate index and coordinate bounds of the work item that is currently executing. For more information, see Coordinate Built-Ins.

3.4.1.5 Kernel-Launch Example

```
// Example showing device function, __device__ __host__
// <- compile for both device and host
float PlusOne(float x)
{
    return x + 1.0;
}
__global__
void
MyKernel (const float *a, const float *b, float *c, unsigned N)
{
    unsigned gid = threadIdx.x; // <- coordinate index function
    if (gid < N) {
        c[gid] = a[gid] + PlusOne(b[gid]);
    }
}
void callMyKernel()
{
    float *a, *b, *c; // initialization not shown...
    unsigned N = 1000000;
    const unsigned blockSize = 256;
    MyKernel<<<dim3(gridDim), dim3(groupDim), 0, 0>>> (a,b,c,n);
    // Alternatively, kernel can be launched by
    // hipLaunchKernel(MyKernel, dim3(N/blockSize), dim3(blockSize), 0, 0,  a,b,c,N);
}
```

3.4.2 Variable-Type Qualifiers

3.4.2.1 __constant__

The __constant__ keyword is supported. The host writes constant memory before launching the kernel; from the GPU, this memory is read-only during kernel execution. The functions for accessing constant memory (hipGetSymbolAddress(), hipGetSymbolSize(), hipMemcpyToSymbol(), hipMemcpyToSymbolAsync(), hipMemcpyFromSymbol(), hipMemcpyFromSymbolAsync()) are available.

3.4.2.2 __shared__

The __*shared*__ keyword is supported.

extern __*shared*__ allows the host to dynamically allocate shared memory and is specified as a launch parameter.

Previously, it was essential to declare dynamic shared memory using the HIP_DYNAMIC_SHARED macro for accuracy, as using static shared memory in the same kernel could result in overlapping memory ranges and data-races.

Now, the HIP-Clang compiler provides support for extern shared declarations, and the HIP_DYNAMIC_SHARED option is no longer required.

3.4.2.3 __managed__

Managed memory, except the __*managed*__ keyword, is supported in HIP-combined host/device compilation.

Support of __*managed*__ keyword is under development.

3.4.2.4 __restrict__

The __restrict__ keyword tells the compiler that the associated memory pointer will not alias with any other pointer in the kernel or function. This feature can help the compiler generate better code. In most cases, all pointer arguments must use this keyword to realize the benefit.

3.4.3 Built-In Variables

3.4.3.1 Coordinate Built-Ins

These built-ins determine the coordinate of the active work item in the execution grid. They are defined in hip_runtime.h (rather than being implicitly defined by the compiler).

HIP Syntax	CUDA Syntax
threadIdx.x	threadIdx.x
threadIdx.y	threadIdx.y
threadIdx.z	threadIdx.z
blockIdx.x	blockIdx.x
blockIdx.y	blockIdx.y
blockIdx.z	blockIdx.z
blockDim.x	blockDim.x
blockDim.y	blockDim.y
blockDim.z	blockDim.z
gridDim.x	gridDim.x
gridDim.y	gridDim.y
gridDim.z	gridDim.z

3.4.3.2 warpSize

The warpSize variable is of type int and contains the warp size (in threads) for the target device. Note that all current Nvidia devices return 32 for this variable, and all current AMD devices return 64. Device code should use the warpSize built-in to develop portable wave-aware code.

3.4.4 Vector Types

Note that these types are defined in hip_runtime.h and are not automatically provided by the compiler.

3.4.4.1 Short Vector Types

Short vector types derive from the basic integer and floating-point types. They are structures defined in hip_vector_types.h. The first, second, third, and fourth components of the vector are accessible through the x, y, z, and w fields, respectively. All the short vector types support a constructor function of the form make_<type_name>(). For example, float4 make_float4(float x, float y, float z, float w) creates a vector of type float4 and value (x,y,z,w).

HIP supports the following short vector formats:

Signed Integers

- char1, char2, char3, char4
- short1, short2, short3, short4
- int1, int2, int3, int4
- long1, long2, long3, long4
- longlong1, longlong2, longlong3, longlong4

Unsigned Integers

- uchar1, uchar2, uchar3, uchar4
- ushort1, ushort2, ushort3, ushort4
- uint1, uint2, uint3, uint4
- ulong1, ulong2, ulong3, ulong4
- ulonglong1, ulonglong2, ulonglong3, ulonglong4

Floating Points

- float1, float2, float3, float4
- double1, double2, double3, double4

3.4.4.2 dim3

dim3 is a three-dimensional integer vector type commonly used to specify grid and group dimensions. Unspecified dimensions are initialized to 1.

```
typedef struct dim3 {
  uint32_t x;
  uint32_t y;
  uint32_t z;
  dim3(uint32_t _x=1, uint32_t _y=1, uint32_t _z=1) : x(_x), y(_y), z(_z) {};
};
```

3.4.5 Memory-Fence Instructions

HIP supports __threadfence() and __threadfence_block().

HIP provides a workaround for threadfence_system() under the HIP-Clang path. To enable the workaround, HIP should be built with environment variable HIP_COHERENT_HOST_ALLOC enabled.

Also, the kernels that use __threadfence_system() should be modified as follows:

- The kernel should only operate on finegrained system memory; which should be allocated with hipHostMalloc().
- Remove all memcpy for those allocated finegrained system memory regions.

3.4.6 Synchronization Functions

The __syncthreads() built-in function is supported in HIP. The __syncthreads_count(int), __syncthreads_and(int) and __syncthreads_or(int) functions are under development.

3.4.7 Math Functions

HIP-Clang supports a set of math operations callable from the device.

3.4.7.1 Single Precision Mathematical Functions

Following is the list of supported single-precision mathematical functions.

Function	Supported on Host	Supported on Device
float acosf (float x) Calculate the arc cosine of the input argument.	✓	✓
float acoshf (float x) Calculate the nonnegative arc hyperbolic cosine of the input argument.	✓	✓
float asinf (float x) Calculate the arc sine of the input argument.	✓	✓
float asinhf (float x) Calculate the arc hyperbolic sine of the input argument.	✓	✓
float atan2f (float y, float x) Calculate the arc tangent of the ratio of first and second input arguments.	✓	✓
float atanf (float x) Calculate the arc tangent of the input argument.	✓	✓
float atanhf (float x) Calculate the arc hyperbolic tangent of the input argument.	✓	✓
float cbrtf (float x) Calculate the cube root of the input argument.	✓	✓
float ceilf (float x) Calculate ceiling of the input argument.	✓	✓

Function	Supported on Host	Supported on Device
float copysignf (float x, float y) Create value with given magnitude, copying sign of second value.	✓	✓
float cosf (float x) Calculate the cosine of the input argument.	✓	✓
float coshf (float x) Calculate the hyperbolic cosine of the input argument.	✓	✓
float erfcf (float x) Calculate the complementary error function of the input argument.	✓	✓
float erff (float x) Calculate the error function of the input argument.	✓	✓
float exp10f (float x) Calculate the base 10 exponential of the input argument.	✓	✓
float exp2f (float x) Calculate the base 2 exponential of the input argument.	✓	✓
float expf (float x) Calculate the base e exponential of the input argument.	✓	✓
float expm1f (float x) Calculate the base e exponential of the input argument, minus 1.	✓	✓
float fabsf (float x) Calculate the absolute value of its argument.	✓	✓
float fdimf (float x, float y) Compute the positive difference between x and y.	✓	✓
float floorf (float x) Calculate the largest integer less than or equal to x.	✓	✓

Function	Supported on Host	Supported on Device
float fmaf (float x, float y, float z) Compute x × y + z as a single operation.	✓	✓
float fmaxf (float x, float y) Determine the maximum numeric value of the arguments.	✓	✓
float fminf (float x, float y) Determine the minimum numeric value of the arguments.	✓	✓
float fmodf (float x, float y) Calculate the floating-point remainder of x / y.	✓	✓
float frexpf (float x, int* nptr) Extract mantissa and exponent of a floating-point value.	✓	x
float hypotf (float x, float y) Calculate the square root of the sum of squares of two arguments.	✓	✓
int ilogbf (float x) Compute the unbiased integer exponent of the argument.	✓	✓
__RETURN_TYPE1 isfinite (float a) Determine whether the argument is finite.	✓	✓
__RETURN_TYPE1 isinf (float a) Determine whether the argument is infinite.	✓	✓
__RETURN_TYPE1 isnan (float a) Determine whether the argument is a NaN.	✓	✓
float ldexpf (float x, int exp) Calculate the value of x · 2exp.	✓	✓
float log10f (float x) Calculate the base 10 logarithm of the input argument.	✓	✓

Function	Supported on Host	Supported on Device
float log1pf (float x) Calculate the value of loge(1 + x).	✓	✓
float logbf (float x) Calculate the floating-point representation of the exponent of the input argument.	✓	✓
float log2f (float x) Calculate the base 2 logarithm of the input argument.	✓	✓
float logf (float x) Calculate the natural logarithm of the input argument.	✓	✓
float modff (float x, float* iptr) Break down the input argument into fractional and integral parts.	✓	x
float nanf (const char* tagp) Returns "Not a Number" value.	x	✓
float nearbyintf (float x) Round the input argument to the nearest integer.	✓	✓
float powf (float x, float y) Calculate the value of the first argument to the power of the second argument.	✓	✓
float remainderf (float x, float y) Compute single-precision floating-point remainder.	✓	✓
float remquof (float x, float y, int* quo) Compute single-precision floating-point remainder and part of quotient.	✓	x
float roundf (float x) Round to nearest integer value in floating-point.	✓	✓
float scalbnf (float x, int n) Scale floating-point input by an integer power of two.	✓	✓

Function	Supported on Host	Supported on Device
__RETURN_TYPE1 signbit (float a) Return the sign bit of the input.	✓	✓
void sincosf (float x, float* sptr, float* cptr) Calculate the sine and cosine of the first input argument.	✓	x
float sinf (float x) Calculate the sine of the input argument.	✓	✓
float sinhf (float x) Calculate the hyperbolic sine of the input argument.	✓	✓
float sqrtf (float x) Calculate the square root of the input argument.	✓	✓
float tanf (float x) Calculate the tangent of the input argument.	✓	✓
float tanhf (float x) Calculate the hyperbolic tangent of the input argument.	✓	✓
float truncf (float x) Truncate input argument to an integral part.	✓	✓
float tgammaf (float x) Calculate the gamma function of the input argument.	✓	✓
float erfcinvf (float y) Calculate the inverse complementary function of the input argument.	✓	✓
float erfcxf (float x) Calculate the scaled complementary error function of the input argument.	✓	✓
float erfinvf (float y) Calculate the inverse error function of the input argument.	✓	✓

Function	Supported on Host	Supported on Device
float fdividef (float x, float y) Divide two floating-point values.	✓	✓
float frexpf (float x, int *nptr) Extract mantissa and exponent of a floating-point value.	✓	✓
float j0f (float x) Calculate the value of the Bessel function of the first kind of order 0 for the input argument.	✓	✓
float j1f (float x) Calculate the value of the Bessel function of the first kind of order 1 for the input argument.	✓	✓
float jnf (int n, float x) Calculate the value of the Bessel function of the first kind of order n for the input argument.	✓	✓
float lgammaf (float x) Calculate the natural logarithm of the absolute value of the gamma function of the input argument.	✓	✓
long long int llrintf (float x) Round input to nearest integer value.	✓	✓
long long int llroundf (float x) Round to nearest integer value.	✓	✓
long int lrintf (float x) Round input to the nearest integer value.	✓	✓
long int lroundf (float x) Round to nearest integer value.	✓	✓
float modff (float x, float *iptr) Break down the input argument into fractional and integral parts.	✓	✓
float nextafterf (float x, float y)	✓	✓

Function	Supported on Host	Supported on Device
Returns next representable single-precision floating-point value after an argument.		
float norm3df (float a, float b, float c) Calculate the square root of the sum of squares of three coordinates of the argument.	✓	✓
float norm4df (float a, float b, float c, float d) Calculate the square root of the sum of squares of four coordinates of the argument.	✓	✓
float normcdff (float y) Calculate the standard normal cumulative distribution function.	✓	✓
float normcdfinvf (float y) Calculate the inverse of the standard normal cumulative distribution function.	✓	✓
float normf (int dim, const float *a) Calculate the square root of the sum of squares of any number of coordinates.	✓	✓
float rcbrtf (float x) Calculate the reciprocal cube root function.	✓	✓
float remquof (float x, float y, int *quo) Compute single-precision floating-point remainder and part of quotient.	✓	✓
float rhypotf (float x, float y) Calculate one over the square root of the sum of squares of two arguments.	✓	✓
float rintf (float x) Round input to nearest integer value in floating-point.	✓	✓
float rnorm3df (float a, float b, float c) Calculate one over the square root of the sum of squares of three coordinates of the argument.	✓	✓
float rnorm4df (float a, float b, float c, float d)	✓	✓

Function	Supported on Host	Supported on Device
Calculate one over the square root of the sum of squares of four coordinates of the argument.		
float rnormf (int dim, const float *a) Calculate the reciprocal of square root of the sum of squares of any number of coordinates.	✓	✓
float scalblnf (float x, long int n) Scale floating-point input by an integer power of two.	✓	✓
void sincosf (float x, float *sptr, float *cptr) Calculate the sine and cosine of the first input argument.	✓	✓
void sincospif (float x, float *sptr, float *cptr) Calculate the sine and cosine of the first input argument multiplied by PI.	✓	✓
float y0f (float x) Calculate the value of the Bessel function of the second kind of order 0 for the input argument.	✓	✓
float y1f (float x) Calculate the value of the Bessel function of the second kind of order 1 for the input argument.	✓	✓
float ynf (int n, float x) Calculate the value of the Bessel function of the second kind of order n for the input argument.	✓	✓

3.4.7.2 Double Precision Mathematical Functions

The following table consists of supported double-precision mathematical functions.

