PARALLEL PROGRAMMING...

Parallel Computing Using HIP

SESSION 3/3

GOAL

Programming Interface for parallel computing With HIP

- What is HIP ?
- Synchronization and streams
- Memory allocations, access and unified memory
- Kernel optimization and profiling
- Device Code, Shared Memory,and Thread Synchronization
- Multi-GPU programming and HIP+MPI

API Examples

What is HIP ?

What is HIP ?

Heterogeneous-Compute Interface for Portability (HIP)

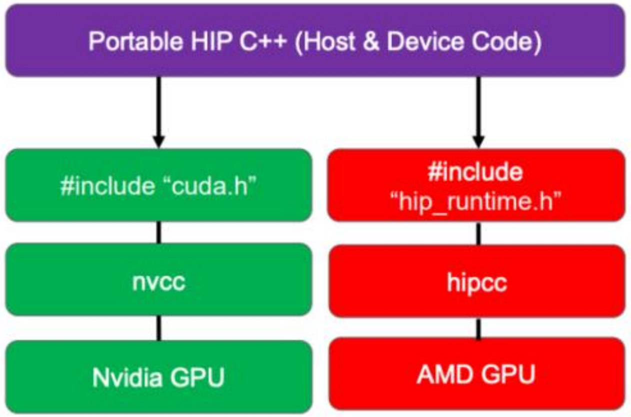
HIP, is a C++ runtime API and kernel language that allows developers to create portable applications that can run on **AMD's accelerators** as well as **CUDA** devices Portable **HIP C++**

AMD effort to offer a common programming interface that works on both devices: **CUDA** and **ROCm**.

What is HIP ?

Heterogeneous-Compute Interface for Portability (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.
- C++ runtime API and kernel language that allows developers to create portable applications.



ROCm: is an Advanced Micro Devices (AMD) software stack for graphics processing unit (GPU) programming and spans several domains: GPGPU, HPC ,heterogeneous computing

HIP comparison with CUDA

CUDA

```
cudaMalloc((void**)&nodes_dev, N*sizeof(float) );
```

HIP

```
hipMalloc((void**)&nodes_dev, N*sizeof(float) );
```

CUDA

```
dim3 threadsPerBlock(nthreads,nthreads,nthreads);
dim3 blocks(n_elements);
GPUKernel<<<blocks,threadsPerBlock>>>( input );
```

HIP

```
dim3 threadsPerBlock(nthreads,nthreads,nthreads);
dim3 blocks(n_elements);
hipLaunchKernelGGL(GPUKernel, dim3(blocks), dim3(threadsPerBlock), 0, 0, input);
```

COMPILATION

With CUDA

```
==>$ nvcc source_code.cu
```

With HIP

```
==>$ hipcc source_code.cu
```

Difference between HIP and CUDA

Some things to be aware of writing HIP, or porting from CUDA:

- AMD GCN hardware 'warp' size = 64 (warps are referred to as 'wavefronts' in AMD documentation)
- Device and host pointers allocated by HIP API use flat addressing
  - Unified virtual addressing is enabled by default
  - Unified memory is available, but does not perform optimally currently
- Dynamic parallelism not currently supported
- CUDA 9+ thread independent scheduling not supported (e.g., no __syncwarp)
- Some CUDA library functions do not have AMD equivalents
- Shared memory and registers per thread can differ between AMD and Nvidia hardware
- Inline PTX or AMD GCN assembly is not portable

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

AMD GPU Terminology

Compute Unit
- one of the parallel vector processors in a GPU

Kernel
- function launched to the GPU that is executed by multiple parallel workers

Thread
- individual lane in a wavefront

Wavefront (cf. CUDA warp)
- collection of threads that execute in lockstep and execute the same instructions
- each wavefront has 64 threads
- number of wavefronts per workgroup is chosen at kernel launch

Workgroup (cf. CUDA thread block)
- group of wavefronts (threads) that are on the GPU at the same time and
- are part of the same compute unit (CU)
-can synchronise together and communicate through memory in the CU

HIP Kernel Language

HIP Kernel Language

Qualifiers: __device__, __global__, __shared__, ...
Built-in variables: threadIdx.x, blockIdx.y, ...
Vector types: int3, float2, dim3, ...
Math functions: sqrt, powf, sinh, ...
Intrinsic functions: synchronisation, memory-fences etc.

