DISCLAIMER

The information contained herein is for informational purposes only and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. All open source software listed in this presentation is governed by the associated open source license. Terms and limitations applicable to the purchase or use of AMD’s products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale.

©2019 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, Ryzen, Threadripper, EPYC, Infinity Fabric, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

PCIe is a trademark (or registered trademark) of PCI-SIG Corporation.

OpenCL is a trademark of Apple Inc. used by permission by Khronos Group, Inc.

Linux is a trademark (or registered trademark) of Linus Torvalds.
Agenda

- Introduction (5 minutes)
- AMD GPU Hardware (10 minutes)
- GPU Programming Concepts (45 minutes)
- GPU Programming Software (15 minutes)
- Porting existing CUDA codes to HIP (15 mins)
What we won’t cover today (but is still important)

- Profiling:
  - `rcprof`: command line profiler
  - `rocprofiler` / `roctracer`: libraries for collecting GPU hardware counters and application traces
  - `CodeXL`: visualize output of rcprof
  - Install: `sudo apt install rocprofiler-dev roctracer-dev rocm-profiler cxlactivitylogger`

- Debugging:
  - `rocr_debug_agent`: print state of wavefronts on memory violation / signals
  - HIP debugging tips
  - `in kernel printf`
What we won’t cover today (but is still important)

- **AOMP** (AMD OpenMP Compiler):
  - OpenMP 4.5+ support, “target” pragmas, device offloading

- **GPU Libraries:**
  - **hipBLAS**: BLAS functionality on GPUs
  - **rocFFT**: FFTs on GPUs
  - **rocRAND**: random number generation
  - **rocPRIM** / **hipCUB**: high performance GPU primitives
  - **Tensile**: GEMMs, tensor contractions
  - **hipSPARSE**: BLAS for sparse matrices / vectors
  - **rocALUTION**: iterative sparse solvers
  - **MIOpen, TensorFlow, PyTorch**: machine learning
Comments

- Slides will be provided
- Ask questions in the google doc; we will be monitoring it
- Focus is on single node / device:
  - Little discussion of MPI, or multi-node

Please share any feedback or ask questions in the Google Doc
AMD GCN GPU Hardware

René van Oostrum <rene.vanoostrum@amd.com>
(based on a presentation by Joe Greathouse, RTG, AMD)
# AMD GCN GPU HARDWARE

## AGENDA

<table>
<thead>
<tr>
<th>Introduction</th>
<th>GCN Hardware Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD GPU Compute Terminology</td>
<td></td>
</tr>
<tr>
<td>AMD GPU Architecture</td>
<td>GPU Memory and I/O System</td>
</tr>
<tr>
<td>GCN Compute Unit Internals</td>
<td></td>
</tr>
</tbody>
</table>
AMD GCN Hardware Overview
AMD GCN GPU Hardware Layout

Command Processor

 Queues reside in user-visible DRAM

Command Queue
## Hardware Configuration Parameters on Modern AMD GPUs

<table>
<thead>
<tr>
<th>GPU SKU</th>
<th>Shader Engines</th>
<th>CUs / SE</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Radeon Instinct™ MI60</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>AMD Radeon Instinct™ MI50</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>AMD Radeon™ VII</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>AMD Radeon Instinct™ MI25</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>AMD Radeon™ Vega 64</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>AMD Radeon™ Vega 56</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>AMD Radeon Instinct™ MI6</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>AMD Ryzen™ 5 2400G</td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>
AMD GPU Compute Terminology
Overview of GPU Kernels

GPU Kernel

Functions launched to the GPU that are executed by multiple parallel workers

Examples: GEMM, triangular solve, vector copy, scan, convolution
Overview of GPU Kernels

<table>
<thead>
<tr>
<th>Workgroup 0</th>
<th>Group of threads that are on the GPU at the same time. Also on the same compute unit. Can synchronize together and communicate through memory in the CU.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workgroup 1</td>
<td></td>
</tr>
<tr>
<td>Workgroup 2</td>
<td></td>
</tr>
<tr>
<td>Workgroup 3</td>
<td></td>
</tr>
<tr>
<td>Workgroup 4</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Workgroup n</td>
<td>Programmer controls the number of workgroups – it’s usually a function of problem size.</td>
</tr>
</tbody>
</table>

CUDA Terminology

Thread Block
## Overview of GPU Kernels

### GPU Kernel

<table>
<thead>
<tr>
<th>Workgroup 0</th>
<th>Wavefront 0</th>
<th>Collection of resources that execute in lockstep, run the same instructions, and follow the same control-flow path. Individual lanes can be masked off. Can think of this as a vectorized thread.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workgroup 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Workgroup 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Workgroup 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Workgroup 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Workgroup n</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CUDA Terminology**

- **Warp**
Overview of GPU Kernels

<table>
<thead>
<tr>
<th>Workgroup 0</th>
<th>Workgroup 1</th>
<th>Workgroup 2</th>
<th>Workgroup 3</th>
<th>Workgroup 4</th>
<th>...</th>
<th>Workgroup n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wavefront 0</td>
<td>Wavefront 1</td>
<td>...</td>
<td>Wavefront 15</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64 work items (threads)

Number of wavefronts / workgroup is chosen by developer. GCN hardware allows up to 16 wavefronts in a workgroup.
Scheduling work to a GPU

Command Processor

Command Queue

Workload Manager
GPU Memory and I/O
GPU Memory, I/O, and Connectivity

- Command Queue
- Command Processor
- DMA Engines
- PCIe® Controllers
- Memory Controllers
- HBM/GDDR Memory
- DMA Engines
- Infinity Fabric Controllers
- Other GPUs
- System Memory
- Other GPUs
DMA Engines Accept Work from the Same Queues

Step 1
CPU submits a DMA Transfer packet to the command queue

This is done with user-level memory writes in Radeon Open Compute (ROCm). No kernel drivers involved.
DMA Engines Accept Work from the Same Queues

- Command Queue
- Command Queue

Step 2
CP Reads the packet and understands the transfer request

GPU

System Memory

DMA Engines

PCle® Controllers

Memory Controllers

HBM/GDDR Memory

Other GPUs
DMA Engines Accept Work from the Same Queues

Step 3
CP sends transfer command to the target DMA Engine

This can take place in parallel with other compute work & transfers
DMA Engines Accept Work from the Same Queues

Step 4
DMA Engines Transfer Data to/from GPU Memory

Transfer takes place within process’s virtual memory space

System Memory

DMA Engines

Pcie® Controllers

Memory Controllers

HBM/GDDR Memory

Other GPUs

DMA Engines

Infinity Fabric Controllers

Command Processor

Command Queue

Command Queue
GCN Compute Unit Internals
GCN Compute Unit

- Scalar Unit
  - Shared by all threads of a wavefront
  - Used for flow control, pointer arithmetic, etc.
  - Has own GPR pool, scalar data cache, etc.
- 8KB Scalar General Purpose Registers (sGPR)
4x Vector Units (16-lane SIMD)