Function	Supported on Host	Supported on Device
double acos (double x) Calculate the arc cosine of the input argument.	✓	✓

Function	Supported on Host	Supported on Device
double acosh (double x) Calculate the nonnegative arc hyperbolic cosine of the input argument.	✓	✓
double asin (double x) Calculate the arc sine of the input argument.	✓	✓
double asinh (double x) Calculate the arc hyperbolic sine of the input argument.	✓	✓
double atan (double x) Calculate the arc tangent of the input argument.	✓	✓
double atan2 (double y, double x) Calculate the arc tangent of the ratio of first and second input arguments.	✓	✓
double atanh (double x) Calculate the arc hyperbolic tangent of the input argument.	✓	✓
double cbrt (double x) Calculate the cube root of the input argument.	✓	✓
double ceil (double x) Calculate ceiling of the input argument.	✓	✓
double copysign (double x, double y) Create value with given magnitude, copying sign of second value.	✓	✓
double cos (double x) Calculate the cosine of the input argument.	✓	✓
double cosh (double x) Calculate the hyperbolic cosine of the input argument.	✓	✓
double erf (double x) Calculate the error function of the input argument.	✓	✓

Function	Supported on Host	Supported on Device
double erfc (double x) Calculate the complementary error function of the input argument.	✓	✓
double exp (double x) Calculate the base e exponential of the input argument.	✓	✓
double exp10 (double x) Calculate the base 10 exponential of the input argument.	✓	✓
double exp2 (double x) Calculate the base 2 exponential of the input argument.	✓	✓
double expm1 (double x) Calculate the base e exponential of the input argument, minus 1.	✓	✓
double fabs (double x) Calculate the absolute value of the input argument.	✓	✓
double fdim (double x, double y) Compute the positive difference between x and y.	✓	✓
double floor (double x) Calculate the largest integer less than or equal to x.	✓	✓
double fma (double x, double y, double z) Compute x × y + z as a single operation.	✓	✓
double fmax (double , double) Determine the maximum numeric value of the arguments.	✓	✓
double fmin (double x, double y) Determine the minimum numeric value of the arguments.	✓	✓
double fmod (double x, double y) Calculate the floating-point remainder of x / y.	✓	✓

Function	Supported on Host	Supported on Device
double frexp (double x, int* nptr) Extract mantissa and exponent of a floating-point value.	✓	**x**
double hypot (double x, double y) Calculate the square root of the sum of squares of two arguments.	✓	✓
int ilogb (double x) Compute the unbiased integer exponent of the argument.	✓	✓
__RETURN_TYPE1 isfinite (double a) Determine whether an argument is finite.	✓	✓
__RETURN_TYPE1 isinf (double a) Determine whether an argument is infinite.	✓	✓
__RETURN_TYPE1 isnan (double a) Determine whether an argument is a NaN.	✓	✓
double ldexp (double x, int exp) Calculate the value of x · 2exp.	✓	✓
double log (double x) Calculate the base e logarithm of the input argument.	✓	✓
double log10 (double x) Calculate the base 10 logarithm of the input argument.	✓	✓
double log1p (double x) Calculate the value of loge(1 + x).	✓	✓
double log2 (double x) Calculate the base 2 logarithm of the input argument.	✓	✓
double logb (double x) Calculate the floating-point representation of the exponent of the input argument.	✓	✓

Function	Supported on Host	Supported on Device
double modf (double x, double* iptr) Break down the input argument into fractional and integral parts.	✓	x
double nan (const char* tagp) Returns "Not a Number" value.	x	✓
double nearbyint (double x) Round the input argument to the nearest integer.	✓	✓
double pow (double x, double y) Calculate the value of the first argument to the power of the second argument.	✓	✓
double remainder (double x, double y) Compute double-precision floating-point remainder.	✓	✓
double remquo (double x, double y, int* quo) Compute double-precision floating-point remainder and part of quotient.	✓	x
double round (double x) Round to nearest integer value in floating-point.	✓	✓
double scalbn (double x, int n) Scale floating-point input by an integer power of two.	✓	✓
__RETURN_TYPE1 signbit (double a) Return the sign bit of the input.	✓	✓
double sin (double x) Calculate the sine of the input argument.	✓	✓
void sincos (double x, double* sptr, double* cptr) Calculate the sine and cosine of the first input argument.	✓	x
double sinh (double x) Calculate the hyperbolic sine of the input argument.	✓	✓

Function	Supported on Host	Supported on Device
double sqrt (double x) Calculate the square root of the input argument.	✓	✓
double tan (double x) Calculate the tangent of the input argument.	✓	✓
double tanh (double x) Calculate the hyperbolic tangent of the input argument.	✓	✓
double tgamma (double x) Calculate the gamma function of the input argument.	✓	✓
double trunc (double x) Truncate input argument to an integral part.	✓	✓
double erfcinv (double y) Calculate the inverse complementary function of the input argument.	✓	✓
double erfcx (double x) Calculate the scaled complementary error function of the input argument.	✓	✓
double erfinv (double y) Calculate the inverse error function of the input argument.	✓	✓
double frexp (float x, int *nptr) Extract mantissa and exponent of a floating-point value.	✓	✓
double j0 (double x) Calculate the value of the Bessel function of the first kind of order 0 for the input argument.	✓	✓
double j1 (double x) Calculate the value of the Bessel function of the first kind of order 1 for the input argument.	✓	✓
double jn (int n, double x)	✓	✓

Function	Supported on Host	Supported on Device
Calculate the value of the Bessel function of the first kind of order n for the input argument.		
double lgamma (double x) Calculate the natural logarithm of the absolute value of the gamma function of the input argument.	✓	✓
long long int llrint (double x) Round input to a nearest integer value.	✓	✓
long long int llround (double x) Round to nearest integer value.	✓	✓
long int lrint (double x) Round input to a nearest integer value.	✓	✓
long int lround (double x) Round to nearest integer value.	✓	✓
double modf (double x, double *iptr) Break down the input argument into fractional and integral parts.	✓	✓
double nextafter (double x, double y) Returns next representable single-precision floating-point value after an argument.	✓	✓
double norm3d (double a, double b, double c) Calculate the square root of the sum of squares of three coordinates of the argument.	✓	✓
float norm4d (double a, double b, double c, double d) Calculate the square root of the sum of squares of four coordinates of the argument.	✓	✓
double normcdf (double y) Calculate the standard normal cumulative distribution function.	✓	✓
double normcdfinv (double y)	✓	✓

Function	Supported on Host	Supported on Device
Calculate the inverse of the standard normal cumulative distribution function.		
double rcbrt (double x) Calculate the reciprocal cube root function.	✓	✓
double remquo (double x, double y, int *quo) Compute single-precision floating-point remainder and part of quotient.	✓	✓
double rhypot (double x, double y) Calculate one over the square root of the sum of squares of two arguments.	✓	✓
double rint (double x) Round input to the nearest integer value in floating-point.	✓	✓
double rnorm3d (double a, double b, double c) Calculate one over the square root of the sum of squares of three coordinates of the argument.	✓	✓
double rnorm4d (double a, double b, double c, double d) Calculate one over the square root of the sum of squares of four coordinates of the argument.	✓	✓
double rnorm (int dim, const double *a) Calculate the reciprocal of the square root of the sum of squares of any number of coordinates.	✓	✓
double scalbln (double x, long int n) Scale floating-point input by an integer power of two.	✓	✓
void sincos (double x, double *sptr, double *cptr) Calculate the sine and cosine of the first input argument.	✓	✓
void sincospi (double x, double *sptr, double *cptr) Calculate the sine and cosine of the first input argument multiplied by PI.	✓	✓

Function	Supported on Host	Supported on Device
double y0f (double x) Calculate the value of the Bessel function of the second kind of order 0 for the input argument.	✓	✓
double y1 (double x) Calculate the value of the Bessel function of the second kind of order 1 for the input argument.	✓	✓
double yn (int n, double x) Calculate the value of the Bessel function of the second kind of order n for the input argument.	✓	✓

NOTE: [1] __RETURN_TYPE is dependent on the compiler. It is usually 'int' for C compilers and 'bool' for C++ compilers.

3.4.7.3 Integer Intrinsics

The following table lists supported integer intrinsics. Note, intrinsics are supported on devices only.

Function
unsigned int __brev (unsigned int x) Reverse the bit order of a 32-bit unsigned integer.
unsigned long long int __brevll (unsigned long long int x) Reverse the bit order of a 64-bit unsigned integer.
int __clz (int x) Return the number of consecutive high-order zero bits in a 32-bit integer.
unsigned int __clz(unsigned int x) Return the number of consecutive high-order zero bits in 32-bit unsigned integer.
int __clzll (long long int x) Count the number of consecutive high-order zero bits in a 64-bit integer.
unsigned int __clzll(long long int x)

Function
Return the number of consecutive high-order zero bits in 64-bit signed integer.
unsigned int __ffs(unsigned int x) Find the position of least significant bit set to 1 in a 32-bit unsigned integer.1
unsigned int __ffs(int x) Find the position of least significant bit set to 1 in a 32-bit signed integer.
unsigned int __ffsll(unsigned long long int x) Find the position of least significant bit set to 1 in a 64-bit unsigned integer.1
unsigned int __ffsll(long long int x) Find the position of least significant bit set to 1 in a 64 bit signed integer.
unsigned int __popc (unsigned int x) Count the number of bits that are set to 1 in a 32-bit integer.
int __popcll (unsigned long long int x) Count the number of bits that are set to 1 in a 64-bit integer.
int __mul24 (int x, int y) Multiply two 24-bit integers.
unsigned int __umul24 (unsigned int x, unsigned int y) Multiply two 24-bit unsigned integers.

NOTE: The HIP-Clang implementation of __ffs() and __ffsll() contains code to add a constant +1 to produce the *ffs* result format. For the cases where this overhead is not acceptable and the programmer is willing to specialize for the platform, HIP-Clang provides __lastbit_u32_u32(unsigned int input) and __lastbit_u32_u64(unsigned long long int input).

3.4.7.4 Floating-point Intrinsics

The following table provides a list of supported floating-point intrinsics. Note, intrinsics are supported on devices only.

Function
float __cosf (float x) Calculate the fast approximate cosine of the input argument.
float __expf (float x) Calculate the fast approximate base e exponential of the input argument.
float __frsqrt_rn (float x) Compute 1 / √x in round-to-nearest-even mode.
float __fsqrt_rd (float x) Compute √x in round-down mode.
float __fsqrt_rn (float x) Compute √x in round-to-nearest-even mode.
float __fsqrt_ru (float x) Compute √x in round-up mode.
float __fsqrt_rz (float x) Compute √x in round-towards-zero mode.
float __log10f (float x) Calculate the fast approximate base 10 logarithm of the input argument.
float __log2f (float x) Calculate the fast approximate base 2 logarithm of the input argument.
float __logf (float x) Calculate the fast approximate base e logarithm of the input argument.
float __powf (float x, float y) Calculate the fast approximate of xy.
float __sinf (float x) Calculate the fast approximate sine of the input argument.
float __tanf (float x) Calculate the fast approximate tangent of the input argument.

Function
double __dsqrt_rd (double x) Compute √x in round-down mode.
double __dsqrt_rn (double x) Compute √x in round-to-nearest-even mode.
double __dsqrt_ru (double x) Compute √x in round-up mode.
double __dsqrt_rz (double x) Compute √x in round-towards-zero mode.

3.4.7.5 Texture Functions

The supported Texture functions are listed in the following header files:

- *"texture_functions.h"*

For more information, see

https://github.com/ROCm-Developer-Tools/HIP/blob/main/include/hip/hcc_detail/texture_functions.h

- *"texture_indirect_functions.h"*

For more information, see

https://github.com/ROCm-Developer-Tools/HIP/blob/main/include/hip/hcc_detail/texture_indirect_functions.h

3.4.7.6 Surface Functions

Surface functions are not supported.

3.4.7.7 Timer Functions

HIP provides the following built-in functions for reading a high-resolution timer from the device.

```
clock_t clock()
long long int clock64()
```

Returns the value of a counter that is incremented every clock cycle on devices. The difference in values returned provides the cycles used.

3.4.7.8 Atomic Functions

Atomic functions execute as read-modify-write operations residing in global or shared memory. No other device or thread can observe or modify the memory location during an atomic operation. If multiple instructions from different devices or threads target the same memory location, the instructions are serialized in an undefined order.

HIP adds new APIs with _system as suffix to support system scope atomic operations. For example, atomicAnd atomic is dedicated to the GPU device, atomicAnd_system will allow developers to extend the atomic operation to system scope, from the GPU device to other CPUs and GPU devices in the system.

HIP supports the following atomic operations:

Function	Supported in HIP	Supported in CUDA
int atomicAdd(int* address, int val)	✓	✓
int atomicAdd_system(int* address, int val)	✓	✓
unsigned int atomicAdd(unsigned int* address,unsigned int val)	✓	✓
unsigned int atomicAdd_system(unsigned int* address, unsigned int val)	✓	✓
unsigned long long atomicAdd(unsigned long long* address,unsigned long long val)	✓	✓
unsigned long long atomicAdd_system(unsigned long long* address, unsigned long long val)	✓	✓
float atomicAdd(float* address, float val)	✓	✓
float atomicAdd_system(float* address, float val)	✓	✓
double atomicAdd(double* address, double val)	✓	✓
double atomicAdd_system(double* address, double val)	✓	✓
int atomicSub(int* address, int val)	✓	✓
int atomicSub_system(int* address, int val)	✓	✓
unsigned int atomicSub(unsigned int* address,unsigned int val)	✓	✓
unsigned int atomicSub_system(unsigned int* address, unsigned int val)	✓	✓
int atomicExch(int* address, int val)	✓	✓
int atomicExch_system(int* address, int val)	✓	✓
unsigned int atomicExch(unsigned int* address,unsigned int val)	✓	✓
unsigned int atomicExch_system(unsigned int* address, unsigned int val)	✓	✓
unsigned long long atomicExch(unsigned long long int* address,unsigned long long int val)	✓	✓

unsigned long long atomicExch_system(unsigned long long* address, unsigned long long val)	✓	✓
unsigned long long atomicExch_system(unsigned long long* address, unsigned long long val)	✓	✓
float atomicExch(float* address, float val)	✓	✓
int atomicMin(int* address, int val)	✓	✓
int atomicMin_system(int* address, int val)	✓	✓
unsigned int atomicMin(unsigned int* address,unsigned int val)	✓	✓
unsigned int atomicMin_system(unsigned int* address, unsigned int val)	✓	✓
unsigned long long atomicMin(unsigned long long* address,unsigned long long val)	✓	✓
int atomicMax(int* address, int val)	✓	✓
int atomicMax_system(int* address, int val)	✓	✓
unsigned int atomicMax(unsigned int* address,unsigned int val)	✓	✓
unsigned int atomicMax_system(unsigned int* address, unsigned int val)	✓	✓
unsigned long long atomicMax(unsigned long long* address,unsigned long long val)	✓	✓
unsigned int atomicInc(unsigned int* address)	✗	✓
unsigned int atomicDec(unsigned int* address)	✗	✓
int atomicCAS(int* address, int compare, int val)	✓	✓
int atomicCAS_system(int* address, int compare, int val)	✓	✓
unsigned int atomicCAS(unsigned int* address,unsigned int compare,unsigned int val)	✓	✓
unsigned int atomicCAS_system(unsigned int* address, unsigned int compare, unsigned int val)	✓	✓
unsigned long long atomicCAS(unsigned long long* address,unsigned long long compare,unsigned long long val)	✓	✓
unsigned long long atomicCAS_system(unsigned long long* address, unsigned long long compare, unsigned long long val)	✓	✓
int atomicAnd(int* address, int val)	✓	✓
int atomicAnd_system(int* address, int val)	✓	✓
unsigned int atomicAnd(unsigned int* address,unsigned int val)	✓	✓
unsigned int atomicAnd_system(unsigned int* address, unsigned int val)	✓	✓
unsigned long long atomicAnd(unsigned long long* address,unsigned long long val)	✓	✓
unsigned long long atomicAnd_system(unsigned long long* address, unsigned long long val)	✓	✓
int atomicOr(int* address, int val)	✓	✓
int atomicOr_system(int* address, int val)	✓	✓

unsigned int atomicOr(unsigned int* address,unsigned int val)	✓	✓
unsigned int atomicOr_system(unsigned int* address, unsigned int val)	✓	✓
unsigned int atomicOr_system(unsigned int* address, unsigned int val)	✓	✓
unsigned long long atomicOr(unsigned long long int* address,unsigned long long val)	✓	✓
unsigned long long atomicOr_system(unsigned long long* address, unsigned long long val)	✓	✓
int atomicXor(int* address, int val)	✓	✓
int atomicXor_system(int* address, int val)	✓	✓
unsigned int atomicXor(unsigned int* address,unsigned int val)	✓	✓
unsigned int atomicXor_system(unsigned int* address, unsigned int val)	✓	✓
unsigned long long atomicXor(unsigned long long* address,unsigned long long val))	✓	✓
unsigned long long atomicXor_system(unsigned long long* address, unsigned long long val)	✓	✓

Caveats and Features Under-Development

HIP enables atomic operations on 32-bit integers. Additionally, it supports an atomic float add. AMD hardware, however, implements the float add using a CAS loop, so this function may not perform efficiently.