API

Device init and management
Memory management
Execution control
Synchronisation: device, stream, events
Error handling, context handling

Programming models

GPU accelerator is often called a device and CPU a host

Parallel code (kernel) is launched by the host and executed on a device by several threads

Code is written from the point of view of a single thread each thread has a unique ID

Device Kernels: The Grid

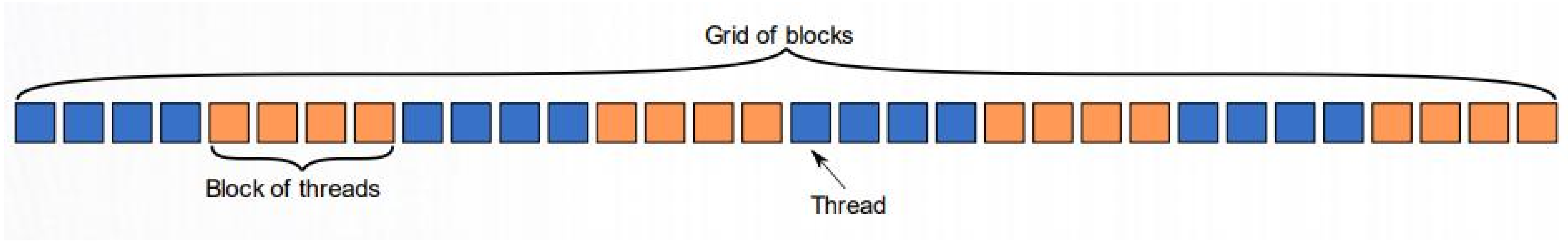
- In HIP, kernels are executed on a 3D ”grid”
  -You might feel comfortable thinking in terms of a mesh of points, but it’s not required
- The “grid” is what you can map your problem to
  - -It’s not a physical thing, but it can be useful to think that way
- AMD devices (GPUs) support 1D, 2D, and 3D 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

Device Kernels: The Grid

Some Terminology

CUDA	HIP	OpenCL™
grid	grid	NDRange
block	block	work group
thread	work item / thread	work item
warp	wavefront	sub-group

Device Kernels: The Grid blocks of threads in 1D



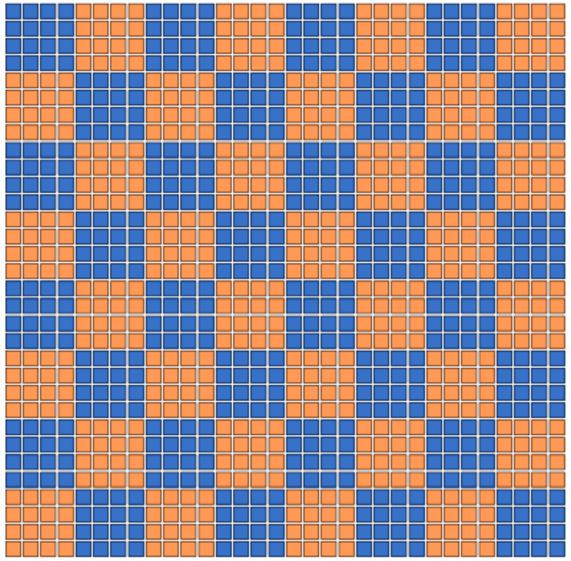
Threads in grid have access to:

- Their respective block: blockIdx.x
- Their respective thread ID in a block: threadIdx.x
- Their block's dimension: blockDim.x
- The number of blocks in the grid: gridDim.x

Device Kernels: 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: blockIdx.x, blockIdx.y
  - Their respective thread IDs in a block: threadIdx.x, threadIdx.y



Device Kernels

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)‹

Device Kernels

Kernels are launched from 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 LDS space (see extra slides)
                   0,                 //Stream (0=NULL stream)
                   N, a);             //Kernel arguments
```

Analogous to CUDA kernel launch syntax:

```
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 Kernels

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

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

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

    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);
```

Can copy strided sections of arrays:

```
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 = hipPeekLastError();
- 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; \ std::abort(); } }
```

Putting 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(); } }
```

GPU Programming consideration

GPU model requires many small tasks executing a kernel
- e.g. can replace iterations of loop with a GPU kernel call

Need to adapt CPU code to run on the GPU
- rethink algorithm to fit better into the execution model
- keep reusing data on the GPU to reach high occupancy of the hardware
- if necessary, manage data transfers between CPU and GPU memories carefully (can easily become a bottleneck!)