CU Total Throughput: 64 Single-Precision (SP) ops/clock

4x64KB Vector Registers (vGPR)
Each Compute Unit (CU) contains 4 SIMD; each SIMD has:

- A 16-lane IEEE-754 vector ALU (vALU)
- 64KB of vector register file (vGPR)
- 256 total registers – each register is 64 4-byte-wide entries
- Instruction buffer for 10 wavefronts
GCN Compute Unit

64kb Local Data Share (LDS) – Scratchpad Memory

32 banks, with conflict resolution

Can be used to shared data between all threads within a workgroup
## Nvidia/CUDA Terminology

<table>
<thead>
<tr>
<th>Nvidia/CUDA Terminology</th>
<th>AMD Terminology</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streaming Multiprocessor</td>
<td>Compute Unit (CU)</td>
<td>One of many parallel vector processors in a GPU that contain parallel ALUs. All waves in a workgroups are assigned to the same CU.</td>
</tr>
<tr>
<td>Kernel</td>
<td>Kernel</td>
<td>Functions launched to the GPU that are executed by multiple parallel workers on the GPU. Kernels can work in parallel with CPU.</td>
</tr>
<tr>
<td>Warp</td>
<td>Wavefront</td>
<td>Collection of operations that execute in lockstep, run the same instructions, and follow the same control-flow path. Individual lanes can be masked off. Think of this as a vector thread. A 64-wide wavefront is a 64-wide vector op.</td>
</tr>
<tr>
<td>Thread block</td>
<td>Workgroup</td>
<td>Group of wavefronts that are on the GPU at the same time. Can synchronize together and communicate through local memory.</td>
</tr>
<tr>
<td>Thread</td>
<td>Work item / Thread</td>
<td>Individual lane in a wavefront. On AMD GPUs, must run in lockstep with other work items in the wavefront. Lanes can be individually masked off. GPU programming models can treat this as a separate thread of execution, though you do not necessarily get forward sub-wavefront progress.</td>
</tr>
</tbody>
</table>
# Software Terminology

<table>
<thead>
<tr>
<th>Nvidia/CUDA Terminology</th>
<th>AMD Terminology</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Memory</td>
<td>Global Memory</td>
<td>DRAM memory accessible by the GPU that goes through some layers cache</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Local memory</td>
<td>Scratchpad that allows communication between wavefronts in a workgroup.</td>
</tr>
<tr>
<td>Local memory</td>
<td>Private memory</td>
<td>Per-thread private memory, often mapped to registers.</td>
</tr>
</tbody>
</table>
AMD GPU Programming Concepts

Programming with HIP: Kernels, blocks, threads, and more

Noel Chalmers <noel.chalmers@amd.com>
What is HIP?

AMD’s Heterogeneous-compute Interface for Portability, or HIP, is a C++ runtime API and kernel language that allows developers to create portable applications that can run on AMD’s accelerators as well as CUDA devices.

HIP:
- Is open-source.
- Provides an API for an application to leverage GPU acceleration for both AMD and CUDA devices.
- Syntactically similar to CUDA. Most CUDA API calls can be converted in place: cuda -> hip
- Supports a strong subset of CUDA runtime functionality.
A Tale of Host and Device

Source code in HIP has two flavors: Host code and Device code

- The Host is the CPU
- Host code runs here
- Usual C++ syntax and features
- Entry point is the ‘main’ function
- HIP API can be used to create device buffers, move between host and device, and launch device code.

- The Device is the GPU
- Device code runs here
- C-like syntax
- Device codes are launched via “kernels”
- Instructions from the Host are enqueued into “streams”
HIP API

- Device Management:
  - `hipSetDevice()`, `hipGetDevice()`, `hipGetDeviceProperties()`
- Memory Management
  - `hipMalloc()`, `hipMemcpy()`, `hipMemcpyAsync()`, `hipFree()`
- Streams
  - `hipStreamCreate()`, `hipSynchronize()`, `hipStreamSynchronize()`, `hipStreamFree()`
- Events
  - `hipEventCreate()`, `hipEventRecord()`, `hipStreamWaitEvent()`, `hipEventElapsedTime()`
- Device Kernels
  - `__global__`, `__device__`, `hipLaunchKernelGGL()`
- Device code
  - `threadIdx`, `blockIdx`, ` blockDim`, `__shared__`
  - 200+ math functions covering entire CUDA math library.
- Error handling
  - `hipGetLastError()`, `hipGetErrorString()`
Kernels, memory, and structure of host code
Device Kernels: The Grid

- In HIP, kernels are executed on a 3D "grid"
  - You might feel comfortable thinking in terms of a mesh of points, but it’s not required
- The “grid” is what you can map your problem to
  - It’s not a physical thing, but it can be useful to think that way
- AMD devices (GPUs) support 1D, 2D, and 3D grids, but most work maps well to 1D
- Each dimension of the grid partitioned into equal sized “blocks”
- Each block is made up of multiple “threads”
- The grid and its associated blocks are just organizational constructs
  - The threads are the things that do the work
- If you’re familiar with CUDA already, the grid+block structure is very similar in HIP
Some Terminology:

<table>
<thead>
<tr>
<th>CUDA</th>
<th>HIP</th>
<th>OpenCL™</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid</td>
<td>grid</td>
<td>NDRange</td>
</tr>
<tr>
<td>block</td>
<td>block</td>
<td>work group</td>
</tr>
<tr>
<td>thread</td>
<td>thread</td>
<td>work item</td>
</tr>
<tr>
<td>warp</td>
<td>wavefront</td>
<td>sub-group (?)</td>
</tr>
</tbody>
</table>
Threads in grid have access to:

- Their respective block: \texttt{blockIdx.x}
- Their respective thread ID in a block: \texttt{threadIdx.x}
- Their block's dimension: \texttt{blockDim.x}
- The number of blocks in the grid: \texttt{gridDim.x}
The Grid: blocks of threads in 2D

- Each color is a block of threads
- Each small square is a thread
- The concept is the same in 1D and 2D
- In 2D each block and thread now has a two-dimensional index

Threads in grid have access to:
- Their respective block IDs: \( \text{blockIdx.x} \), \( \text{blockIdx.y} \)
- Their respective thread IDs in a block: \( \text{threadIdx.x} \), \( \text{threadIdx.y} \)
- Etc.
Kernels
A simple embarrassingly parallel loop

for (int i=0;i<N;i++) {
    h_a[i] *= 2.0;
}

Can be translated into a GPU kernel:

__global__ void myKernel(int N, double *d_a) {
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<N) {
        d_a[i] *= 2.0;
    }
}

- A device function that will be launched from the host program is called a kernel and is declared with the __global__ attribute
- Kernels should be declared void
- All pointers passed to kernels must point to memory on the device (more later)
- All threads execute the kernel’s body “simultaneously”
- Each thread uses its unique thread and block IDs to compute a global ID
- There could be more than N threads in the grid (we’ll see why in a minute)
Kernels

Kernels are launched from the host:

```c
dim3 threads(256,1,1);  //3D dimensions of a block of threads
dim3 blocks((N+256-1)/256,1,1);  //3D dimensions the grid of blocks

hipLaunchKernelGGL(myKernel,  //Kernel name (__global__ void function)
    blocks,  //Grid dimensions
    threads,  //Block dimensions
    0,  //Bytes of dynamic LDS space (ignore for now)
    0,  //Stream (0=NULL stream)
    N, a);  //Kernel arguments
```

Analogous to CUDA kernel launch syntax:

```c
myKernel<<blocks, threads, 0, 0>>(N,a);
```
SIMD operations

Why blocks and threads?