3.4.7.9 Warp Cross-Lane Functions

Warp cross-lane functions operate across all lanes in a warp. The hardware guarantees that all warp lanes will execute in lockstep, so additional synchronization is unnecessary and the instructions use no shared memory.

Note that Nvidia and AMD devices have different warp sizes, so portable code should use the warpSize built-ins to query the warp size. Hipified code from the CUDA path requires careful review to ensure it doesn't assume a waveSize of 32. "Wave-aware" code that assumes a waveSize of 32 will run on a wave-64 machine, but it will utilize only half of the machine resources.

WarpSize built-ins should only be used in device functions and its value depends on GPU arch. Host functions should use *hipGetDeviceProperties* to get the default warp size of a GPU device:

```
cudaDeviceProp props;
cudaGetDeviceProperties(&props, deviceID);
    int w = props.warpSize;
    // implement portable algorithm based on w (rather than assume 32 or 64)
```

Note, assembly kernels may be built for warp size, which is different than the default warp size.

3.4.7.10 Warp Vote and Ballot Functions

```
int __all(int predicate)
int __any(int predicate)
uint64_t __ballot(int predicate)
```

Threads in a warp are referred to as lanes and are numbered from 0 to warpSize -- 1. For these functions, each warp lane contributes 1 -- the bit value (the predicate), which is efficiently broadcast to all lanes in the warp. The 32-bit int predicate from each lane reduces to a 1-bit value: 0 (predicate = 0) or 1 (predicate != 0). __any and __all provide a summary view of the predicates that the other warp lanes contribute:

- __any() returns 1 if any warp lane contributes a nonzero predicate, or 0 otherwise
- __all() returns 1 if all other warp lanes contribute nonzero predicates, or 0 otherwise

Applications can test whether the target platform supports the any/all instruction using the hasWarpVote device property or the HIP_ARCH_HAS_WARP_VOTE compiler define.

__ballot provides a bit mask containing the 1-bit predicate value from each lane. The nth bit of the result contains the 1 bit contributed by the nth warp lane. Note that HIP's __ballot function supports a 64-bit return value (compared with 32 bits). Code ported from CUDA should support the larger warp sizes that the HIP version of this instruction supports. Applications can test whether the target platform supports the ballot instruction using the hasWarpBallot device property or the HIP_ARCH_HAS_WARP_BALLOT compiler define.

3.4.7.11 Warp Shuffle Functions

Half-float shuffles are not supported. The default width is warpSize---see Warp Cross-Lane Functions. Applications should not assume the warpSize is 32 or 64.

```
int   __shfl      (int var,   int srcLane, int width=warpSize);
float __shfl      (float var, int srcLane, int width=warpSize);
int   __shfl_up   (int var,   unsigned int delta, int width=warpSize);
float __shfl_up   (float var, unsigned int delta, int width=warpSize);
int   __shfl_down (int var,   unsigned int delta, int width=warpSize);
float __shfl_down (float var, unsigned int delta, int width=warpSize) ;
int   __shfl_xor  (int var,   int laneMask, int width=warpSize)
float __shfl_xor  (float var, int laneMask, int width=warpSize);
```

3.4.7.12 Cooperative Groups Functions

Cooperative Groups is a mechanism for forming and communicating between groups of threads at a granularity different than the block. This feature was introduced in CUDA 9. HIP supports the following kernel language cooperative groups types or functions.

Function	HIP	CUDA
void thread_group.sync() ;	✓	✓
unsigned thread_group.size();	✓	✓
unsigned thread_group.thread_rank() ;	✓	✓
bool thread_group.is_valid();	✓	✓
grid_group this_grid();	✓	✓
void grid_group.sync() ;	✓	✓
unsigned grid_group.size() ;	✓	✓
unsigned grid_group.thread_rank() ;	✓	✓
bool grid_group.is_valid();	✓	✓
multi_grid_group this_multi_grid() ;	✓	✓
void multi_grid_group.sync();	✓	✓
unsigned multi_grid_group.size() ;	✓	✓
unsigned multi_grid_group.thread_rank() ;	✓	✓
bool multi_grid_group.is_valid() ;	✓	✓
unsigned multi_grid_group.num_grids() ;	✓	✓
unsigned multi_grid_group.grid_rank();	✓	✓
thread_block this_thread_block() ;	✓	✓
multi_grid_group this_multi_grid() ;	✓	✓
void multi_grid_group.sync();	✓	✓
void thread_block.sync() ;	✓	✓
unsigned thread_block.size() ;	✓	✓
unsigned thread_block.thread_rank() ;	✓	✓
bool thread_block.is_valid() ;	✓	✓
dim3 thread_block.group_index() ;	✓	✓
dim3 thread_block.thread_index()	✓	✓

3.4.7.13 Warp Matrix Functions

Warp matrix functions allow a warp to cooperatively operate on small matrices whose elements are spread over the lanes in an unspecified manner. This feature was introduced in CUDA 9.

HIP does not support any of the kernel language warp matrix types or functions.

Function	Supported in HIP	Supported in CUDA
void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned lda)		✓
void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned lda, layout_t layout)		✓
void store_matrix_sync(T* mptr, fragment<...> &a, unsigned lda, layout_t layout)		✓
void fill_fragment(fragment<...> &a, const T &value)		✓
void mma_sync(fragment<...> &d, const fragment<...> &a, const fragment<...> &b, const fragment<...> &c , bool sat)		✓

3.4.7.14 Independent Thread Scheduling

The hardware support for independent thread scheduling introduced in certain architectures supporting CUDA allows threads to progress independently of each other and enables intra-warp synchronizations that were previously not allowed.

HIP does not support this type of thread scheduling.

3.4.7.15 Profiler Counter Function

The Cuda __prof_trigger() instruction is not supported.

3.4.7.16 Assert

The assert function is under development. HIP does support an "abort" call which will terminate the process execution from inside the kernel.

3.4.7.17 Printf

The printf function is supported.

3.4.8 Device-Side Dynamic Global Memory Allocation

Device-side dynamic global memory allocation is under development.

3.4.9 __launch_bounds__

GPU multiprocessors have a fixed pool of resources (primarily registers and shared memory) which are shared by the actively running warps. Using more resources can increase IPC of the kernel but reduces the resources available for other warps and limits the number of warps that can be simultaneously running. Thus, GPUs have a complex relationship between resource usage and performance.

__launch_bounds__ allows the application to provide usage hints that influence the resources (primarily registers) used by the generated code. It is a function attribute that must be attached to a __global__ function:

```
__global__ void `-__launch_bounds__`-(MAX_THREADS_PER_BLOCK, MIN_WARPS_PER_EU)
MyKernel(...) ...
MyKernel(...)
```

launch_bounds supports two parameters:

- MAX_THREADS_PER_BLOCK - The programmers guarantees that the kernel will be launched with threads less than MAX_THREADS_PER_BLOCK. (On NVCC this maps to the .maxntid PTX directive). If no launch_bounds is specified, MAX_THREADS_PER_BLOCK is the maximum block size supported by the device (typically 1024 or larger). Specifying MAX_THREADS_PER_BLOCK less than the maximum effectively allows the compiler to use more resources than a default unconstrained compilation that supports all possible block sizes at launch time. The threads-per-block is the product of (hipBlockDim_x * hipBlockDim_y * hipBlockDim_z).

- MIN_WARPS_PER_EU - directs the compiler to minimize resource usage so that the requested number of warps can be simultaneously active on a multi-processor. Since active warps compete for the same fixed pool of resources, the compiler must reduce resources required by each warp(primarily registers). MIN_WARPS_PER_EU is optional and defaults to 1 if not specified. Specifying a MIN_WARPS_PER_EU greater than the default 1 effectively constrains the compiler's resource usage.

3.4.9.1 Compiler Impact

The compiler uses these parameters as follows:

- The compiler uses the hints only to manage register usage and does not automatically reduce shared memory or other resources.
- Compilation fails if the compiler cannot generate a kernel that meets the requirements of the specified launch bounds.
- From MAX_THREADS_PER_BLOCK, the compiler derives the maximum number of warps/block that can be used at launch time. Values of MAX_THREADS_PER_BLOCK less than the default allows the compiler to use a larger pool of registers: each warp uses registers, and this hint contains the launch to a warps/block size that is less than maximum.

- From MIN_WARPS_PER_EU, the compiler derives a maximum number of registers that can be used by the kernel (to meet the required #simultaneous active blocks). If MIN_WARPS_PER_EU is 1, then the kernel can use all registers supported by the multiprocessor.
- The compiler ensures that the registers used in the kernel is less than both allowed maximums, typically by spilling registers (to shared or global memory), or by using more instructions.
- The compiler may use heuristics to increase register usage or may simply be able to avoid spilling. The MAX_THREADS_PER_BLOCK is particularly useful in this case, since it allows the compiler to use more registers and avoid situations where the compiler constrains the register usage (potentially spilling) to meet the requirements of a large block size that is never used at launch time.

3.4.9.2 CU and EU Definitions

A compute unit (CU) is responsible for executing the waves of a workgroup. It is composed of one or more execution units (EU) that are responsible for executing waves. An EU can have enough resources to maintain the state of more than one executing wave. This allows an EU to hide latency by switching between waves in a similar way to symmetric multithreading on a CPU. To allow the state for multiple waves to fit on an EU, the resources used by a single wave have to be limited. Limiting such resources can allow greater latency hiding but it can result in having to spill some register state to memory. This attribute allows an advanced developer to tune the number of waves that are capable of fitting within the resources of an EU. It can be used to ensure at least a certain number will fit to help hide latency and can also be used to ensure no more than a certain number will fit to limit cache thrashing.

3.4.9.3 Porting from CUDA __launch_bounds

CUDA defines a *__launch_bounds,* which is also designed to control occupancy:

```
__launch_bounds(MAX_THREADS_PER_BLOCK, MIN_BLOCKS_PER_MULTIPROCESSOR)
```

The second parameter *__launch_bounds* parameters must be converted to the format used *__hip_launch_bounds*, which uses warps and execution-units rather than blocks and multi-processors (this conversion is performed automatically by hipify tools).

```
MIN_WARPS_PER_EXECUTION_UNIT = (MIN_BLOCKS_PER_MULTIPROCESSOR * MAX_THREADS_PER_BLOCK) /
32
```

The key differences in the interface are:

- Warps (rather than blocks): The developer is trying to tell the compiler to control resource utilization to guarantee some amount of active Warps/EU for latency hiding. Specifying active warps in terms of blocks appears to hide the micro-architectural details of the warp size, however, makes the interface more confusing since the developer ultimately needs to compute the number of warps to obtain the desired level of control.
- Execution Units (rather than multiProcessor): The use of execution units rather than multiprocessors provides support for architectures with multiple execution units/multi-processor. For example, the AMD GCN architecture has 4 execution units per multiProcessor. The hipDeviceProps has a field executionUnitsPerMultiprocessor. Platform-specific coding techniques such as #ifdef can be used to specify different launch_bounds for NVCC and HIP-Clang platforms if desired.

3.4.9.4 Maxregcount

Unlike nvcc, HIP-Clang does not support the "--maxregcount" option. Instead, users are encouraged to use the hip_launch_bounds directive since the parameters are more intuitive and portable than micro-architecture details like registers, and also the directive allows per-kernel control rather than an entire file. hip_launch_bounds works on both HIP-Clang and nvcc targets.

3.4.10 Register Keyword

The register keyword is deprecated in C++ and is silently ignored by both nvcc and HIP-Clang. You can pass the option `-Wdeprecated-register` to the compiler warning message.

3.4.11 Pragma Unroll

Unroll with a bound that is known at compile-time is supported. For example:

```
#pragma unroll 16 /* hint to compiler to unroll next loop by 16 */
for (int i=0; i<16; i++) ...
#pragma unroll 1  /* tell compiler to never unroll the loop */
for (int i=0; i<16; i++) ...
#pragma unroll /* hint to compiler to completely unroll next loop. */
for (int i=0; i<16; i++) ...
```

3.4.12 In-Line Assembly

GCN ISA In-line assembly is supported. For example:

```
asm volatile ("v_mac_f32_e32 %0, %2, %3" : "=v" (out[i]) : "0"(out[i]), "v" (a), "v"
(in[i]));
```

The HIP compiler inserts the GCN into the kernel using asm() Assembler statement. volatile keyword is used so that the optimizers must not change the number of volatile operations or change their order of execution relative to other volatile operations. v_mac_f32_e32 is the GCN instruction. For more information, refer to the AMD GCN3 ISA architecture manual Index for the respective operand in the ordered fashion is provided by % followed by a position in the list of operands "v" is the constraint code (for target-specific AMDGPU) for 32-bit VGPR register. For more information, refer to the Supported Constraint Code List for AMDGPU. Output Constraints are specified by an "=" prefix as shown above ("=v"). This indicates that assembly will write to this operand, and the operand will then be made available as a return value of the asm expression. Input constraints do not have a prefix - just the constraint code. The constraint string of "0" says to use the assigned register for output as an input as well (it being the 0'th constraint).

