Introduction to HIP

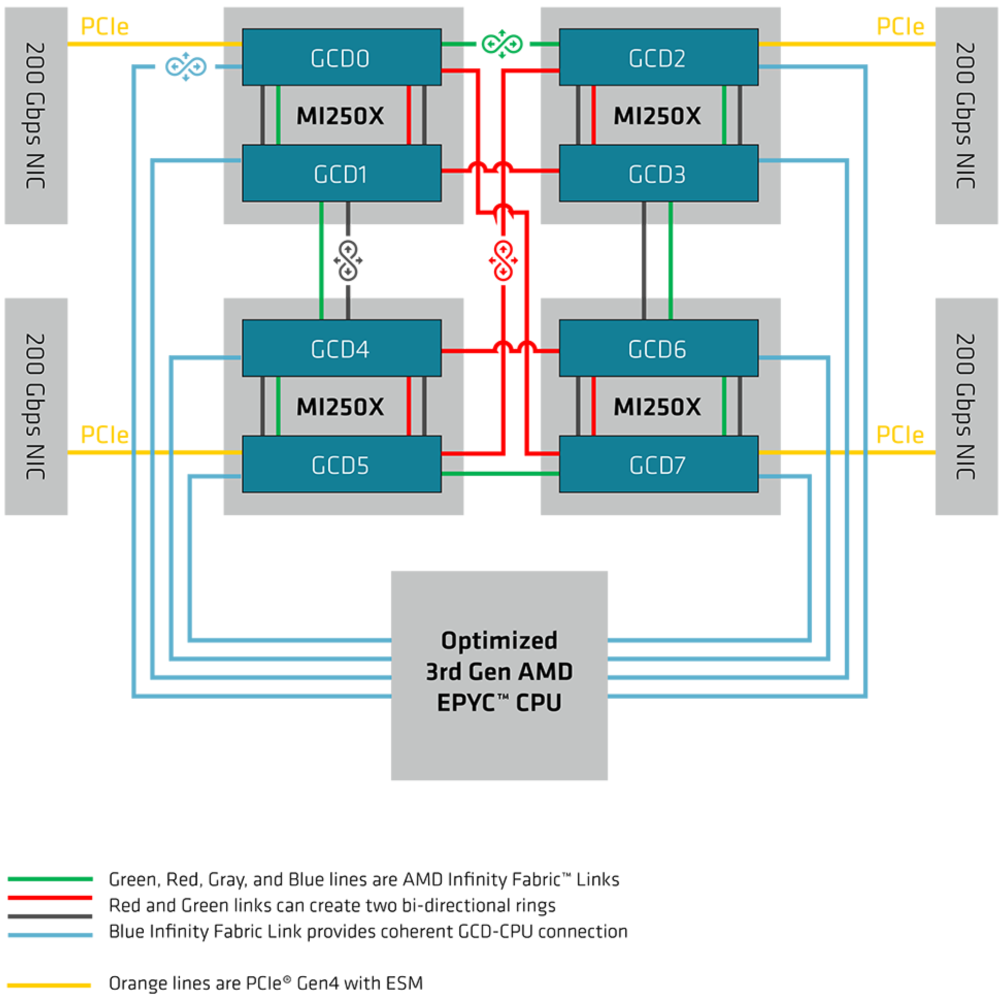
Jonathan Vincent

What is HIP

- HIP is AMD’s Heterogeneous-compute Interface for Portability
- Open Source
- C++ Runtime
  - No native FORTRN
- Syntax is similar to CUDA (relatively easy to convert CUDA to HIP)

Reminder - Host and Device

Optimized 3rd Gen AMD EPYC™ Processor + AMD Instinct™ MI250X Accelerator



Each Dardell GPU node has

- One AMD EPYC™ processor with 64 cores
- Four AMD Instinct™ MI250X GPUs connected by AMD Infinity Fabric™ Links
    - Each MI250x has two GPU dies, so 8 GPUs per node
- 4 Network cards (one connected to each MI250x)

- CPU and GPU have separate memory
- CPU also known as HOST
- GPU also known as DEVICE

CUDA vs HIP terminology

HIP Term	CUDA Term	Definition
Compute Unit (CU)	Streaming Multiprocessor (SM)	Parallel vector processor where computation is done
Work Item/Thread	Thread	Single thread of work
Wavefront	Warp	Collection of threads that run in parallel. On HIP systems run in lockstep
Workgroup	Thread Block	Group of threads/wavefronts
Local memory	Shared memory	Memory shared between workgroups

Thread Grid

- Kernels are launched in on a 3D grid of threads
  - Dimensions can be set to size 1 if needed
- Threads are orginised into groups of threads called blocks
- Each thread is then given an unique ID
  - ThreadIdx.x, ThreadIdx.y ThreadIdx.z
  - BlockIdx.x, BlockIdx.y BlockIdx.z
  - BlockDim.x, BlockDim.y, BlockDim.z
- For 1D grid
  - `int i = threadIdx.x + blockIdx.x*blockDim.x;`

Kernels

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

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

- Kernels are device functions to be launched on the device. This is denoted by the __global__ attribute
- Kernels should be declared void (i.e. not have a return value)
- Kernels act on device memory, so pointers must be pointers to device memory
- All threads potentially run at the same time (beware race conditions)
- threadIdx and BlockIdx used to compute unique ID for the thread
- Unique ID potentially larger than N

Launching Kernels

- Kernels are launched by the host.

```
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 local memory
                   0,               //Stream (0=NULL stream)
                   N, a);           //Kernel arguments
```