Natural mapping of kernels to hardware:

▪ Blocks are dynamically scheduled onto CUs
▪ All threads in a block execute on the same CU
▪ Threads in a block share LDS memory and L1 cache
▪ Threads in a block are executed in **64-wide** chunks called “wavefronts”
▪ Wavefronts execute on SIMD units (Single Instruction Multiple Data)
▪ If a wavefront stalls (e.g. data dependency) CUs can quickly context switch to another wavefront

A good practice is to make the block size a multiple of 64 and have several wavefronts (e.g. 256 threads)
Device Memory

The host instructs the device to allocate memory in VRAM and records a pointer to device memory:

```c
int main() {
    ...
    int N = 1000;
    size_t Nbytes = N*sizeof(double);
    double *h_a = (double*) malloc(Nbytes); //Host memory

    double *d_a = NULL;
    hipMalloc(&d_a, Nbytes); //Allocate Nbytes on device

    ...

    free(h_a); //free host memory
    hipFree(d_a); //free device memory
}
```
Device Memory

The host queues memory transfers:

//copy data from host to device

hipMemcpy(d_a, h_a, Nbytes, hipMemcpyHostToDevice);

//copy data from device to host

hipMemcpy(h_a, d_a, Nbytes, hipMemcpyDeviceToHost);

//copy data from one device buffer to another

hipMemcpy(d_b, d_a, Nbytes, hipMemcpyDeviceToDevice);
Device Memory

Can copy strided sections of arrays:

```c
hipMemcpy2D(d_a, //pointer to destination
    DLDAbytes, //pitch of destination array
    h_a, //pointer to source
    LDAbytes, //pitch of source array
    Nbytes, //number of bytes in each row
    Nrows, //number of rows to copy
    hipMemcpyHostToDevice);
```
Error Checking

- Most HIP API functions return error codes of type `hipError_t`
  
  ```
  hipError_t status1 = hipMalloc(...);
  hipError_t status2 = hipMemcpy(...);
  ```

- If API function was error-free, returns `hipSuccess`, otherwise returns an error code.

- Can also peek/get at last error returned with
  
  ```
  hipError_t status3 = hipGetLastError();
  hipError_t status4 = hipPeakLastError();
  ```

- Can get a corresponding error string using `hipGetErrorString(status)`. Helpful for debugging, e.g.

```
#define HIP_CHECK(command) { \
    hipError_t status = command; \
    if (status!=hipSuccess) { \
        std::cerr << "Error: HIP reports " << hipGetErrorString(status) << std::endl; \n        std::abort(); } }
```

```
Putting it all together

#include "hip/hip_runtime.h"

int main() {
    int N = 1000;
    size_t Nbytes = N*sizeof(double);
    double *h_a = (double*) malloc(Nbytes); //host memory
    double *d_a = NULL;
    HIP_CHECK(hipMalloc(&d_a, Nbytes));
    ...
    HIP_CHECK(hipMemcpy(d_a, h_a, Nbytes, hipMemcpyHostToDevice)); //copy data to device

    hipLaunchKernelGGL(myKernel, dim3((N+256-1)/256,1,1), dim3(256,1,1), 0, 0, N, d_a); //Launch kernel
    HIP_CHECK(hipGetLastError());

    HIP_CHECK(hipMemcpy(h_a, d_a, Nbytes, hipMemcpyDeviceToHost));
    ...
    free(h_a); //free host memory
    HIP_CHECK(hipFree(d_a)); //free device memory
}

__global__ void myKernel(int N, double *d_a) {
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<N) {
        d_a[i] *= 2.0;
    }
}

#define HIP_CHECK(command) {
    hipError_t status = command;
    if (status!=hipSuccess) {
        std::cerr << "Error: HIP reports "
        << hipGetErrorString(status) << std::endl;
        std::abort();
    }
}
Device management and asynchronous computing
Device Management

Multiple GPUs in system? Multiple host threads/MPI ranks? What device are we running on?

- Host can query number of devices visible to system:
  ```
  int numDevices = 0;
  hipGetDeviceCount(&numDevices);
  ```

- Host tells the runtime to issue instructions to a particular device:
  ```
  int deviceID = 0;
  hipSetDevice(deviceID);
  ```

- Host can query what device is currently selected:
  ```
  hipGetDevice(&deviceID);
  ```

- The host can manage several devices by swapping the currently selected device during runtime.
- MPI ranks can set different devices or over-subscribe (share) devices.
Device Properties

The host can also query a device’s properties:

```c
hipDeviceProp_t  props;
hipGetDeviceProperties(&props, deviceID);
```

- `hipDeviceProp_t` is a struct that contains useful fields like the device’s name, total VRAM, clock speed, and GCN architecture.
  - See “hip/hip_runtime_api.h” for full list of fields.
Blocking vs Nonblocking API functions

- The kernel launch function, `hipLaunchKernelGGL`, is **non-blocking** for the host.
  - After sending instructions/data, the host continues immediately while the device executes the kernel
  - If you know the kernel will take some time, this is a good area to do some work (i.e. MPI comms) on the host

- However, `hipMemcpy` is **blocking**.
  - The data pointed to in the arguments can be accessed/modified after the function returns.

- The non-blocking version is `hipMemcpyAsync`

  ```
  hipMemcpyAsync(d_a, h_a, Nbytes, hipMemcpyHostToDevice, stream);
  ```

- Like `hipLaunchKernelGGL`, this function takes an argument of type `hipStream_t`

- It is not safe to access/modify the arguments of `hipMemcpyAsync` without some sort of synchronization.
Putting it all together

```c
#include "hip/hip_runtime.h"

int main() {
    int N = 1000;
    size_t Nbytes = N*sizeof(double);
    double *h_a = (double*)malloc(Nbytes); //host memory
    double *d_a = NULL;
    HIP_CHECK(hipMalloc(&d_a, Nbytes));