3.4.13 C++ Support

The following C++ features are not supported:

- Run-time-type information (RTTI)
- Virtual functions
- Try/catch

3.4.14 Kernel Compilation

hipcc now supports compiling C++/HIP kernels to binary code objects.

The file format for binary is \`.co\` which means Code Object. The following command builds the code object using \`hipcc\`.

```
`hipcc --genco --offload-arch=[TARGET GPU] [INPUT FILE] -o [OUTPUT FILE]`
[TARGET GPU] = GPU architecture
[INPUT FILE] = Name of the file containing kernels
[OUTPUT FILE] = Name of the generated code object file
```

NOTE: When using binary code objects is that the number of arguments to the kernel is different on HIP-Clang and NVCC path. Refer to the sample in samples/0_Intro/module_api for differences in the arguments to be passed to the kernel.

3.4.15 gfx-arch-specific-kernel

Clang defined '__gfx*__' macros can be used to execute gfx arch-specific codes inside the kernel. Refer to the *sample 14_gpu_arch in samples/2_Cookbook.*

3.5 ROCm Code Object Tooling

ROCm compiler-generated code objects (executables, object files, and shared object libraries) can be examined and extracted with the tools listed in this section.

3.5.1 Uniform Resource Identifier Syntax

ROCm code objects can be listed or accessed using the following Uniform Resource Identifier (URI) syntax:

```
code_object_uri ::== file_uri | memory_uri
          file_uri        ::== file:// extract_file [ range_specifier ]
          memory_uri      ::== memory:// process_id range_specifier
          range_specifier ::== [ # | ? ] offset= number & size= number
          extract_file    ::== URI_ENCODED_OS_FILE_PATH
          process_id      ::== DECIMAL_NUMBER
          number          ::== HEX_NUMBER | DECIMAL_NUMBER | OCTAL_NUMBER
```

3.5.1.1 Examples

- file://dir1/dir2/hello_world#offset=133&size=14472
- memory://1234#offset=0x20000&size=3000

3.5.2 List Available ROCm Code Objects

Use this tool to list available ROCm code objects in a given executable. Code objects are listed using the URI syntax.

Usage: *roc-obj-ls [-v|h] <executable-name> ...*

Options:

- -v Verbose output (includes Entry ID)
- -h Show this help message

Example:

roc-obj-ls ./hipLaunchParm

Output:

file://./hipLaunchParm#offset=24576&size=0

file://./hipLaunchParm#offset=24576&size=46816

file://./hipLaunchParm#offset=73728&size=46816

file://./hipLaunchParm#offset=122880&size=46816

3.5.3 ROCm Code Objects Extraction

You can extract the available ROCm code objects from a specified URI.

Usage: *rocm-obj-extract [-o|v|h] URI...*

NOTE

- URIs can be read from STDIN, one per line.
- The specified URIs extracts code objects into files named: *<executable_name>-[pid]-offset-size.co*

Options:

- *-o* Path for output. If "-" specified, the code object is printed to STDOUT.
- *-v* Verbose output (includes Entry ID).
- *-h* Shows the Help message

3.5.4 ROCm Code Object Tooling Examples

3.5.4.1 Dump all code objects to current directory

```
roc-obj-ls <exe> | roc-obj-extract
```

3.5.4.2 Dump the ISA for a specific target: e.g gfx906

```
roc-obj-ls -v <exe> | grep "gfx908" | awk '{print $2}' | roc-obj-extract -o - | llvm-
readelf -h - | grep Flags
```

3.5.4.3 Check the e_flags for the gfx908 code object

```
roc-obj-ls -v <exe> | grep "gfx908" | awk '{print $2}' | roc-obj-extract -o - | llvm-
readelf -h - | grep Flags
```

3.5.4.4 Disassemble the fourth code object

```
roc-obj-ls <exe> | sed -n 4p | roc-obj-extract -o - | llvm-objdump -d -
```

3.5.4.5 Sort embedded code objects by size

```
for uri in $(roc-obj-ls <exe>); do printf "%d: %s\n" "$(roc-obj-extract -o - "$uri" | wc
-c)" "$uri"; done | sort -n
```

3.5.4.6 Compare disassembly of gfx803 and gfx900 code objects

```
dis() { roc-obj-ls -v <exe> | grep "$1" | awk '{print $2}' | roc-obj-extract -o - |
llvm-objdump -d -; }
```

3.6 HIP Logging

HIP provides a logging mechanism, which is a convenient way of printing important information to trace HIP API and runtime codes during the execution of a HIP application. It assists the HIP development team in the development of HIP runtime and is useful for HIP application developers as well. Depending on the setting of logging level and logging mask, HIP logging will print different kinds of information, for different types of functionalities such as HIP APIs, executed kernels, queue commands, and queue contents, etc.

3.6.1 HIP Logging Level

By default, HIP logging is disabled, it can be enabled via environment setting,

AMD_LOG_LEVEL

The value of the setting controls different logging level.

```
enum LogLevel {
LOG_NONE = 0,
LOG_ERROR = 1,
LOG_WARNING = 2,
LOG_INFO = 3,
LOG_DEBUG = 4
};
```

3.6.2 HIP Logging Mask

Logging mask is designed to print types of functionalities during the execution of HIP application. It can be set as one of the following values:

```
enum LogMask {
  LOG_API       = 0x00000001, //!< API call
  LOG_CMD       = 0x00000002, //!< Kernel and Copy Commands and Barriers
  LOG_WAIT      = 0x00000004, //!< Synchronization and waiting for commands to finish
  LOG_AQL       = 0x00000008, //!< Decode and display AQL packets
  LOG_QUEUE     = 0x00000010, //!< Queue commands and queue contents
  LOG_SIG       = 0x00000020, //!< Signal creation, allocation, pool
  LOG_LOCK      = 0x00000040, //!< Locks and thread-safety code.
  LOG_KERN      = 0x00000080, //!< kernel creations and arguments, etc.
  LOG_COPY      = 0x00000100, //!< Copy debug
  LOG_COPY2     = 0x00000200, //!< Detailed copy debug
  LOG_RESOURCE  = 0x00000400, //!< Resource allocation, performance-impacting events.
  LOG_INIT      = 0x00000800, //!< Initialization and shutdown
  LOG_MISC      = 0x00001000, //!< misc debug, not yet classified
  LOG_AQL2      = 0x00002000, //!< Show raw bytes of AQL packet
  LOG_CODE      = 0x00004000, //!< Show code creation debug
  LOG_CMD2      = 0x00008000, //!< More detailed command info, including barrier
commands
  LOG_LOCATION  = 0x00010000, //!< Log message location
  LOG_ALWAYS    = 0xFFFFFFFF, //!< Log always even mask flag is zero
};
```

Once AMD_LOG_LEVEL is set, the logging mask is set as default with the value 0x7FFFFFFF. However, for different purpose of logging functionalities, logging mask can be defined as well via an environment variable,

AMD_LOG_MASK

3.6.3 HIP Logging Command

To print HIP logging information, the function is defined as

```
#define ClPrint(level, mask, format, ...)
  do {
    if (AMD_LOG_LEVEL >= level) {
      if (AMD_LOG_MASK & mask || mask == amd::LOG_ALWAYS) {
        if (AMD_LOG_MASK & amd::LOG_LOCATION) {
          amd::log_printf(level, __FILENAME__, __LINE__, format, ##__VA_ARGS__);
        } else {
          amd::log_printf(level, "", 0, format, ##__VA_ARGS__);
        }
      }
    }
  } while (false)
```

In the HIP code, call ClPrint() function with proper input variables as needed, for example,

```
ClPrint(amd::LOG_INFO, amd::LOG_INIT, "Initializing HSA stack.");
```

3.6.4 HIP Logging Example

Below is an example to enable HIP logging and get logging information during execution of hipinfo,

```
user@user-test:~/hip/bin$ export AMD_LOG_LEVEL=4
user@user-test:~/hip/bin$ ./hipinfo
:3:rocdevice.cpp            :453 : 23647210092: Initializing HSA stack.
:3:comgrctx.cpp             :33  : 23647639336: Loading COMGR library.
:3:rocdevice.cpp            :203 : 23647687108: Numa select cpu
agent[0]=0x13407c0(fine=0x13409a0,coarse=0x1340ad0) for gpu agent=0x1346150
```

```
:4:runtime.cpp              :82  : 23647698669: init
:3:hip_device_runtime.cpp   :473 : 23647698869: 5617 : [7fad295dd840] hipGetDeviceCount:
Returned hipSuccess
:3:hip_device_runtime.cpp   :502 : 23647698990: 5617 : [7fad295dd840] hipSetDevice ( 0 )
:3:hip_device_runtime.cpp   :507 : 23647699042: 5617 : [7fad295dd840] hipSetDevice:
Returned hipSuccess
--------------------------------------------------------------------------------
device#                           0
:3:hip_device.cpp           :150 : 23647699276: 5617 : [7fad295dd840]
hipGetDeviceProperties ( 0x7ffdbe7db730, 0 )
:3:hip_device.cpp           :237 : 23647699335: 5617 : [7fad295dd840]
hipGetDeviceProperties: Returned hipSuccess
Name:                             Device 7341
pciBusID:                         3
pciDeviceID:                      0
pciDomainID:                      0
multiProcessorCount:              11
maxThreadsPerMultiProcessor:      2560
isMultiGpuBoard:                  0
clockRate:                        1900 Mhz
memoryClockRate:                  875 Mhz
memoryBusWidth:                   0
clockInstructionRate:             1000 Mhz
totalGlobalMem:                   7.98 GB
maxSharedMemoryPerMultiProcessor: 64.00 KB
totalConstMem:                    8573157376
sharedMemPerBlock:                64.00 KB
canMapHostMemory:                 1
regsPerBlock:                     0
warpSize:                         32
l2CacheSize:                      0
computeMode:                      0
maxThreadsPerBlock:               1024
maxThreadsDim.x:                  1024
maxThreadsDim.y:                  1024
maxThreadsDim.z:                  1024
maxGridSize.x:                    2147483647
maxGridSize.y:                    2147483647
maxGridSize.z:                    2147483647
major:                            10
minor:                            12
concurrentKernels:                1
cooperativeLaunch:                0
cooperativeMultiDeviceLaunch:     0
arch.hasGlobalInt32Atomics:       1
arch.hasGlobalFloatAtomicExch:    1
arch.hasSharedInt32Atomics:       1
arch.hasSharedFloatAtomicExch:    1
arch.hasFloatAtomicAdd:           1
arch.hasGlobalInt64Atomics:       1
arch.hasSharedInt64Atomics:       1
arch.hasDoubles:                  1
arch.hasWarpVote:                 1
arch.hasWarpBallot:               1
arch.hasWarpShuffle:              1
arch.hasFunnelShift:              0
arch.hasThreadFenceSystem:        1
arch.hasSyncThreadsExt:           0
arch.hasSurfaceFuncs:             0
arch.has3dGrid:                   1
arch.hasDynamicParallelism:       0
gcnArch:                          1012
isIntegrated:                     0
maxTexture1D:                     65536
maxTexture2D.width:               16384
maxTexture2D.height:              16384
maxTexture3D.width:               2048
maxTexture3D.height:              2048
maxTexture3D.depth:               2048
isLargeBar:                       0
:3:hip_device_runtime.cpp   :471 : 23647701557: 5617 : [7fad295dd840] hipGetDeviceCount
( 0x7ffdbe7db714 )
:3:hip_device_runtime.cpp   :473 : 23647701608: 5617 : [7fad295dd840] hipGetDeviceCount:
Returned hipSuccess
:3:hip_peer.cpp             :76  : 23647701731: 5617 : [7fad295dd840]
hipDeviceCanAccessPeer ( 0x7ffdbe7db728, 0, 0 )
:3:hip_peer.cpp             :60  : 23647701784: 5617 : [7fad295dd840] canAccessPeer:
Returned hipSuccess
:3:hip_peer.cpp             :77  : 23647701831: 5617 : [7fad295dd840]
hipDeviceCanAccessPeer: Returned hipSuccess
peers:
:3:hip_peer.cpp             :76  : 23647701921: 5617 : [7fad295dd840]
hipDeviceCanAccessPeer ( 0x7ffdbe7db728, 0, 0 )
:3:hip_peer.cpp             :60  : 23647701965: 5617 : [7fad295dd840] canAccessPeer:
Returned hipSuccess
:3:hip_peer.cpp             :77  : 23647701998: 5617 : [7fad295dd840]
hipDeviceCanAccessPeer: Returned hipSuccess
non-peers:                        device#0

:3:hip_memory.cpp           :345 : 23647702191: 5617 : [7fad295dd840] hipMemGetInfo (
0x7ffdbe7db718, 0x7ffdbe7db720 )
:3:hip_memory.cpp           :360 : 23647702243: 5617 : [7fad295dd840] hipMemGetInfo:
Returned hipSuccess
memInfo.total:                    7.98 GB
memInfo.free:                     7.98 GB (100%)
```

3.6.5 HIP Logging Tips

- HIP logging works for both release and debug version of HIP application.
- Logging function with different logging level can be called in the code as needed.
- Information with a logging level less than AMD_LOG_LEVEL will be printed.
- If need to save the HIP logging output information in a file, just define the file at the command when running the application at the terminal, for example,

```
user@user-test:~/hip/bin$ ./hipinfo > ~/hip_log.txt
```

3.7 Debugging HIP

This section information for HIP developers to trace and debug code during execution.

3.7.1 Debugging tools

3.7.1.1 Using ltrace

ltrace is a standard linux tool which provides a message to stderr on every dynamic library call. Since ROCr and the ROCt (the ROC thunk, which is the thin user-space interface to the ROC kernel driver) are both dynamic libraries, this provides an easy way to trace the activity in these libraries. Tracing can be a powerful way to quickly observe the flow of the application before diving into the details with a command-line debugger. ltrace is a helpful tool to visualize the runtime behavior of the entire ROCm software stack. The trace can also show performance issues related to accidental calls to expensive API calls on the critical path.