GPU Memory Management

- GPU and CPU have separate memory so need to transfer data
- Allocate memory on GPU
  - `hipMalloc((void**)&Ad, BYTES);`
- Free memory on GPU
  - `hipFree(Ad);`
- Transfer to GPU
  - `hipMemcpy(dest_array, source_array, bytes, hipMemcpyHostToDevice);`
- Transfer from GPU
  - `hipMemcpy(dest_array, source_array, bytes, hipMemcpyDeviceToHost);`

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:
  - `hipStream_t stream;`
  - `hipStreamCreate(&stream);`
- And destroyed via:
  - `hipStreamDestroy(stream);`

Kernel Synchronisation

- Kernels are launched asynchronously
  - CPU code will continue after Kernel launch
  - Need to
- Sync within kernel
  - `__syncthreads();`
- Sync between kernels
  - `hipStreamSynchronize();`
  - `hipMemcpy();`
    - > *Memcpy is placed in stream, and does not complete until copy complete*
    - > *Memcpy can also be done asynchronously with hipMemcpyAsync*

Error Handling

- Kernels are launched asymmetrically
  - Potential problem knowing which kernel generated the error
- Good practice to check for errors after each kernel launch (normally with macro)
  - Potential to get wrong kernel due to asymmetry.
- Error handling

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

Complete Program

```
#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));
   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 (and blocks until kernel finished)
   …
   free(h_a); //free host memory
   HIP_CHECK(hipFree(d_a)); //free device memory
}
```

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

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

Multiple Devices – Controlling

- Dardel GPU nodes have effectively 8 GPUs
  - Need to manage which device is accessed
- Two methods
  - One MPI rank per GPU
  - Single CPU process controls all GPUs

Multiple Devices – Single CPU Process

- Single CPU process controls all GPUs
  - Query number of devices visible to system
    - `hipSetDevice(deviceID);`
  - Let the runtime know which device to use
    - *int deviceID = 0;*
    - *hipSetDevice(deviceID);*

Optimisation

- Local memory
- Pinned copies
- Tools
  - Rocprof
  - Omnitrace

Local memory

- GPU L1 cache memory can also be used directly by the programmer as local memory
  - You are effectively manually copying data to the L1 cache and attempting to beat compiler at optimisation
  - Local memory only seen by workgroup/threadblock
  - __syncthreads() required before all threads see data
  - You are effectively manually copying data to the L1 cache and attempting to beat compiler at optimisation
    - > *Potential performance optimisation*
    - > *Can make things worse if done badly*

Local memory – Statically allocated

- Statically allocated Local memory

```
__global__ void myKernel(int N, double *d_a) {
    __shared__ float  A[10];
…
}
```

- Size of array is fixed at runtime

Local memory – Dynamically allocated

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

- Size of array defined dynamically at runtime

```
__global__ void myKernel(int N, double *d_a) {
    extern __shared__ float  A[];
…
}
```

Pinned Memory - hipHostMalloc

- Linux supports virtual memory
  - CPU memory can potentially be paged out to disk at runtime
    - > *Not normally needed for HPC*
  - CPU memory can also be "pinned" so that it cannot be paged
  - For data transfers to GPU pinned memory must be used
    - > *If starting data is not pinned, it is first copied to pinned memory then to GPU*
    - > *Simpler to pin yourself*
- Allocating pinned memory on CPU (Host)

```
hipHostMalloc(&A_h, sizeBytes);
```

HIP API - Summary

- Device Management:
  - hipSetDevice(), hipGetDevice(), hipGetDeviceProperties()
- Memory Management
  - hipMalloc(), hipMemcpy(), hipMemcpyAsync(), hipFree()
- Streams
  - hipStreamCreate(), hipSynchronize(), hipStreamSynchronize(), hipStreamFree()
- Events
  - hipEventCreate(), hipEventRecord(), hipStreamWaitEvent(), hipEventElapsedTime()

HIP API - Summary

- Device Kernels
  - __global__, __device__, hipLaunchKernelGGL()
- Device code
  - threadIdx, blockIdx, blockDim, __shared__
- Error handling
  - hipGetLastError(), hipGetErrorString()

- More Information
  - https://rocm.docs.amd.com/projects/HIP/en/latest/

Questions?

Email – jonvin@kth.se

Calling HIP from Fortran

- Unlike CUDA, HIP does not have a FORTRAN interface
- Calling C from Fortran has long history and is well established
  - Function calls and numerical data easy to change
  - Strings more complicated
- The FORTRAN runtime libraries make it simpler if the main program is in FORTRAN and C routines are called from FORTRAN
- Two methods
  - Historical method
    - > *Manual processing using understanding of how FORTRAN compiler changes (mangles) function names and stores data*
  - Modern method
    - > *Dedicated interface methods added in modern FORTRAN (FORTRAN 2003)*

General

- Integer and floating point variables are compatable
- Variables are passed by reference in Fortran
- Array order is reversed
  - A[I][J] in C
  - A(J,I) in Fortran
- Strings harder to transfer

Historical method

- Fortran is case insensitive, C is case sensitive
  - All C to be called from FORTRAN routines need to be lower case
  - Add a trailing underscore to the C routine name (e.g. cfunction_)
  - Historically FORTRAN had different name mangling, but this is most common

Historical method - Example

- Fortran

```
integer i
real r
external CSim
i = 100
call CSim(i,r)
...
```

- C

```
void csim_(int *i, float *r)
{
*r = *i;
}
```

Modern method – Introduced Fortran 2003

- Fortran 2003 introduced iso_c_binding interface

```
Interface
  function CSim(i,r) &
          bind(C,name="CSim")
  Integer I
  Real R
End Interface
```

- C

```
void CSim(int *i, float *r)
{
  *r = *i;
}
```

Questions?

Email – jonvin@kth.se