    HIP_CHECK(hipMemcpy(d_a, h_a, Nbytes, hipMemcpyHostToDevice)); //copy data to device

    hipLaunchKernelGGL(myKernel, dim3((N+256-1)/256,1,1), dim3(256,1,1), 0, 0, N, d_a); //Launch kernel

    HIP_CHECK(hipGetLastError());

    HIP_CHECK(hipMemcpy(h_a, d_a, Nbytes, hipMemcpyDeviceToHost)); //copy results back to host

    free(h_a); //free host memory
    HIP_CHECK(hipFree(d_a)); //free device memory
}

__global__ void myKernel(int N, double *d_a) {
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<N) {
        d_a[i] *= 2.0;
    }
}
```

The host waits for the kernel to finish here
Streams

- A stream in HIP is a queue of tasks (e.g. kernels, memcpys, events).
  - Tasks enqueued in a stream **complete in order on that stream**.
  - Tasks being executed in different streams are allowed to overlap and share device resources.

- Streams are created via:
  ```c
  hipStream_t stream;
  hipStreamCreate(&stream);
  ```

- And destroyed via:
  ```c
  hipStreamDestroy(stream);
  ```

- Passing \(0\) or `NULL` as the `hipStream_t` argument to a function instructs the function to execute on a stream called the ‘NULL Stream’: 
  - No task on the NULL stream will begin until **all previously enqueued tasks in all other streams have completed**.
  - Blocking calls like `hipMemcpy` run on the NULL stream.
Suppose we have 4 small kernels to execute:

- hipLaunchKernelGGL(myKernel1, dim3(1), dim3(256), 0, 0, 256, d_a1);
- hipLaunchKernelGGL(myKernel2, dim3(1), dim3(256), 0, 0, 256, d_a2);
- hipLaunchKernelGGL(myKernel3, dim3(1), dim3(256), 0, 0, 256, d_a3);
- hipLaunchKernelGGL(myKernel4, dim3(1), dim3(256), 0, 0, 256, d_a4);

Even though these kernels use only one block each, they’ll execute in serial on the NULL stream:
Streams

- With streams we can effectively share the GPU’s compute resources:

  \[
  \text{hipLaunchKernelGGL}(
  \text{myKernel1}, \text{dim3}(1), \text{dim3}(256), 0, \text{stream1}, 256, d\_a1);
  \]

  \[
  \text{hipLaunchKernelGGL}(
  \text{myKernel2}, \text{dim3}(1), \text{dim3}(256), 0, \text{stream2}, 256, d\_a2);
  \]

  \[
  \text{hipLaunchKernelGGL}(
  \text{myKernel3}, \text{dim3}(1), \text{dim3}(256), 0, \text{stream3}, 256, d\_a3);
  \]

  \[
  \text{hipLaunchKernelGGL}(
  \text{myKernel4}, \text{dim3}(1), \text{dim3}(256), 0, \text{stream4}, 256, d\_a4);
  \]

---

**Note 1:** Check that the kernels modify different parts of memory to avoid data races.

**Note 2:** With large kernels, overlapping computations may not help performance.
Streams

- There is another use for streams besides concurrent kernels:
  - Overlapping kernels with data movement.

- AMD GPUs have separate engines for:
  - Host->Device memcpys
  - Device->Host memcpys
  - Compute kernels.

- These three different operations can overlap without dividing the GPU’s resources.
  - The overlapping operations should be in separate, non-NULL, streams.
  - The host memory should be pinned.
Pinned Memory

Host data allocations are pageable by default. The GPU can directly access Host data if it is pinned instead.

- Allocating pinned host memory:
  
  ```c
  double *h_a = NULL;
  hipHostMalloc(&h_a, Nbytes);
  ```

- Free pinned host memory:
  
  ```c
  hipHostFree(h_a);
  ```

- Host<->Device memcpy **bandwidth increases significantly when host memory is pinned.**
  - It is good practice to allocate host memory that is frequently transferred to/from the device as pinned memory.
Streams
Suppose we have 3 kernels which require moving data to and from the device:

```c
hipMemcpy(d_a1, h_a1, Nbytes, hipMemcpyHostToDevice);
hipMemcpy(d_a2, h_a2, Nbytes, hipMemcpyHostToDevice);
hipMemcpy(d_a3, h_a3, Nbytes, hipMemcpyHostToDevice);
```

```c
hipLaunchKernelGGL(myKernel1, blocks, threads, 0, 0, N, d_a1);
hipLaunchKernelGGL(myKernel2, blocks, threads, 0, 0, N, d_a2);
hipLaunchKernelGGL(myKernel3, blocks, threads, 0, 0, N, d_a3);
```

```c
hipMemcpy(h_a1, d_a1, Nbytes, hipMemcpyDeviceToHost);
hipMemcpy(h_a2, d_a2, Nbytes, hipMemcpyDeviceToHost);
hipMemcpy(h_a3, d_a3, Nbytes, hipMemcpyDeviceToHost);
```
Streams

Changing to asynchronous memcpys and using streams:

```c
hipMemcpyAsync(d_a1, h_a1, Nbytes, hipMemcpyHostToDevice, stream1);
hipMemcpyAsync(d_a2, h_a2, Nbytes, hipMemcpyHostToDevice, stream2);
hipMemcpyAsync(d_a3, h_a3, Nbytes, hipMemcpyHostToDevice, stream3);

hipLaunchKernelGGL(myKernel1, blocks, threads, 0, stream1, N, d_a1);
hipLaunchKernelGGL(myKernel2, blocks, threads, 0, stream2, N, d_a2);
hipLaunchKernelGGL(myKernel3, blocks, threads, 0, stream3, N, d_a3);

hipMemcpyAsync(h_a1, d_a1, Nbytes, hipMemcpyDeviceToHost, stream1);
hipMemcpyAsync(h_a2, d_a2, Nbytes, hipMemcpyDeviceToHost, stream2);
hipMemcpyAsync(h_a3, d_a3, Nbytes, hipMemcpyDeviceToHost, stream3);
```
Synchronization

How do we coordinate execution on device streams with host execution? Need some synchronization points.

- `hipDeviceSynchronize();`
  - Heavy-duty sync point.
  - Blocks host until all work in all device streams has reported complete.

- `hipStreamSynchronize(stream);`
  - Blocks host until all work in stream has reported complete.

Can a stream synchronize with another stream? For that we need ‘Events’.
Events

A `hipEvent_t` object is created on a device via:

```c
hipEvent_t event;
hipEventCreate(&event);
```

We queue an event into a stream:

```c
hipEventRecord(event, stream);
```

- The event records what work is currently enqueued in the stream.
- When the stream’s execution reaches the event, the event is considered ‘complete’.

At the end of the application, event objects should be destroyed:

```c
hipEventDestroy(event);
```
Events

What can we do with queued events?

- **hipEventSynchronize**(event);
  - Block host until event reports complete.
  - Only a synchronization point with respect to the stream where event was enqueued.

- **hipEventElapsedTime**(\&time, startEvent, endEvent);
  - Returns the time in ms between when two events, startEvent and endEvent, completed.
  - Can be very useful for timing kernels/memcpys.

- **hipStreamWaitEvent**(stream, event);
  - Non-blocking for host.
  - Instructs all future work submitted to stream to wait until event reports complete.
  - Primary way we enforce an ‘ordering’ between tasks in separate streams.
Streams

A common use-case for streams is MPI traffic:

//Queue local compute kernel
hipLaunchKernelGGL(myKernel, blocks, threads, 0, computeStream, N, d_a);

//Copy halo data to host
hipMemcpyAsync(h_commBuffer, d_commBuffer, Nbytes, hipMemcpyDeviceToHost, dataStream);
hipStreamSynchronize(dataStream); //Wait for data to arrive

//Exchange data with MPI
MPI_Data_Exchange(h_commBuffer);

//Send new data back to device
hipMemcpyAsync(d_commBuffer, h_commBuffer, Nbytes, hipMemcpyHostToDevice, dataStream);

<table>
<thead>
<tr>
<th>NULL Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>computeStream</td>
</tr>
<tr>
<td>dataStream</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>myKernel</td>
</tr>
<tr>
<td>DToH</td>
</tr>
<tr>
<td>HToD</td>
</tr>
</tbody>
</table>

MPI
Device code, shared memory, and thread synchronization
Function Qualifiers

hipcc make two compilation passes through source code. One to compile host code, and one to compile device code.

▪ **__global__** functions:
  ▪ These are entry points to device code, called from the host
  ▪ Code in these regions will execute on SIMD units

▪ **__device__** functions:
  ▪ Can be called from **__global__** and other **__device__** functions.
  ▪ Cannot be called from host code.
  ▪ Not compiled into host code – essentially ignored during host compilation pass

▪ **__host__ __device__** functions:
  ▪ Can be called from **__global__**, **__device__**, and host functions.
  ▪ Will execute on SIMD units when called from device code!
SIMD Execution

On SIMD units, be aware of divergence.

- Branching logic (if – else) can be costly:
  - Wavefront encounters an if statement
  - Evaluates conditional
    - If true, continues to statement body
    - If false, also continues to statement body with all instructions replaced with NoOps.
  - Known as ‘thread divergence’

- Generally, wavefronts diverging from each other is okay.
- Thread divergence within a wavefront can impact performance.
if (threadIdx.x % 2) {
    a *= 2.0;
} else {
    a *= 3.14;
}

//if (threadIdx.x % 2) {
//    NoOp;
//} else {
//    a *= 3.14;
//}
Memory declarations in Device Code

▪ Malloc/free not supported in device code.

▪ Variables/arrays can be declared on the stack.

▪ Stack variables declared in device code are allocated in registers and are private to each thread.

▪ Threads can all access common memory via device pointers, but otherwise do not share memory.
  ▪ Important exception: __shared__ memory

▪ Stack variables declared as __shared__:
  ▪ Allocated once per block in LDS memory
  ▪ Shared and accessible by all threads in the same block
  ▪ Access is faster than device global memory (but slower than register)
  ▪ Must have size known at compile time
Shared Memory

__global__ void reverse(double *d_a) {
    __shared__ double s_a[256]; //array of doubles, shared in this block

    int tid = threadIdx.x;
    s_a[tid] = d_a[tid]; //each thread fills one entry

    //all wavefronts must reach this point before any wavefront is allowed to continue.
    //something is missing here...
    __syncthreads();

    d_a[tid] = s_a[255-tid]; //write out array in reverse order
}

int main() {
    ...
    hipLaunchKernelGGL(reverse, dim3(1), dim3(256), 0, 0, d_a); //Launch kernel
    ...
}
Thread Synchronization

- **__syncthreads():**
  - Blocks a wavefront from continuing execution until all wavefronts have reached __syncthreads()
  - Memory transactions made by a thread before __syncthreads() are visible to all other threads in the block after __syncthreads()
  - Can have a noticeable overhead if called repeatedly

- **Best practice:** Avoid deadlocks by checking that all threads in a block execute the same __syncthreads() instruction.

- **Note 1:** So long as at least one thread in the wavefront encounters __syncthreads(), the whole wavefront is considered to have encountered __syncthreads().

- **Note 2:** Wavefronts can synchronize at different __syncthreads() instructions, and if a wavefront exits a kernel completely, other wavefronts waiting at a __syncthreads() may be allowed to continue.
HIP API

- **Device Management:**
  - `hipSetDevice()`, `hipGetDevice()`, `hipGetDeviceProperties()`

- **Memory Management**
  - `hipMalloc()`, `hipMemcpy()`, `hipMemcpyAsync()`, `hipFree()`

- **Streams**
  - `hipStreamCreate()`, `hipSynchronize()`, `hipStreamSynchronize()`, `hipStreamFree()`

- **Events**
  - `hipEventCreate()`, `hipEventRecord()`, `hipStreamWaitEvent()`, `hipEventElapsedTime()`

- **Device Kernels**
  - `__global__`, `__device__`, `hipLaunchKernelGGL()`

- **Device code**
  - threadIdx, blockIdx, blockDim, __shared__
  - 200+ math functions covering entire CUDA math library.

- **Error handling**
  - `hipGetLastError()`, `hipGetErrorString()`
Dynamic Shared Memory

- Can actually use `_shared_` arrays when sizes aren’t known at compile time
  - Called dynamic shared memory
  - Declare one array using `HIP_DYNAMIC_SHARED` macro, use for all dynamic LDS space
  - Use the `hipLaunchKernelGGL` argument we haven’t discussed yet
Dynamic Shared Memory

__global__ void reverse(double *d_a, int N) {
    HIP_DYNAMIC_SHARED(double, s_a); //dynamic array of doubles, shared in this block

    int tid = threadIdx.x;
    s_a[tid] = d_a[tid];       //each thread fills one entry

    //all wavefronts should reach this point before any wavefront is allowed to continue.
    __syncthreads();

    d_a[tid] = s_a[N-1-tid];   //write out array in reverse order
}

int main() {
    ...
    size_t NsharedBytes = N*sizeof(double);
    hipLaunchKernelGGL(reverse, dim3(1), dim3(N), NsharedBytes, 0, d_a, N); //Launch kernel
    ...
}
Atomic Operations

Atomic functions:
- Perform a read+write of a single 32 or 64-bit word in device global or LDS memory
- Can be called by multiple threads in device code
- Performed in a conflict-free manner

- AMD GPUs support atomic operations on 32-bit integers in hardware
  - Float /double atomics implemented as atomicCAS (Compare And Swap) loops, may have poor performance

- Can check at compile time if 32 or 64-bit atomic instructions are supported on target device
  - #ifdef __HIP_ARCH_HAS_GLOBAL_INT32_ATOMICS__
  - #ifdef __HIP_ARCH_HAS_GLOBAL_INT64_ATOMICS__
Atomic Operations

Supported atomic operations in HIP:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type, T</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>T atomicAdd(T* address, T val)</td>
<td>int, long long int, float, double</td>
<td>Adds val to *address</td>
</tr>
<tr>
<td>T atomicExch(T* address, T val)</td>
<td>int, long long int, float</td>
<td>Replace *address with val and return old value</td>
</tr>
<tr>
<td>T atomicMin(T* address, T val)</td>
<td>int, long long int</td>
<td>Replaces *address if val is smaller</td>
</tr>
<tr>
<td>T atomicMax(T* address, T val)</td>
<td>int, long long int</td>
<td>Replaces *address if val is larger</td>
</tr>
<tr>
<td>T atomicAnd(T* address, T val)</td>
<td>int, long long int</td>
<td>Bitwise AND between *address and val</td>
</tr>
<tr>
<td>T atomicOr(T* address, T val)</td>
<td>int, long long int</td>
<td>Bitwise OR between *address and val</td>
</tr>
<tr>
<td>T atomicXor(T* address, T val)</td>
<td>int, long long int</td>
<td>Bitwise XOR between *address and val</td>
</tr>
</tbody>
</table>
AMD GPU Compilers

- AMD supports several compilers that emit AMDGCN assembly
  - `hcc`
  - AOMP (in development, but mentioning for completeness)
  - To inspect AMDGCN, instead of `objdump` you need `extractkernel` (installed with `hcc`)
  - The GCN ISA is free and open: https://developer.amd.com/resources/developer-guides-manuals/

- `hcc`
  - Invoked by `hipcc` and is just a Perl script that wraps around `hcc`
  - Compiles HIP code
  - HIP (Heterogeneous Interface for Portability) is an interface that looks similar to CUDA
  - `hcc` is a fork of clang
  - It understands HIP and emits AMDGCN in the resulting binary
  - `hipcc -> hcc (clang) -> amdgcn`
  - All the x86 pieces are dealt with in the same way

- AOMP (AMD OpenMP Compiler)
  - Compiles C/C++ code with OpenMP “target” pragmas
  - Links with `libomptarget` to produce a binary that can offload work to the GPU
AMD GPU Compilers: the FORTRAN story

- FORTRAN is a technology important to the US Department of Energy
- AMD has plans to support OpenMP 4.5+ target offload from FORTRAN with two open source options
  - F18 (based on llvm)
  - gfortran
- FORTRAN compiler work is an ongoing effort
- See the Frontier spec sheet for what is expected to be supported on Frontier
Usage is pretty simple and accepts all/any flags that vanilla clang accepts

- `hipcc file.cpp -o a.out`

What is happening here?

- Set `HIP_VERBOSE=7` to see a bunch of useful information
  - Compile and link lines
  - Various paths