Samples

Command-line to trace HIP APIs and output:

```
$ ltrace -C -e "hip*" ./hipGetChanDesc

hipGetChanDesc->hipCreateChannelDesc(0x7ffdc4b66860, 32, 0, 0) = 0x7ffdc4b66860

hipGetChanDesc->hipMallocArray(0x7ffdc4b66840, 0x7ffdc4b66860, 8, 8) = 0

hipGetChanDesc->hipGetChannelDesc(0x7ffdc4b66848, 0xa63990, 5, 1) = 0

hipGetChanDesc->hipFreeArray(0xa63990, 0, 0x7f8c7fe13778, 0x7ffdc4b66848) = 0

PASSED!

+++ exited (status 0) +++
```

Command-line only trace hsa APIs and output:

```
$ ltrace -C -e "hsa*" ./hipGetChanDesc

libamdhip64.so.4->hsa_init(0, 0x7fff325a69d0, 0x9c80e0, 0 <unfinished ...>

libhsa-runtime64.so.1->hsaKmtOpenKFD(0x7fff325a6590, 0x9c38c0, 0, 1) = 0

libhsa-runtime64.so.1->hsaKmtGetVersion(0x7fff325a6608, 0, 0, 0) = 0

libhsa-runtime64.so.1->hsaKmtReleaseSystemProperties(3, 0x80084b01, 0, 0) = 0

libhsa-runtime64.so.1->hsaKmtAcquireSystemProperties(0x7fff325a6610, 0, 0, 1) = 0

libhsa-runtime64.so.1->hsaKmtGetNodeProperties(0, 0x7fff325a66a0, 0, 0) = 0

libhsa-runtime64.so.1->hsaKmtGetNodeMemoryProperties(0, 1, 0x9c42b0, 0x936012) = 0
...

<... hsaKmtCreateEvent resumed> )                 = 0

libhsa-runtime64.so.1->hsaKmtAllocMemory(0, 4096, 64, 0x7fff325a6690) = 0

libhsa-runtime64.so.1->hsaKmtMapMemoryToGPUNodes(0x7f1202749000, 4096, 0x7fff325a6690,
0) = 0

libhsa-runtime64.so.1->hsaKmtCreateEvent(0x7fff325a6700, 0, 0, 0x7fff325a66f0) = 0

libhsa-runtime64.so.1->hsaKmtAllocMemory(1, 0x100000000, 576, 0x7fff325a67d8) = 0

libhsa-runtime64.so.1->hsaKmtAllocMemory(0, 8192, 64, 0x7fff325a6790) = 0

libhsa-runtime64.so.1->hsaKmtMapMemoryToGPUNodes(0x7f120273c000, 8192, 0x7fff325a6790,
0) = 0

libhsa-runtime64.so.1->hsaKmtAllocMemory(0, 4096, 4160, 0x7fff325a6450) = 0

libhsa-runtime64.so.1->hsaKmtMapMemoryToGPUNodes(0x7f120273a000, 4096, 0x7fff325a6450,
0) = 0

libhsa-runtime64.so.1->hsaKmtSetTrapHandler(1, 0x7f120273a000, 4096, 0x7f120273c000) = 0

<... hsa_init resumed> )                          = 0

libamdhip64.so.4->hsa_system_get_major_extension_table(513, 1, 24, 0x7f1202597930) = 0

libamdhip64.so.4->hsa_iterate_agents(0x7f120171f050, 0, 0x7fff325a67f8, 0 <unfinished
...>

libamdhip64.so.4->hsa_agent_get_info(0x94f110, 17, 0x7fff325a67e8, 0) = 0

libamdhip64.so.4->hsa_amd_agent_iterate_memory_pools(0x94f110, 0x7f1201722816,
0x7fff325a67f0, 0x7f1201722816 <unfinished ...>

libamdhip64.so.4->hsa_amd_memory_pool_get_info(0x9c7fb0, 0, 0x7fff325a6744,
0x7fff325a67f0) = 0

libamdhip64.so.4->hsa_amd_memory_pool_get_info(0x9c7fb0, 1, 0x7fff325a6748,
0x7f1200d82df4) = 0

...

<... hsa_amd_agent_iterate_memory_pools resumed> ) = 0

libamdhip64.so.4->hsa_agent_get_info(0x9dbf30, 17, 0x7fff325a67e8, 0) = 0

<... hsa_iterate_agents resumed> )                = 0

libamdhip64.so.4->hsa_agent_get_info(0x9dbf30, 0, 0x7fff325a6850, 3) = 0

libamdhip64.so.4->hsa_agent_get_info(0x9dbf30, 0xa000, 0x9e7cd8, 0) = 0

libamdhip64.so.4->hsa_agent_iterate_isas(0x9dbf30, 0x7f1201720411, 0x7fff325a6760,
0x7f1201720411) = 0
```

```
libamdhip64.so.4->hsa_isa_get_info_alt(0x94e7c8, 0, 0x7fff325a6728, 1) = 0

libamdhip64.so.4->hsa_isa_get_info_alt(0x94e7c8, 1, 0x9e7f90, 0) = 0

libamdhip64.so.4->hsa_agent_get_info(0x9dbf30, 4, 0x9e7ce8, 0) = 0

...

<... hsa_amd_memory_pool_allocate resumed> )     = 0

libamdhip64.so.4->hsa_ext_image_create(0x9dbf30, 0xa1c4c8, 0x7f10f2800000, 3 <unfinished
...>

libhsa-runtime64.so.1->hsaKmtAllocMemory(0, 4096, 64, 0x7fff325a6740) = 0

libhsa-runtime64.so.1->hsaKmtQueryPointerInfo(0x7f1202736000, 0x7fff325a65e0, 0, 0) = 0

libhsa-runtime64.so.1->hsaKmtMapMemoryToGPUNodes(0x7f1202736000, 4096, 0x7fff325a66e8,
0) = 0

<... hsa_ext_image_create resumed> )             = 0

libamdhip64.so.4->hsa_ext_image_destroy(0x9dbf30, 0x7f1202736000, 0x9dbf30, 0
<unfinished ...>

libhsa-runtime64.so.1->hsaKmtUnmapMemoryToGPU(0x7f1202736000, 0x7f1202736000, 4096,
0x9c8050) = 0

libhsa-runtime64.so.1->hsaKmtFreeMemory(0x7f1202736000, 4096, 0, 0) = 0

<... hsa_ext_image_destroy resumed> )            = 0

libamdhip64.so.4->hsa_amd_memory_pool_free(0x7f10f2800000, 0x7f10f2800000, 256,
0x9e76f0) = 0

PASSED!
```

3.7.1.2 Using ROCgdb

HIP developers on ROCm can use AMD's ROCgdb for debugging and profiling. ROCgdb is the ROCm source-level debugger for Linux, based on GDB, the GNU source-level debugger. It is similar to cuda-gdb. It can be used with debugger frontends, such as eclipse, vscode, or gdb-dashboard.

For details, see *https://github.com/ROCm-Developer-Tools/ROCgdb*.

The sample below shows you how to use the ROCgdb run and debug HIP applications.

Note, ROCgdb is installed with the ROCM package in the folder */opt/rocm/bin*.

```
$ export PATH=$PATH:/opt/rocm/bin

$ rocgdb ./hipTexObjPitch

GNU gdb (rocm-dkms-no-npi-hipclang-6549) 10.1

Copyright (C) 2020 Free Software Foundation, Inc.

License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

...

For bug reporting instructions, please see:

<https://github.com/ROCm-Developer-Tools/ROCgdb/issues>.

Find the GDB manual and other documentation resources online at:

    <http://www.gnu.org/software/gdb/documentation/>.

...

Reading symbols from ./hipTexObjPitch...

(gdb) break main

Breakpoint 1 at 0x4013d1: file /home/test/hip/tests/src/texture/hipTexObjPitch.cpp, line
98.

(gdb) run

Starting program: /home/test/hip/build/directed_tests/texture/hipTexObjPitch

[Thread debugging using libthread_db enabled]

Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".



Breakpoint 1, main ()

    at /home/test/hip/tests/src/texture/hipTexObjPitch.cpp:98

98            texture2Dtest<float>();

(gdb)c
```

3.7.2 Debugging HIP Applications

The following example shows how to get useful information from the debugger while running an application, which causes the GPUVM fault issue.

```
Memory access fault by GPU node-1 on address 0x5924000. Reason: Page not present or
supervisor privilege.



Program received signal SIGABRT, Aborted.

[Switching to Thread 0x7fffdffb5700 (LWP 14893)]

0x00007ffff2057c37 in __GI_raise (sig=sig@entry=6) at
../nptl/sysdeps/unix/sysv/linux/raise.c:56

56      ../nptl/sysdeps/unix/sysv/linux/raise.c: No such file or directory.

(gdb) bt

#0  0x00007ffff2057c37 in __GI_raise (sig=sig@entry=6) at
../nptl/sysdeps/unix/sysv/linux/raise.c:56

#1  0x00007ffff205b028 in __GI_abort () at abort.c:89

#2  0x00007ffff6f960eb in ?? () from /opt/rocm/hsa/lib/libhsa-runtime64.so.1

#3  0x00007ffff6f99ea5 in ?? () from /opt/rocm/hsa/lib/libhsa-runtime64.so.1

#4  0x00007ffff6f78107 in ?? () from /opt/rocm/hsa/lib/libhsa-runtime64.so.1

#5  0x00007ffff744f184 in start_thread (arg=0x7fffdffb5700) at pthread_create.c:312

#6  0x00007ffff211b37d in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:111

(gdb) info threads

  Id   Target Id         Frame

  4    Thread 0x7fffdd521700 (LWP 14895) "caffe" pthread_cond_wait@@GLIBC_2.3.2 () at
../nptl/sysdeps/unix/sysv/linux/x86_64/pthread_cond_wait.S:185

  3    Thread 0x7fffddd22700 (LWP 14894) "caffe" pthread_cond_wait@@GLIBC_2.3.2 () at
../nptl/sysdeps/unix/sysv/linux/x86_64/pthread_cond_wait.S:185

* 2    Thread 0x7fffdffb5700 (LWP 14893) "caffe" 0x00007ffff2057c37 in __GI_raise
(sig=sig@entry=6) at ../nptl/sysdeps/unix/sysv/linux/raise.c:56

  1    Thread 0x7ffff7fa6ac0 (LWP 14892) "caffe" 0x00007ffff6f934d5 in ?? () from
/opt/rocm/hsa/lib/libhsa-runtime64.so.1

(gdb) thread 1

[Switching to thread 1 (Thread 0x7ffff7fa6ac0 (LWP 14892))]
```

```
#0  0x00007ffff6f934d5 in ?? () from /opt/rocm/hsa/lib/libhsa-runtime64.so.1

(gdb) bt

#0  0x00007ffff6f934d5 in ?? () from /opt/rocm/hsa/lib/libhsa-runtime64.so.1

#1  0x00007ffff6f929ba in ?? () from /opt/rocm/hsa/lib/libhsa-runtime64.so.1

#2  0x00007fffe080beca in HSADispatch::waitComplete() () from
/opt/rocm/hcc/lib/libmcwamp_hsa.so

#3  0x00007fffe080415f in HSADispatch::dispatchKernelAsync(Kalmar::HSAQueue*, void
const*, int, bool) () from /opt/rocm/hcc/lib/libmcwamp_hsa.so

#4  0x00007fffe080238e in
Kalmar::HSAQueue::dispatch_hsa_kernel(hsa_kernel_dispatch_packet_s const*, void const*,
unsigned long, hc::completion_future*) () from /opt/rocm/hcc/lib/libmcwamp_hsa.so

#5  0x00007ffff7bb7559 in hipModuleLaunchKernel () from /opt/rocm/hip/lib/libhip_hcc.so

#6  0x00007ffff2e6cd2c in mlopen::HIPOCKernel::run (this=0x7fffffffb5a8,
args=0x7fffffffb2a8, size=80) at /root/MIOpen/src/hipoc/hipoc_kernel.cpp:15

...
```

3.7.3 Useful Environment Variables

HIP provides environment variables which allow HIP, hip-clang, or HSA driver to disable features or optimizations. These are not intended for production but can be useful to diagnose synchronization problems in the application (or driver).

See the sections below for a description of environment variables. They are supported on the ROCm path.

3.7.3.1 Kernel Enqueue Serialization

Developers can control kernel command serialization from the host using the environment variable,

- AMD_SERIALIZE_KERNEL, for serializing kernel enqueue.
- AMD_SERIALIZE_KERNEL = 1, Wait for completion before enqueue,
- AMD_SERIALIZE_KERNEL = 2, Wait for completion after enqueue,
- AMD_SERIALIZE_KERNEL = 3, Both. Or AMD_SERIALIZE_COPY, for serializing copies.
- AMD_SERIALIZE_COPY = 1, Wait for completion before enqueue
- AMD_SERIALIZE_COPY = 2, Wait for completion after enqueue
- AMD_SERIALIZE_COPY = 3, Both.

Note, HIP runtime can wait for GPU idle before/after any GPU command depending on the environment setting.

3.7.3.2 Making Device Visible

For system with multiple devices, it is possible to make only certain device(s) visible to HIP via the setting environment varible, HIP_VISIBLE_DEVICES. Only devices whose index is present in the sequence are visible to HIP.

For example,

```
$ HIP_VISIBLE_DEVICES=0,1
```

or in the appliation,

```
if (totalDeviceNum > 2) {

  setenv("HIP_VISIBLE_DEVICES", "0,1,2", 1);

  assert(getDeviceNumber(false) == 3);

  ... ...

}
```

3.7.3.3 Dump code object

Developers can dump code object to anylize compiler related issues via setting environment variable, GPU_DUMP_CODE_OBJECT

3.7.3.4 HSA related environment variables

HSA provides environment varibles help to analyze issues in drivers or hardware. For example,

- HSA_ENABLE_SDMA=0 It causes host-to-device and device-to-host copies to use compute shader blit kernels rather than the dedicated DMA copy engines. Compute shader copies have low latency (typically < 5us) and can achieve approximately 80% of the bandwidth of the DMA copy engine. This environment variable is useful to isolate issues with the hardware copy engines.

- HSA_ENABLE_INTERRUPT=0 Causes completion signals to be detected with memory-based polling rather than interrupts. This environment variable can be useful to diagnose interrupt storm issues in the driver.

3.7.4 General Debugging Tips

- 'gdb --args' can be used to conviently pass the executable and arguments to gdb.
- From GDB, you can set environment variables "set env". Note the command does not use an '=' sign:

```
(gdb) set env AMD_SERIALIZE_KERNEL 3
```

- The fault will be caught by the runtime but was actually generated by an asynchronous command running on the GPU. So, the GDB backtrace will show a path in the runtime.
- To determine the true location of the fault, force the kernels to execute synchronously by seeing the environment variables AMD_SERIALIZE_KERNEL=3 AMD_SERIALIZE_COPY=3. This will force HIP runtime to wait for the kernel to finish executing before retuning. If the fault occurs during the execution of a kernel, you can see the code which launched the kernel inside the backtrace. A bit of guesswork is required to determine which thread is actually causing the issue - typically it will the thread which is waiting inside the libhsa-runtime64.so.
- VM faults inside kernels can be caused by:
    - incorrect code (ie a for loop which extends past array boundaries),
    - memory issues - kernel arguments which are invalid (null pointers, unregistered host pointers, bad pointers),
    - synchronization issues,
    - compiler issues (incorrect code generation from the compiler),
    - runtime issues.