```
$ HIPVERBOSE=7 hipcc dotprod.cpp -o dotprod
HIP_PATH=/opt/rocm
HIP_PLATFORM=hcc
HSA_PATH=/opt/rocm/hsa
HCC_HOME=/opt/rocm/hcc

hipcc-args: dotprod.cpp -o dotprod

hipcc-cmd: /opt/rocm/hcc/bin/hcc -hc -D__HIPCC__ -I/opt/rocm/hcc/include -I/opt/rocm/include/hip/hcc_detail/cuda -I/opt/rocm/hsa/include -Wno-deprecated-register -I/opt/rocm/include -D__HIP_VERSION_MAJOR=1 -D__HIP_VERSION_MINOR=5 -D__HIP_VERSION_PATCH=19025 -D__HIP_ARCH_GFX803__=1 -Wl,--

```
Installing ROCm

- Requirements:
  - Linux®!
    - Ubuntu: ROCm can be installed using a Debian repo.
    - CentOS/RHEL: ROCm can be installed using a yum repo.
    - Other distros: you must build from source (support planned for SUSE based distributions).
  - To run on AMD hardware you need discrete GPUs in families GFX8 (Polaris) or GFX9 (Vega)
  - APUs aren’t currently officially supported

- ROCm has been on a monthly release cycle
- ROCm is now compatible with AMD drivers in some upstream linux kernels.
- ROCm can be installed with:
  - ROCK kernel driver (from ROCm repos)
    - Only supported on Ubuntu, CentOS/RHEL
  - AMD drivers in some upstream kernels
    - Should work on more distributions

- Latest install instructions can be found here: [https://github.com/RadeonOpenCompute/ROCM](https://github.com/RadeonOpenCompute/ROCM)
- Also check out: [https://github.com/RadeonOpenCompute/Experimental_ROC](https://github.com/RadeonOpenCompute/Experimental_ROC)
Querying System

- **rocminfo**: Queries and displays information on the system’s hardware
  - More info at: [https://github.com/RadeonOpenCompute/rocminfo](https://github.com/RadeonOpenCompute/rocminfo)

- **rocm-smi**: Queries and sets AMD GPU frequencies, power usage, and fan speeds
  - `sudo` privileges are needed to set frequencies and power limits
  - `sudo` privileges are not needed to query information
  - Get more info by running `rocm-smi -h` or looking at: [https://github.com/RadeonOpenCompute/ROC-smi](https://github.com/RadeonOpenCompute/ROC-smi)

- **Querying ROCm version**:
  - If you install ROCm in the standard location (`/opt/rocm`) version info is at: `/opt/rocm/.info/version-dev`
  - Can also run the command ‘dkms status’ and the ROCm version will be displayed
AMD GPU Libraries

- A note on naming conventions:
  - roc* -> AMGCN library usually written in HIP
  - cu* -> NVIDIA PTX libraries
  - hip* -> usually interface layer on top of roc*/cu* backends

- hip* libraries:
  - Can be compiled by hipcc and can generate a call for the device you have:
    - hipcc->AMDGCU
    - hipcc->nvcc (inlined)->NVPTX
  - Just a thin wrapper that marshals calls off to a “backend” library:
    - corresponding roc* library backend containing optimized GCN or
    - corresponding cu* library backend containing NVPTX for NVIDIA devices
  - E.g., hipBLAS is a marshalling library:

  ![Diagram](hipBLAS.png)
### Decoder ring: Math library equivalents

<table>
<thead>
<tr>
<th>CUBLAS</th>
<th>ROCBLAS</th>
<th>Basic Linear Algebra Subroutines</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUFFT</td>
<td>ROCFFT</td>
<td>Fast Fourier Transforms</td>
</tr>
<tr>
<td>CUDNN</td>
<td>MIOPEN</td>
<td>Deep Learning Library</td>
</tr>
<tr>
<td>CUB</td>
<td>ROCPRIM</td>
<td>Optimized Parallel Primitives</td>
</tr>
<tr>
<td>EIGEN</td>
<td>EIGEN</td>
<td>C++ Template Library for Linear Algebra</td>
</tr>
</tbody>
</table>

MORE INFO AT: GITHUB.COM/ROCM-DEVELOPER-TOOLS/HIP → HIP_PORTING_GUIDE.MD
AMD GPU Libraries: BLAS

- rocBLAS – `sudo apt install rocblas`
  - Source code: https://github.com/ROCmSoftwarePlatform/rocBLAS
  - Documentation: https://rocblas.readthedocs.io/en/latest/
  - Basic linear algebra functionality
    - axpy, gemv, trsm, etc
  - Use hipBLAS if you need portability between AMD and NVIDIA devices

- hipBLAS - `sudo apt install hipblas`
  - Documentation: https://github.com/ROCmSoftwarePlatform/hipBLAS/wiki/Exported-functions
  - Use this if you need portability between AMD and NVIDIA
  - It is just a thin wrapper:
    - It can dispatch calls to rocBLAS for AMD devices
    - It can dispatch calls to cuBLAS for NVIDIA devices
AMD GPU Libraries: rocBLAS example

- **rocBLAS**
  - Level 1, 2, and 3 functionality
    - `axpy`, `gemv`, `trsm`, etc
  - Use `hipBLAS` only if you need portability between AMD and NVIDIA devices
  - Link with: `-lrocblas`

```c
#include <rocblas.h>

int main(int argc, char **argv) {
  rocblas_int N = 500000;

  // Allocate device memory
  double *dx, *dy;
  hipMalloc(&dx, sizeof(double) * N);
  hipMalloc(&dy, sizeof(double) * N);

  // Allocate host memory (and fill up the arrays) here
  std::vector<double> hx(N), hy(N);

  // Copy host arrays to device
  hipMemcpy(dx, hx.data(), sizeof(double) * N, hipMemcpyHostToDevice);
  hipMemcpy(dy, hy.data(), sizeof(double) * N, hipMemcpyHostToDevice);

  const double alpha = 1.0;
  rocblas_handle handle;
  rocblas_create_handle(&handle);
  rocblas_status status;
  status = rocblas_daxpy(handle, N, &alpha, dx, 1, dy, 1);
  rocblas_destroy_handle(handle);

  // Copy result back to host
  hipMemcpy(dy.data(), dy, sizeof(double) * N, hipMemcpyDeviceToHost);
  hipFree(dx);
  hipFree(dy);
  return 0;
}
```
Links to key libraries

- rocFFT: https://github.com/ROCmSoftwarePlatform/rocFFT
- hipSPARSE: https://github.com/ROCmSoftwarePlatform/hipSPARSE
- Rocalution: https://github.com/ROCmSoftwarePlatform/rocALUTION
- Tensile: https://github.com/ROCmSoftwarePlatform/Tensile
- RCCL (ROCm analogue of NCCL): https://github.com/ROCmSoftwarePlatform/rccl
- rocPRIM: https://github.com/ROCmSoftwarePlatform/rocPRIM
- rocRAND: https://github.com/ROCmSoftwarePlatform/rocRAND
- MIOpen: https://github.com/ROCmSoftwarePlatform/MIOpen
- Pytorch: https://github.com/ROCmSoftwarePlatform/pytorch
- Thrust: https://github.com/ROCmSoftwarePlatform/Thrust
Porting CUDA Applications to HIP

Nicholas Malaya <nicholas.malaya@amd.com>
Objectives

• This training:
  • demonstrates how to convert CUDA codes into HIP
  • explains the meaning of the term ‘hipify’
  • provides a simple means to examine port quality
  • provides an idea of the common ‘gotchas’ of porting apps
HIP Snapshot

• HIP: Heterogeneous-Compute Interface for Portability
  • C++ runtime API and kernel language with CUDA-like APIs
  • [https://github.com/ROCm-Developer-Tools/HIP](https://github.com/ROCm-Developer-Tools/HIP)

• Capabilities:
  • HIP 2.4 is the latest version (monthly releases)
  • Supports features up to CUDA 8.0
  • CUDA 9 and 10 features not supported
  • Based on LLVM
Getting started with HIP

<table>
<thead>
<tr>
<th>CUDA DAXPY</th>
<th>HIP DAXPY</th>
</tr>
</thead>
</table>
| __global__ void add(int n, double *x, double *y)  
  {  
    int index = blockIdx.x * blockDim.x + threadIdx.x;  
    int stride = blockDim.x * gridDim.x;  
    for (int i = index; i < n; i += stride)  
      {  
        y[i] = x[i] + y[i];  
      }  
  }  |
| __global__ void add(int n, double *x, double *y)  
  {  
    int index = blockIdx.x * blockDim.x + threadIdx.x;  
    int stride = blockDim.x * gridDim.x;  
    for (int i = index; i < n; i += stride)  
      {  
        y[i] = x[i] + y[i];  
      }  
  }  |

KERNELS ARE SYNTACTICALLY THE SAME
## CUDA APIs vs HIP API

<table>
<thead>
<tr>
<th>CUDA</th>
<th>HIP</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cudaMalloc(&amp;d_x, N*sizeof(double));</code></td>
<td><code>hipMalloc(&amp;d_x, N*sizeof(double));</code></td>
</tr>
<tr>
<td><code>cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);</code></td>
<td><code>hipMemcpy(d_x, x, N*sizeof(double), hipMemcpyHostToDevice);</code></td>
</tr>
<tr>
<td><code>cudaDeviceSynchronize();</code></td>
<td><code>hipDeviceSynchronize();</code></td>
</tr>
</tbody>
</table>
Launching a kernel

CUDA KERNEL LAUNCH SYNTAX

some_kernel<<<gridsize, blocksize, shared_mem_size, stream>>>(arg0, arg1, ...);

HIP KERNEL LAUNCH SYNTAX

hipLaunchKernelGGL(some_kernel, dim3(gridsize), dim3(blocksize), shared_mem_size, stream, arg0, arg1, ...);
How to port a CUDA code?
Enter HIPify

- Hipify tools are designed to help automatically convert CUDA code
- In practice, large portions of many HPC codes have been automatically Hipified:
  - ~90% of CUDA code in CORAL-2 HACC
  - ~80% of CUDA code in CORAL-2 PENNANT
  - The remaining code requires programmer intervention
- Tools (discussed in detail on subsequent slide):
  - Hipify-perl
  - Hipify-clang
HIPify tools

- **Hipify-perl**:
  - Easy to use – point at a directory and it will attempt to hipify CUDA code
  - Very simple string replacement technique: may make incorrect translations
    - `sed -e 's/cuda/hip/g'`, (e.g., `cudaMemcpy` becomes `hipMemcpy`)
  - Recommended for quick scans of projects

- **Hipify-clang**:
  - Requires clang compiler
  - More robust translation of the code. Uses clang to parse files and perform semantic translation
  - Can generate warnings and assistance for code for additional user analysis
  - High quality translation, particularly for cases where the user is familiar with the make system
Hipify-perl

- Sits in $HIP/bin/ (export PATH=$PATH:[MYHIP]/bin)
- Command line tool: hipify-perl foo.cu > new_foo.cpp
- Compile: hipcc new_foo.cpp
- How does this this work in practice?
  - Hipify source code
  - Check it in to your favorite version control
  - Try to build
  - Manually work on the rest
Hipify-clang

- Build from source
- hipify-clang has unit tests using LLVM lit/FileCheck (44 tests)

- Hipification requires same headers that would be needed to compile it with clang:
  - ./hipify-clang foo.cu -I /usr/local/cuda-8.0/samples/common/inc

- [https://github.com/ROCm-Developer-Tools/HIP/tree/master/hipify-clang](https://github.com/ROCm-Developer-Tools/HIP/tree/master/hipify-clang)
Example: HACC

- Hardware Accelerated Cosmology Code
- Simulates time-evolution of universe
  - \( \text{Mpc} = \text{Megaparsec} = 3.09 \times 10^{22} \text{ meters} \)
- **Our HIP success story:**
  - **Ported in an afternoon**
- Profiling:
  - 10% of time is spent in the tree walk
  - >80% in the short force kernels
    - **(GPU kernel)**
    - 5% in the 3d Transposes / FFTs

\[
f_{SR} = (s + \epsilon)^{-3/2} - f_{grid}(s)
\]

where,

\[
s = r \cdot r
\]

and,

\[
f_{grid}(s) = POLY_5(s)
\]
HACC: What made it a success

• What was easy?
  • Simple GPU kernel
  • Few library dependencies (FFTW, not in kernel)
  • No advanced CUDA features

• What was difficult?
  • Inline PTX: required translation to AMD GCN
  • Hand-written wave-32 code (for a reduction)
### CUDA