3.8 HIP Version

HIP version definition is updated since the ROCm v4.2 release as follows:

```
HIP_VERSION=HIP_VERSION_MAJOR * 10000000 + HIP_VERSION_MINOR * 100000 +
HIP_VERSION_PATCH)
```

HIP version can be queried from the following HIP API call,

```
hipRuntimeGetVersion(&runtimeVersion);
```

The version returned will be greater than the version in previous ROCm releases.

Chapter 4 Transiting from CUDA to HIP

4.1 Transition Tool: HIPIFY

4.1.1 Sample and Practice

Here is a simple test, which shows how to use hipify-Perl to port CUDA code to HIP. See a related *blog* that explains the example. Now, it is even simpler and requires no manual modification to the hipified source code - just hipify and compile:

1. Add hip/bin path to the PATH.

```
$ export PATH=$PATH:[MYHIP]/bin
```

2. Define the environment variable.

```
$ export HIP_PATH=[MYHIP]
```

3. Build an executable file.

```
$ cd ~/hip/samples/0_Intro/square
$ make
/home/user/hip/bin/hipify-perl square.cu > square.cpp
/home/user/hip/bin/hipcc  square.cpp -o square.out
/home/user/hip/bin/hipcc -use-staticlib  square.cpp -o square.out.static
```

4. Execute the file.

```
$ ./square.out
info: running on device Vega20 [Radeon Pro W5500]
info: allocate host mem (  7.63 MB)
info: allocate device mem (  7.63 MB)
info: copy Host2Device
info: launch 'vector_square' kernel
info: copy Device2Host
info: check result
PASSED!
```

4.2 HIP Porting Process

4.2.1 Porting a New CUDA Project

4.2.1.1 General Tips

- Starting the port on a CUDA machine is often the easiest approach since you can incrementally port pieces of the code to HIP while leaving the rest in CUDA. (Recall that on CUDA machines HIP is just a thin layer over CUDA, so the two code types can interoperate on nvcc platforms.) Also, the HIP port can be compared with the original CUDA code for function and performance.

- Once the CUDA code is ported to HIP and is running on the CUDA machine, compile the HIP code using the HIP compiler on an AMD machine.

- HIP ports can replace CUDA versions: HIP can deliver the same performance as a native CUDA implementation, with the benefit of portability to both Nvidia and AMD architectures as well as a path to future C++ standard support. You can handle platform-specific features through the conditional compilation or by adding them to the open-source HIP infrastructure.

- Use *bin/hipconvertinplace-perl.sh* to hipify all code files in the CUDA source directory.

4.2.1.2 Scanning existing CUDA code to scope the porting effort

The *hipexamine-perl.sh* tool will scan a source directory to determine which files contain CUDA code and how much of that code can be automatically hipified.

```
> cd examples/rodinia_3.0/cuda/kmeans
> $HIP_DIR/bin/hipexamine-perl.sh.
info: hipify ./kmeans.h =====>
info: hipify ./unistd.h =====>
info: hipify ./kmeans.c =====>
info: hipify ./kmeans_cuda_kernel.cu =====>
  info: converted 40 CUDA->HIP refs( dev:0 mem:0 kern:0 builtin:37 math:0 stream:0
event:0 err:0 def:0 tex:3 other:0 ) warn:0 LOC:185
info: hipify ./getopt.h =====>
info: hipify ./kmeans_cuda.cu =====>
  info: converted 49 CUDA->HIP refs( dev:3 mem:32 kern:2 builtin:0 math:0 stream:0
event:0 err:0 def:0 tex:12 other:0 ) warn:0 LOC:311
info: hipify ./rmse.c =====>
info: hipify ./cluster.c =====>
info: hipify ./getopt.c =====>
info: hipify ./kmeans_clustering.c =====>
info: TOTAL-converted 89 CUDA->HIP refs( dev:3 mem:32 kern:2 builtin:37 math:0 stream:0
event:0 err:0 def:0 tex:15 other:0 ) warn:0 LOC:3607
  kernels (1 total) :   kmeansPoint(1)
```

hipexamine-perl scans each code file (cpp, c, h, hpp, etc.) found in the specified directory:

- Files with no CUDA code (kmeans.h) print a one-line summary just listing the source file name.

- Files with CUDA code print a summary of what was found - for example, the kmeans_cuda_kernel.cu file:

```
info: hipify ./kmeans_cuda_kernel.cu =====>
info: converted 40 CUDA->HIP refs( dev:0 mem:0 kern:0 builtin:37 math:0 stream:0 event:0
```

- Information in kmeans_cuda_kernel.cu :
    - How many CUDA calls were converted to HIP (40)
    - Breakdown of the CUDA functionality used (dev:0 mem:0 etc). This file uses many CUDA builtins (37) and texture functions (3).
    - Warning for code that looks like CUDA API but was not converted (0 in this file).
    - Count Lines-of-Code (LOC) - 185 for this file.

- hipexamine-perl also presents a summary at the end of the process for the statistics collected across all files. This has a similar format to the per-file reporting, and also includes a list of all kernels which have been called. An example from above:

```
info: TOTAL-converted 89 CUDA->HIP refs( dev:3 mem:32 kern:2 builtin:37 math:0 stream:0
event:0 err:0 def:0 tex:15 other:0 ) warn:0 LOC:3607
kernels (1 total) :   kmeansPoint(1)
```

4.2.1.3 Converting a project in-place

```
> hipify-perl --inplace
```

For each input file FILE, this script will: - If FILE.prehip file does not exist, copy the original code to a new file with extension.prehip. Then hipify the code file. If "FILE.prehip" file exists, hipify FILE.prehip and save to FILE.

This is useful for testing improvements to the hipify toolset.

The hipconvertinplace-perl.sh script will perform an in-place conversion for all code files in the specified directory. This can be quite handy when dealing with an existing CUDA code base since the script preserves the existing directory structure and filenames - and includes work. After converting in-place, you can review the code to add additional parameters to directory names.

```
> hipconvertinplace-perl.sh MY_SRC_DIR
```

4.2.1.4 Library Equivalents

CUDA Library	ROCm Library	Comment
cuBLAS	rocBLAS	Basic Linear Algebra Subroutines
cuFFT	rocFFT	Fast Fourier Transfer Library
cuSPARSE	rocSPARSE	Sparse BLAS + SPMV
cuSolver	rocSOLVER	Lapack library
AMG-X	rocALUTION	Sparse iterative solvers and preconditioners with Geometric and Algebraic MultiGrid
Thrust	rocThrust	C++ parallel algorithms library
CUB	rocPRIM	Low Level Optimized Parallel Primitives
cuDNN	MIOpen	Deep learning Solver Library
cuRAND	rocRAND	Random Number Generator Library
EIGEN	EIGEN	C++ template library for linear algebra: matrices, vectors, numerical solvers,
NCCL	RCCL	Communications Primitives Library based on the MPI equivalents

4.2.2 Distinguishing Compiler Modes

4.2.2.1 Identifying HIP Target Platform

All HIP projects target either AMD or NVIDIA platform. The platform affects the headers that are included and libraries that are used for linking.

- HIP_PLATFORM_AMD is defined if the HIP platform targets AMD.
  Note, HIP_PLATFORM_HCC was previously defined if the HIP platform targeted AMD. It is now deprecated.
- HIP_PLATFORM_NVDIA is defined if the HIP platform targets NVIDIA.
  Note, HIP_PLATFORM_NVCC was previously defined if the HIP platform targeted NVIDIA. It is now deprecated.

4.2.2.2 Identifying the Compiler: HIP-Clang or NVIDIA

Often, it is useful to know whether the underlying compiler is HIP-Clang or NVIDIA. This knowledge can guard platform-specific code or aid in platform-specific performance tuning.

```
#ifdef __HIP_PLATFORM_AMD__
// Compiled with HIP-Clang
#endif

#ifdef __HIP_PLATFORM_NVIDIA__
// Compiled with nvcc
//  Could be compiling with CUDA language extensions enabled (for example, a ".cu file)
//  Could be in pass-through mode to an underlying host compile OR (for example, a .cpp
file)
#ifdef __CUDACC__
// Compiled with nvcc (CUDA language extensions enabled)
```

HIP-Clang directly generates the host code (using the Clang x86 target) without passing the code to another host compiler. Thus, they have no equivalent of the __CUDACC__ define.

4.2.2.3 Identifying Current Compilation Pass: Host or Device

NVCC makes two passes over the code: one for host code and one for device code. HIP-Clang will have multiple passes over the code: one for the host code, and one for each architecture on the device code. __HIP_DEVICE_COMPILE__ is set to a nonzero value when the compiler (HIP-Clang or nvcc) is compiling code for a device inside a __global__ kernel or for a device function. __HIP_DEVICE_COMPILE__ can replace #ifdef checks on the __CUDA_ARCH__ define.

```
// #ifdef __CUDA_ARCH__
#if __HIP_DEVICE_COMPILE__
```

Unlike __CUDA_ARCH__, the __HIP_DEVICE_COMPILE__ value is 1 or undefined, and it does not represent the feature capability of the target device.

4.2.3 Compiler Defines: Summary

Define	HIP-Clang	nvcc	Other (GCC, ICC, Clang, etc.)
HIP-related defines:			
__HIP_PLATFORM_AMD__	Defined	Undefined	Defined if targeting AMD platform; undefined otherwise
__HIP_PLATFORM_NVIDIA__	Undefined	Defined	Defined if targeting NVIDIA platform; undefined otherwise
__HIP_DEVICE_COMPILE__	1 if compiling for device; undefined if compiling for host	1 if compiling for device; undefined if compiling for host	Undefined
__HIPCC__	Defined	Defined	Undefined
__HIP_ARCH_*	0 or 1 depending on feature support (see below)	0 or 1 depending on feature support (see below)	0
nvcc-related defines:			
__CUDACC__	Defined if source code is compiled by nvcc; undefined otherwise	Undefined	
__NVCC__	Undefined	Defined	Undefined
__CUDA_ARCH__	Undefined	Unsigned representing compute capability (e.g., "130") if in device code; 0 if in host code	Undefined
hip-clang-related defines:			
__HIP__	Defined	Undefined	Undefined
HIP-Clang common defines:			
__clang__	Defined	Defined	Undefined

4.3 Identifying Architecture Features

4.3.1 HIP_ARCH Defines

Some CUDA code tests __CUDA_ARCH__ for a specific value to determine whether the machine supports a certain architectural feature. For instance,

```
#if (__CUDA_ARCH__ >= 130)
// doubles are supported
```

This type of code requires special attention since AMD and CUDA devices have different architectural capabilities. Moreover, you cannot determine the presence of a feature using a simple comparison against an architecture's version number. HIP provides a set of defines and device properties to query whether a specific architectural feature is supported.

The __HIP_ARCH_* defines can replace comparisons of __CUDA_ARCH__ values:

```
//#if (__CUDA_ARCH__ >= 130)   // non-portable
if __HIP_ARCH_HAS_DOUBLES__ {  // portable HIP feature query
   // doubles are supported
}
```

For host code, the __HIP_ARCH__* defines are set to 0. You should only use the HIP_ARCH fields in the device code.

4.3.2 Device-Architecture Properties

The host code should query the architecture feature flags in the device properties that hipGetDeviceProperties returns, rather than testing the "major" and "minor" fields directly:

```
hipGetDeviceProperties(&deviceProp, device);
//if ((deviceProp.major == 1 && deviceProp.minor < 2))  // non-portable
if (deviceProp.arch.hasSharedInt32Atomics) {            // portable HIP feature query
    // has shared int32 atomic operations ...
}
```

4.3.3 Table of Architecture Properties

The table below shows the full set of architectural properties that HIP supports.

Define (use only in device code)	Device Property (run-time query)	Comment
32-bit atomics:		
__HIP_ARCH_HAS_GLOBAL_INT32_ATOMICS__	hasGlobalInt32Atomics	32-bit integer atomics for global memory
__HIP_ARCH_HAS_GLOBAL_FLOAT_ATOMIC_EXCH__	hasGlobalFloatAtomicExch	32-bit float atomic exchange for global memory
__HIP_ARCH_HAS_SHARED_INT32_ATOMICS__	hasSharedInt32Atomics	32-bit integer atomics for shared memory
__HIP_ARCH_HAS_SHARED_FLOAT_ATOMIC_EXCH__	hasSharedFloatAtomicExch	32-bit float atomic exchange for shared memory
__HIP_ARCH_HAS_FLOAT_ATOMIC_ADD__	hasFloatAtomicAdd	32-bit float atomic add in global and shared memory
64-bit atomics		
__HIP_ARCH_HAS_GLOBAL_INT64_ATOMICS__	hasGlobalInt64Atomics	64-bit integer atomics for global memory
__HIP_ARCH_HAS_SHARED_INT64_ATOMICS__	hasSharedInt64Atomics	64-bit integer atomics for shared memory
Doubles		
__HIP_ARCH_HAS_DOUBLES__	hasDoubles	Double-precision floating point
Warp cross-lane operations:		
__HIP_ARCH_HAS_WARP_VOTE__	hasWarpVote	Warp vote instructions (any, all)
__HIP_ARCH_HAS_WARP_BALLOT__	hasWarpBallot	Warp ballot instructions
__HIP_ARCH_HAS_WARP_SHUFFLE__	hasWarpShuffle	Warp shuffle operations (shfl_*)
__HIP_ARCH_HAS_WARP_FUNNEL_SHIFT__	hasFunnelShift	Funnel shift two input words into one
Sync		
__HIP_ARCH_HAS_THREAD_FENCE_SYSTEM__	hasThreadFenceSystem	threadfence_system
__HIP_ARCH_HAS_SYNC_THREAD_EXT__	hasSyncThreadsExt	syncthreads_count, syncthreads_and, syncthreads_or
Miscellaneous		
__HIP_ARCH_HAS_SURFACE_FUNCS__	hasSurfaceFuncs	

Define (use only in device code)	Device Property (run-time query)	Comment
__HIP_ARCH_HAS_3DGRID__	has3dGrid	Grids and groups are 3D
__HIP_ARCH_HAS_DYNAMIC_PARALLEL__	hasDynamicParallelism	

4.3.4 Finding HIP

Makefiles can use the following syntax to conditionally provide a default HIP_PATH if one does not exist:

```
HIP_PATH ?= $(shell hipconfig --path)
```

4.3.5 Identifying HIP Runtime

HIP can depend on ROCclr, or CUDA as runtime.

The AMD platform HIP uses the Radeon Open Compute common language runtime called ROCclr. ROCclr is a virtual device interface that HIP runtimes to interact with different backends, allowing runtimes to work on Linux and Windows without much effort.