```c
    cudaMemcpyAsync(d_npos, h_npos, sizeof(float4) * SIZE, cudaMemcpyHostToDevice, stream);
    cudaMemcpyAsync(d_mask, h_mask, sizeof(MASK_T) * cnt, cudaMemcpyHostToDevice, stream);
    calcHHCullenDehnen<<<blocksPerGrid, threadsPerBlock, 0, stream>>>(cnt, SIZE, d_npos, d_mask, rsm);
    cudaMemcpyAsync(h_pos, d_npos + (SIZE - cnt), sizeof(float4) * cnt, cudaMemcpyDeviceToHost, stream);
    cudaMemcpyAsync(h_mask, d_mask, sizeof(MASK_T) * cnt, cudaMemcpyDeviceToHost, stream);
```

### HIP

```c
    hipMemcpyAsync(d_npos, h_npos, sizeof(float4) * SIZE, hipMemcpysHostToDevice, stream);
    hipMemcpyAsync(d_mask, h_mask, sizeof(MASK_T) * cnt, hipMemcpysHostToDevice, stream);
    hipLaunchKernelGGL((calcHHCullenDehnen), dim3(blocksPerGrid), dim3(threadsPerBlock), 0, stream, cnt, SIZE, d_npos, d_mask, rsm);
    hipMemcpyAsync(h_pos, d_npos + (SIZE - cnt), sizeof(float4) * cnt, hipMemcpyDeviceToHost, stream);
    hipMemcpyAsync(h_mask, d_mask, sizeof(MASK_T) * cnt, hipMemcpyDeviceToHost, stream);
```
So you ported an application...
# HACC: Comparison to the CUDA version

<table>
<thead>
<tr>
<th>MEASUREMENT</th>
<th>TITAN-V W/ CUDA 9.0</th>
<th>MI-25 W/ HIP (ROCM 1.9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGISTER USE (VGPR)</td>
<td>32</td>
<td>36</td>
</tr>
<tr>
<td>REGISTER SPILLING</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DIVERGENCE</td>
<td>92.5%</td>
<td>84%</td>
</tr>
</tbody>
</table>
# Performance portability

<table>
<thead>
<tr>
<th>MEASUREMENT</th>
<th>TITAN-V W/ CUDA 9.0</th>
<th>MI-25 W/ HIP (ROCM 1.9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGISTER USE (VGPR)</td>
<td>30, 74, 44, 58, 53</td>
<td>12, 73, 25, 38, 24</td>
</tr>
<tr>
<td>REGISTER SPILLING</td>
<td>0, 0, 0, 0, 0</td>
<td>0, 0, 0, 0, 0</td>
</tr>
<tr>
<td>DIVERGENCE</td>
<td>97, 95, 70, 99, 78</td>
<td>97, 90, 71, 98, 82</td>
</tr>
</tbody>
</table>

Pennant has 5 gpu kernels: Multiple entries denote each GPU kernel (GPUMain1, etc.)

Similar numbers observed in Quicksilver, HACC, etc.
HIPified (ported) codes

QUICKSILVER  HACC

SW4LITE  HPL

PENNANT  LAGHOS

AND (ALREADY) MANY MORE...
What to look for when porting:

- Inline PTX assembly
- CUDA intrinsics
- Hard-coded dependencies on warp size, shared memory size
  - “grep 32”
  - Do not use hard coded dependencies on warp size!
- Code geared toward limited size of register file on NVIDIA hardware
- Functions implicitly inlined
- Unified Memory
QUESTIONS?
AMD GPU programming resources

- ROCm platform: https://github.com/RadeonOpenCompute/ROCm/
  - With instructions for installing from Debian/CentOS/RHEL binary repositories
  - Has links to source repositories for all components, including HIP
- ROCm/HIP libraries: https://github.com/ROCmSoftwarePlatform
- ROC-profiler: https://github.com/ROCM-Developer-Tools/rocprofiler
  - Collects application traces and performance counters
  - Trace timeline can be visualized with chrome://tracing
- AMD GPU ISA docs and more: https://developer.amd.com/resources/developer-guides-manuals/
CUDA features not supported by HIP

- CUDA 5.0:
  - Dynamic Parallelism (not supported)
  - cuIpc functions (under development).

- CUDA 5.5:
  - CUPTI (not directly supported, AMD GPUPerfAPI an alternative in some cases)

- CUDA 6.0
  - Managed memory (under development)
CUDA features not in HIP, cont.

- CUDA 7.0
  - Per-thread-streams (under development)

- CUDA 8.0
  - Page Migration including cudaMemAdvise, cudaMemPrefetch, other cudaMem* APIs (not supported)