On the NVIDIA platform, HIP is just a thin layer on top of CUDA. On a non-AMD platform, HIP runtime determines if CUDA is available and can be used. If available, HIP_PLATFORM is set to NVIDIA, and underneath the CUDA path is used.

4.3.6 hipLaunchKernel

hipLaunchKernel is a variadic macro that accepts as parameters the launch configurations (grid dims, group dims, stream, dynamic shared size) followed by a variable number of kernel arguments. This sequence is then expanded into the appropriate kernel launch syntax depending on the platform. While this can be a convenient single-line kernel launch syntax, the macro implementation can cause issues when nested inside other macros. For example, consider the following:

```
// Will cause compile error:
#define MY_LAUNCH(command, doTrace) \
{\
    if (doTrace) printf ("TRACE: %s\n", #command); \
    (command);   /* The nested ( ) will cause compile error */\
}

MY_LAUNCH (hipLaunchKernel(vAdd, dim3(1024), dim3(1), 0, 0, Ad), true, "firstCall");
```

NOTE: Avoid nesting macro parameters inside parenthesis - here's an alternative that will work:

```
#define MY_LAUNCH(command, doTrace) \
{\
    if (doTrace) printf ("TRACE: %s\n", #command); \
    command;\
}

MY_LAUNCH (hipLaunchKernel(vAdd, dim3(1024), dim3(1), 0, 0, Ad), true, "firstCall");
```

4.3.7 Compiler Options

HIPcc is a portable compiler driver that calls nvcc or HIP-Clang (depending on the target system) and attach all required include and library options. It passes options through to the target compiler. Tools that call hipcc must ensure the compiler options are appropriate for the target compiler. The hipconfig script may help in identifying the target platform, compiler, and runtime. It can also help set options appropriately.

4.3.7.1 Compiler Options Supported on AMD Platforms

Option	Description
--amdgpu-target=<gpu_arch>	[DEPRECATED] This option is replaced by `--offload-arch=<target>`. Generate code for the given GPU target. Supported targets are gfx701, gfx801, gfx802, gfx803, gfx900, gfx906, gfx908, gfx1010, gfx1011, gfx1012, gfx1030, gfx1031. This option could appear multiple times on the same command line to generate a fat binary for multiple targets.
--fgpu-rdc	Generate relocatable device code, which allows kernels or device functions calling device functions in different translation units.
-ggdb	Equivalent to `-g` plus tuning for GDB. This is recommended when using ROCm's GDB to debug GPU code.
--gpu-max-threads-per-block=<num>	Generate code to support up to the specified number of threads per block.
-O<n>	Specify the optimization level.
-offload-arch=<target>	Specify the AMD GPU [target ID] *https://clang.llvm.org/docs/ClangOffloadBundlerFileFormat.html#target-id*
-save-temps	Save the compiler-generated intermediate files.
-v	Show the compilation steps.

4.3.7.2 Option for specifying GPU processor

To specify target ID, use

--offload-arch=X

NOTE: For backward compatibility, hipcc also accepts *--amdgpu-target=X* for specifying target ID. However, it will be deprecated in future releases.

4.3.8 Linking Issues

4.3.8.1 Linking with hipcc

hipcc adds the necessary libraries for HIP as well as for the accelerator compiler (nvcc or AMD compiler). It is recommended to link with hipcc since it automatically links the binary to the necessary HIP runtime libraries. It also enables linking and managing GPU objects.

-lm Option

NOTE: hipcc adds *-lm* by default to the link command.

4.4 Linking Code with Other Compilers

CUDA code often uses nvcc for accelerator code (defining and launching kernels, typically defined in .cu or .cuh files). It also uses a standard compiler (g++) for the rest of the application. nvcc is a preprocessor that employs a standard host compiler (gcc) to generate the host code. The code compiled using this tool can employ only the intersection of language features supported by both nvcc and the host compiler. In some cases, you must take care to ensure the data types and alignment of the host compiler are identical to those of the device compiler. Only some host compilers are supported---for example, recent nvcc versions lack Clang host-compiler capability.

HIP-Clang generates both device and host code using the same Clang-based compiler. The code uses the same API as gcc, which allows code generated by different gcc-compatible compilers to be linked together. For example, code compiled using HIP-Clang can link with code compiled using "standard" compilers (such as gcc, ICC, and Clang). Take care to ensure all compilers use the same standard C++ header and library formats.

4.4.1 libc++ and libstdc++

hipcc links to libstdc++ by default. This provides better compatibility between g++ and HIP.

If you pass "--stdlib=libc++" to hipcc, hipcc will use the libc++ library. Generally, libc++ provides a broader set of C++ features while libstdc++ is the standard for more compilers (notably including g++).

When cross-linking C++ code, any C++ functions that use types from the C++ standard library (including std::string, std::vector and other containers) must use the same standard-library implementation. They include the following:

- Functions or kernels defined in HIP-Clang that are called from a standard compiler
- Functions defined in a standard compiler are called from HIP-Clang.
- Applications with these interfaces should use the default libstdc++ linking.

Applications that are compiled entirely with hipcc, and which benefit from advanced C++ features not supported in libstdc++, and which do not require portability to nvcc, may choose to use libc++.

4.4.2 HIP Headers (hip_runtime.h, hip_runtime_api.h)

The hip_runtime.h and hip_runtime_api.h files define the types, functions and enumerations needed to compile a HIP program:

- hip_runtime_api.h: defines all the HIP runtime APIs (e.g., hipMalloc) and the types required to call them. A source file that is only calling HIP APIs but neither defines nor launches any kernels can include hip_runtime_api.h. hip_runtime_api.h uses no custom hc language features and can be compiled using a standard C++ compiler.

- hip_runtime.h: included in hip_runtime_api.h. It additionally provides the types and defines required to create and launch kernels. It can be compiled using a standard C++ compiler, but will expose a subset of the available functions.

CUDA has slightly different content for these two files. In some cases, you may need to convert hipified code to include the richer hip_runtime.h instead of hip_runtime_api.h.

4.4.3 Using a Standard C++ Compiler

You can compile hip_runtime_api.h using a standard C or C++ compiler (e.g., gcc or ICC). The HIP include paths and defines (__HIP_PLATFORM_AMD__ or _HIP_PLATFORM_NVIDIA__) must pass to the standard compiler; hipconfig then returns the necessary options:

```
> hipconfig --cxx_config
 -D__HIP_PLATFORM_AMD__ -I/home/user1/hip/include
```

You can capture the hipconfig output and passed it to the standard compiler; below is a sample makefile syntax:

```
CPPFLAGS += $(shell $(HIP_PATH)/bin/hipconfig --cpp_config)
```

Nvcc includes some headers by default. However, HIP does not include default headers, and instead, all required files must be explicitly included. Specifically, files that call HIP run-time APIs or define HIP kernels must explicitly include the appropriate HIP headers. If the compilation process reports that it cannot find necessary APIs (for example, "error: identifier ‘hipSetDevice’ is undefined"), ensure that the file includes hip_runtime.h (or hip_runtime_api.h, if appropriate). The hipify-perl script automatically converts "cuda_runtime.h" to "hip_runtime.h," and it converts "cuda_runtime_api.h" to "hip_runtime_api.h", but it may miss nested headers or macros.

4.4.3.1 cuda.h

The HIP-Clang path provides an empty cuda.h file. Some existing CUDA programs include this file but do not require any of the functions.

4.4.4 Choosing HIP File Extensions

Many existing CUDA projects use the ".cu" and ".cuh" file extensions to indicate code that should be run through the nvcc compiler. For quick HIP ports, leaving these file extensions unchanged is often easier, as it minimizes the work required to change file names in the directory and #include statements in the files.

For new projects or ports which can be re-factored, we recommend the use of the extension ".hip.cpp" for source files, and ".hip.h" or ".hip.hpp" for header files. This indicates that the code is standard C++ code, but also provides a unique indication for *make* tools to run hipcc when appropriate.

4.5 Workarounds

4.5.1 memcpyToSymbol

HIP support for hipMemcpyToSymbol is complete. This feature allows a kernel to define a device-side data symbol that can be accessed on the host side. The symbol can be in __constant or device space.

Note that the symbol name needs to be encased in the HIP_SYMBOL macro, as shown in the code example below. This also applies to hipMemcpyFromSymbol, hipGetSymbolAddress, and hipGetSymbolSize.

For example, Device Code:

```
#include<hip/hip_runtime.h>
#include<hip/hip_runtime_api.h>
#include<iostream>
#define HIP_ASSERT(status) \
    assert(status == hipSuccess)
#define LEN 512
#define SIZE 2048
__constant__ int Value[LEN];
__global__ void Get(hipLaunchParm lp, int *Ad)
{
    int tid =threadIdx.x + blockIdx.x *blockDim.x;
    Ad[tid] = Value[tid];
}
int main()
{
    int *A, *B, *Ad;
    A = new int[LEN];
    B = new int[LEN];
    for(unsigned i=0;i<LEN;i++)
    {
        A[i] = -1*i;
        B[i] = 0;
    }
    HIP_ASSERT(hipMalloc((void**)&Ad, SIZE));
    HIP_ASSERT(hipMemcpyToSymbol(HIP_SYMBOL(Value), A, SIZE, 0, hipMemcpyHostToDevice));
    hipLaunchKernel(Get, dim3(1,1,1), dim3(LEN,1,1), 0, 0, Ad);
    HIP_ASSERT(hipMemcpy(B, Ad, SIZE, hipMemcpyDeviceToHost));
    for(unsigned i=0;i<LEN;i++)
    {
        assert(A[i] == B[i]);
    }
    std::cout<<"Passed"<<std::endl;
}
```

4.5.2 CU_POINTER_ATTRIBUTE_MEMORY_TYPE

To get pointer's memory type in HIP/HIP-Clang one should use hipPointerGetAttributes API. The first parameter of the API is hipPointerAttribute_t which has 'memoryType' as a member variable. 'memoryType' indicates the input pointer is allocated on device or host.

For example:

```
double * ptr;
hipMalloc(reinterpret_cast<void**>(&ptr), sizeof(double));
hipPointerAttribute_t attr;
hipPointerGetAttributes(&attr, ptr); /*attr.memoryType will have value as
hipMemoryTypeDevice*/

double* ptrHost;
hipHostMalloc(&ptrHost, sizeof(double));
hipPointerAttribute_t attr;
hipPointerGetAttributes(&attr, ptrHost); /*attr.memoryType will have value as
hipMemoryTypeHost*/
```

4.5.3 threadfence_system

Threadfence_system makes all device memory writes, all writes to mapped host memory, and all writes to peer memory visible to CPU and other GPU devices. Some implementations can provide this behavior by flushing the GPU L2 cache. HIP/HIP-Clang does not provide this functionality. As a workaround, users can set the environment variable HSA_DISABLE_CACHE=1 to disable the GPU L2 cache. This will affect all accesses and for all kernels and so may have a performance impact.

4.5.4 Textures and Cache Control

Compute programs sometimes use textures either to access dedicated texture caches or to use the texture-sampling hardware for interpolation and clamping. The former approach uses simple point samplers with linear interpolation, essentially only reading a single point. The latter approach uses the sampler hardware to interpolate and combine multiple samples. AMD hardware, as well as recent competing hardware, has a unified texture/L1 cache, so it no longer has a dedicated texture cache. But the nvcc path often caches global loads in the L2 cache, and some programs may benefit from explicit control of the L1 cache contents. We recommend the __ldg instruction for this purpose.

AMD compilers currently load all data into both the L1 and L2 caches, so __ldg is treated as a no-op.

We recommend the following for functional portability:

- For programs that use textures only to benefit from improved caching, use the *__ldg* instruction
- Programs that use texture object and reference APIs work well on HIP

4.6 More Tips

4.6.1 HIP Logging

On an AMD platform, set the AMD_LOG_LEVEL environment variable to log HIP application execution information.

Refer to the section on HIP Logging in this document for more information.

4.6.2 Debugging hipcc

To see the detailed commands that hipcc issues, set the environment variable HIPCC_VERBOSE to 1. Doing so will print to stderr the HIP-clang (or nvcc) commands that hipcc generates.

4.6.3 Editor Highlighting

See the utils/vim or utils/gedit directories to add handy highlighting to hip files.

4.7 HIP Porting Driver API

4.7.1 Porting CUDA Driver API

CUDA provides a separate CUDA Driver and Runtime APIs. The two APIs have significant overlap in functionality:

- Both APIs support events, streams, memory management, memory copy, and error handling.
- Both APIs deliver similar performance.
- Driver APIs calls begin with the prefix cu while Runtime APIs begin with the prefix cuda. For example, the Driver API API contains cuEventCreate while the Runtime API contains cudaEventCreate, with similar functionality.
- The Driver API defines a different but largely overlapping error code space than the Runtime API uses a different coding convention. For example, Driver API defines CUDA_ERROR_INVALID_VALUE while the Runtime API defines cudaErrorInvalidValue

NOTE: The Driver API offers two additional pieces of functionality not provided by the Runtime API: cuModule and cuCtx APIs.

4.7.2 cuModule API

The Module section of the Driver API provides additional control over how and when accelerator code objects are loaded. For example, the driver API allows code objects to be loaded from files or memory pointers. Symbols for kernels or global data can be extracted from the loaded code objects. In contrast, the Runtime API automatically loads and (if necessary) compiles all of the kernels from an executable binary when run. In this mode, NVCC must be used to compile kernel code so the automatic loading can function correctly.

Both Driver and Runtime APIs define a function for launching kernels (called cuLaunchKernel or cudaLaunchKernel. The kernel arguments and the execution configuration (grid dimensions, group dimensions, dynamic shared memory, and stream) are passed as arguments to the launch function. The Runtime additionally provides the <<< >>> syntax for launching kernels, which resembles a special function call and is easier to use than explicit launch API (in particular the handling of kernel arguments). However, this syntax is not standard C++ and is available only when NVCC is used to compile the host code.

The Module features are useful in an environment that generates the code objects directly, such as a new accelerator language front-end. Here, NVCC is not used. Instead, the environment may have a different kernel language or a different compilation flow. Other environments have many kernels and do not want them to be all loaded automatically. The Module functions can be used to load the generated code objects and launch kernels. As we will see below, HIP defines a Module API which provides similar explicit control over code object management.

4.7.3 cuCtx API

The Driver API defines "Context" and "Devices" as separate entities. Contexts contain a single device, and a device can theoretically have multiple contexts. Each context contains a set of streams and events specific to the context. Historically contexts also defined a unique address space for the GPU, though this may no longer be the case in Unified Memory platforms (since the CPU and all the devices in the same process share a single unified address space). The Context APIs also provide a mechanism to switch between devices, which allowed a single CPU thread to send commands to different GPUs. HIP as well as a recent version of CUDA Runtime provide other mechanisms to accomplish this feat - for example using streams or cudaSetDevice.

The CUDA Runtime API unifies the Context API with the Device API. This simplifies the APIs and has little loss of functionality since each Context can contain a single device, and the benefits of multiple contexts have been replaced with other interfaces. HIP provides a context API to facilitate easy porting from existing Driver codes. In HIP, the Ctx functions largely provide an alternate syntax for changing the active device. Most new applications will prefer to use hipSetDevice or the stream APIs , therefore HIP has marked hipCtx APIs as deprecated. Support for these APIs may not be available in future releases. For more details on deprecated APIs, refer to HIP deprecated APIs at:

https://github.com/ROCm-Developer-Tools/HIP/blob/main/docs/markdown/hip_deprecated_api_list.md

4.7.4 HIP Module and Ctx APIs

Rather than present two separate APIs, HIP extends the HIP API with new APIs for Modules and Ctx control.

4.7.4.1 hipModule API

Like the CUDA Driver API, the Module API provides additional control over how code is loaded, including options to load code from files or in-memory pointers. NVCC and HIP-Clang target different architectures and use different code object formats: NVCC is `cubin` or `ptx` files, while the HIP-Clang path is the `hsaco` format. The external compilers which generate these code objects are responsible for generating and loading the correct code object for each platform. Notably, there is no fat binary format that can contain code for both NVCC and HIP-Clang platforms. The following table summarizes the formats used on each platform:

Format	APIs	NVCC	HIP-CLANG
Code Object	hipModuleLoad, hipModuleLoadData	.cubin or PTX text	.hsaco
Fat Binary	hipModuleLoadFatBin	.fatbin	.hip_fatbin

`hipcc` uses HIP-Clang or NVCC to compile host codes. Both may embed code objects into the final executable, and these code objects will be automatically loaded when the application starts. The hipModule API can be used to load additional code objects, and in this way provides an extended capability to the automatically loaded code objects. HIP-Clang allows both capabilities to be used together if desired. It is possible to create a program with no kernels and thus no automatic loading.

4.7.5 hipCtx API

HIP provides a Ctx API as a thin layer over the existing Device functions. This Ctx API can be used to set the current context or to query properties of the device associated with the context. The current context is implicitly used by other APIs such as *hipStreamCreate*.

4.7.6 hipify translation of CUDA Driver API

The HIPIFY tools convert CUDA Driver APIs for streams, events, modules, devices, memory management, context, profiler to the equivalent HIP driver calls. For example, cuEventCreate will be translated into hipEventCreate. HIPIFY tools also convert error codes from the Driver namespace and coding convention to the equivalent HIP error code. Thus, HIP unifies the APIs for these common functions. The memory copy API requires additional explanation. The CUDA driver includes the memory direction in the name of the API (ie cuMemcpyH2D) while the CUDA driver API provides a single memory copy API with a parameter that specifies the direction and additionally supports a "default" direction where the runtime determines the direction automatically. HIP provides APIs with both styles: for example, hipMemcpyH2D as well as hipMemcpy. The first flavor may be faster in some cases since they avoid host overhead to detect different memory directions.

HIP defines a single error space and uses camel-case for all errors (i.e. hipErrorInvalidValue)

4.8 HIP-Clang Implementation Notes

4.8.1 .hip_fatbin

hip-clang links device code from different translation units together. For each device target, a code object is generated. Code objects for different device targets are bundled by clang-offload-bundler as one fatbinary, which is embedded as a global symbol __hip_fatbin in the .hip_fatbin section of the ELF file of the executable or shared object.

4.8.2 Initialization and Termination Functions

HIP-Clang generates initialization and termination functions for each translation unit for the host code compilation. The initialization functions call *__hipRegisterFatBinary* to register the fatbinary embedded in the ELF file. They also call *__hipRegisterFunction* and *__hipRegisterVar* to register kernel functions and device-side global variables. The termination functions call *__hipUnregisterFatBinar*y. HIP-Clang emits a global variable *__hip_gpubin_handle* of void** type with linkonce linkage and initial value 0 for each host translation unit. Each initialization function checks *__hip_gpubin_handle* and register the fatbinary only if *__hip_gpubin_handle* is 0 and saves the return value of *__hip_gpubin_handle* to *__hip_gpubin_handle*. This is to guarantee that the fatbinary is only registered once. A similar check is done in the termination functions.

4.8.3 Kernel Launching

HIP-Clang supports kernel launching by CUDA <<<>>> syntax, hipLaunchKernel, and hipLaunchKernelGGL. The latter two are macros that expand to CUDA <<<>>> syntax.

When the executable or shared library is loaded by the dynamic linker, the initialization functions are called. In the initialization functions, when __hipRegisterFatBinary is called, the code objects containing all kernels are loaded; when __hipRegisterFunction is called, the stub functions are associated with the corresponding kernels in code objects. HIP-Clang implements two sets of kernels launching APIs.

By default, in the host code, for the <<<>>> statement, hip-clang first emits call of hipConfigureCall to set up the threads and grids, then emits call of the stub function with the given arguments. In the stub function, hipSetupArgument is called for each kernel argument, then hipLaunchByPtr is called with a function pointer to the stub function. In *hipLaunchByPtr*, the real kernel associated with the stub function is launched.

If HIP program is compiled with -fhip-new-launch-api, in the host code, for the <<<>>> statement, hip-clang first emits call of __hipPushCallConfiguration to save the grid dimension, block dimension, shared memory usage and stream to a stack, then emits call of the stub function with the given arguments. In the stub function, __hipPopCallConfiguration is called to get the saved grid dimension, block dimension, shared memory usage and stream, then hipLaunchKernel is called with a function pointer to the stub function. In hipLaunchKernel, the real kernel associated with the stub function is launched.

4.8.4 Address Spaces

HIP-Clang defines a process-wide address space where the CPU and all devices allocate addresses from a single unified pool. Thus, addresses may be shared between contexts, and unlike the original CUDA definition, a new context does not create a new address space for the device.

4.8.5 Using hipModuleLaunchKernel

`hipModuleLaunchKernel` is `cuLaunchKernel` in HIP world. It takes the same arguments as `cuLaunchKernel`.

4.8.6 Additional Information

HIP-Clang creates a primary context when the HIP API is called. In a pure driver API code, HIP-Clang will create a primary context while HIP/NVCC will have an empty context stack. HIP-Clang will push the primary context to the context stack when it is empty. This can have subtle differences in applications that mix the runtime and driver APIs.

4.9 NVCC Implementation Notes

4.9.1 Interoperation between HIP and CUDA Driver

CUDA applications may want to mix CUDA driver code with HIP code. This table shows the type equivalence to enable this interaction.

HIP Type	CU Driver Type	CUDA Runtime Type
hipModule_t	CUmodule	
hipFunction_t	CUfunction	
hipCtx_t	CUcontext	
hipDevice_t	CUdevice	
hipStream_t	CUstream	cudaStream_t
hipEvent_t	CUevent	cudaEvent_t
hipArray	CUarray	cudaArray

4.9.2 Compilation Options

The hipModule_t interface does not support cuModuleLoadDataEx function, which is used to control PTX compilation options. HIP-Clang does not use PTX and does not support these compilation options. HIP-Clang code objects always contain fully compiled ISA and do not require additional compilation as a part of the load step.

The corresponding HIP function `hipModuleLoadDataEx` behaves as `hipModuleLoadData` on HIP-Clang path (compilation options are not used) and as `cuModuleLoadDataEx` on NVCC path.

For example,

CUDA

```
CUmodule module;
void *imagePtr = ...;  // Somehow populate data pointer with code object
const int numOptions = 1;
CUJit_option options[numOptions];
void * optionValues[numOptions];
options[0] = CU_JIT_MAX_REGISTERS;
unsigned maxRegs = 15;
optionValues[0] = (void*)(&maxRegs);

cuModuleLoadDataEx(module, imagePtr, numOptions, options, optionValues);
CUfunction k;
cuModuleGetFunction(&k, module, "myKernel");
```

HIP

```
hipModule_t module;
void *imagePtr = ...;  // Somehow populate data pointer with code object
const int numOptions = 1;
hipJitOption options[numOptions];
void * optionValues[numOptions];
options[0] = hipJitOptionMaxRegisters;
unsigned maxRegs = 15;
optionValues[0] = (void*)(&maxRegs);
// hipModuleLoadData(module, imagePtr) will be called on HIP-Clang path, JIT options
will not be used, and
// cupModuleLoadDataEx(module, imagePtr, numOptions, options, optionValues) will be
called on NVCC path
hipModuleLoadDataEx(module, imagePtr, numOptions, options, optionValues);
hipFunction_t k;
hipModuleGetFunction(&k, module, "myKernel");
```

The sample below shows how to use hipModuleGetFunction:

```
#include<hip_runtime.h>
#include<hip_runtime_api.h>
#include<iostream>
#include<fstream>
#include<vector>
#define LEN 64
#define SIZE LEN<<2
#ifdef __HIP_PLATFORM_HCC__
#define fileName "vcpy_isa.co"
#endif
#ifdef __HIP_PLATFORM_NVCC__
#define fileName "vcpy_isa.ptx"
#endif
#define kernel_name "hello_world"
int main(){
    float *A, *B;
    hipDeviceptr_t Ad, Bd;
    A = new float[LEN];
    B = new float[LEN];
    for(uint32_t i=0;i<LEN;i++){
        A[i] = i*1.0f;
        B[i] = 0.0f;
        std::cout<<A[i] << " "<<B[i]<<std::endl;
    }

#ifdef __HIP_PLATFORM_NVCC__
          hipInit(0);
          hipDevice_t device;
          hipCtx_t context;
          hipDeviceGet(&device, 0);
          hipCtxCreate(&context, 0, device);
#endif
    hipMalloc((void**)&Ad, SIZE);
    hipMalloc((void**)&Bd, SIZE);
    hipMemcpyHtoD(Ad, A, SIZE);
    hipMemcpyHtoD(Bd, B, SIZE);
    hipModule_t Module;
    hipFunction_t Function;
    hipModuleLoad(&Module, fileName);
    hipModuleGetFunction(&Function, Module, kernel_name);
    std::vector<void*>argBuffer(2);
    memcpy(&argBuffer[0], &Ad, sizeof(void*));
    memcpy(&argBuffer[1], &Bd, sizeof(void*));
    size_t size = argBuffer.size()*sizeof(void*);
    void *config[] = {
      HIP_LAUNCH_PARAM_BUFFER_POINTER, &argBuffer[0],
      HIP_LAUNCH_PARAM_BUFFER_SIZE, &size,
      HIP_LAUNCH_PARAM_END
    };
    hipModuleLaunchKernel(Function, 1, 1, 1, LEN, 1, 1, 0, 0, NULL, (void**)&config);
    hipMemcpyDtoH(B, Bd, SIZE);
    for(uint32_t i=0;i<LEN;i++){
        std::cout<<A[i]<<" - "<<B[i]<<std::endl;
    }

#ifdef __HIP_PLATFORM_NVCC__
          hipCtxDetach(context);
#endif

    return 0;
}
```

4.9.3 HIP Module and Texture Driver API

HIP supports texture driver APIs however texture reference should be declared in host scope. The following code explains the use of texture reference for the HIP_PLATFORM_HCC platform.

```
// Code to generate code object
#include "hip/hip_runtime.h"
extern texture<float, 2, hipReadModeElementType> tex;
__global__ void tex2dKernel(hipLaunchParm lp, float* outputData,
                             int width,
                             int height)
{
int x = blockIdx.x*blockDim.x + threadIdx.x;
int y = blockIdx.y*blockDim.y + threadIdx.y;
outputData[y*width + x] = tex2D(tex, x, y);
}

// Host code:
texture<float, 2, hipReadModeElementType> tex;
```

```
void myFunc ()
{
    // ...
    textureReference* texref;
    hipModuleGetTexRef(&texref, Module1, "tex");
    hipTexRefSetAddressMode(texref, 0, hipAddressModeWrap);
    hipTexRefSetAddressMode(texref, 1, hipAddressModeWrap);
    hipTexRefSetFilterMode(texref, hipFilterModePoint);
    hipTexRefSetFlags(texref, 0);
    hipTexRefSetFormat(texref, HIP_AD_FORMAT_FLOAT, 1);
    hipTexRefSetArray(texref, array, HIP_TRSA_OVERRIDE_FORMAT);
   // ...
}
```

Chapter 5 Appendix A – HIP API

The following appendices are available on the AMD ROCm GitHub documentation website.

5.1 HIP API Guide

You can access the Doxygen-generated HIP API Guide at the following location:

https://github.com/RadeonOpenCompute/ROCm/blob/master/AMD_HIP_API_Guide_v4.3.pdf

5.2 HIP-Supported CUDA API Reference Guide

The HIP-Supported CUDA API Reference Guide consists of CUDA APIs supported in HIP and covers the Driver API, Runtime API, cuComplex API, Device API, and APIs for the following supported libraries:

- cuBLAS
- cuRAND
- cuFFT
- cuSPARSE
- cuDNN

For more information, see

https://github.com/RadeonOpenCompute/ROCm/blob/master/AMD_HIP_Supported_CUDA_API_Reference_Guide_v4.3.pdf

5.3 Deprecated HIP APIs

5.3.1 HIP Context Management APIs

CUDA supports cuCtx API, the Driver API that defines "Context" and "Devices" as separate entities. Contexts contain a single device, and a device can theoretically have multiple contexts. HIP initially added limited support for APIs to facilitate easy porting from existing driver codes. The APIs are marked as deprecated now as there is a better alternate interface (such as hipSetDevice or the stream API) to achieve the required functions.

- hipCtxPopCurrent
- hipCtxPushCurrent
- hipCtxSetCurrent
- hipCtxGetCurrent
- hipCtxGetDevice
- hipCtxGetApiVersion

- hipCtxGetCacheConfig
- hipCtxSetCacheConfig
- hipCtxSetSharedMemConfig
- hipCtxGetSharedMemConfig
- hipCtxSynchronize
- hipCtxGetFlags
- hipCtxEnablePeerAccess
- hipCtxDisablePeerAccess

5.3.2 HIP Memory Management APIs

5.3.2.1 hipMallocHost

Use "hipHostMalloc" instead.

5.3.2.2 hipMemAllocHost

Use "hipHostMalloc" instead.

5.3.2.3 hipHostAlloc

Use "hipHostMalloc" instead.

5.3.2.4 hipFreeHost

Use "hipHostFree" instead.

5.4 Supported HIP Math APIs

You can access the supported HIP Math APIs at:

https://github.com/ROCm-Developer-Tools/HIP/blob/main/docs/markdown/hip-math-api.md

Chapter 6 Appendix C

6.1 HIP FAQ

You can access the HIP FAQ at:

https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-FAQ.html#hip-faq