Grid: thread hierarchy

Kernels are executed on a 3D grid of threads
- threads are partitioned into equalsized blocks

Code is executed by the threads,
the grid is just a way to organise
the work

Dimension of the grid are set at
kernel launch

Built-in variables to be used within a kernel:
- threadIdx, blockIDx, blockDim, gridDim

Kernels

Kernel is a (device) function to be executed by the GPU

Function should be of void type and needs to be declared with the

__global__ or __device__ attribute

All pointers passed to the kernel need to point to memory accessible from the device

Unique thread and block IDs can be used to distribute work

HIP Synchronization and streams

What is a stream ?

- A sequence of operations that execute in issue-order on the GPU
- HIP operations in different streams could run concurrently
- The ROCm 4.5.0 brings the Direct Dispatch, the runtime directly queues a packet to the AQL queue in Dispatch and some of the synchronization.
- The previous ROCm uses queue per stream

What is a stream ?

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);

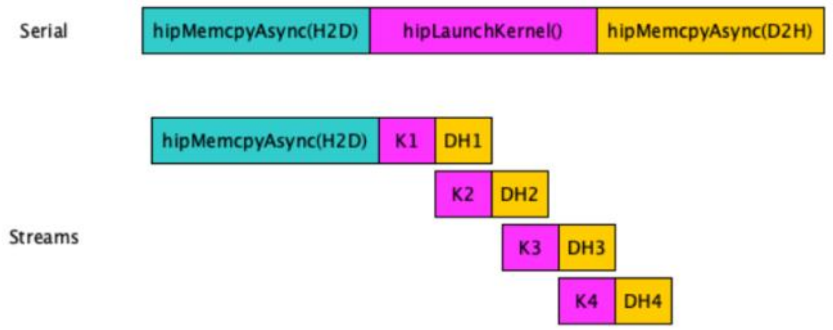
Passing 0 or NULL as the hipStream_t argument to a function instructs the function to execute on a stream called the 'NULL Stream':

- No task on the NULL stream will begin until all previously enqueued tasks in all other streams have completed.

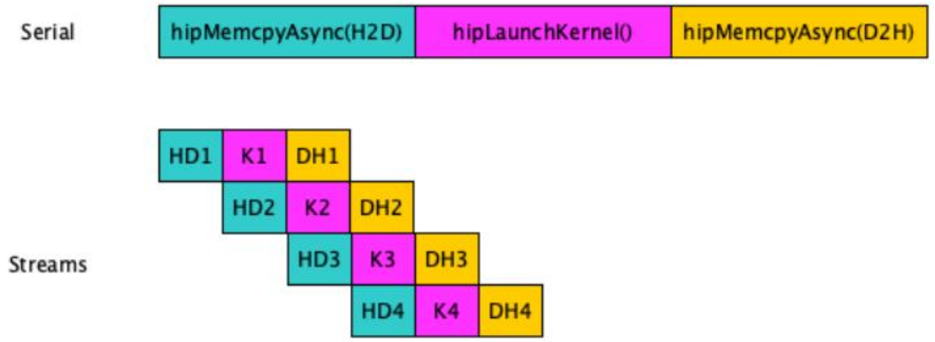
- Blocking calls like hipMemcpy run on the NULL stream.

Stream

Concurreny



Amount of concurrency



Defaut

Only a single stream is used if not defined
Commands are synchronized except the Async calls and Kernels

Stream

Stream/Events API

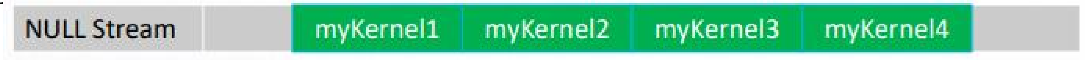
hipStreamCreate:	Creates an asynchronous stream
hipStreamDestroy	Destroy an asynchronous stream
hipStreamCreateWithFlags	Creates an asynchronous stream with specified flags
hipEventCreate	Create an event
hipEventRecord	Record an event in a specified stream
hipEventSynchronize:	Wait for an event to complete
hipEventElapsedTime:	Return the elapsed time between two events
hipEventDestroy:	Destroy the specified event

Stream

▪ Suppose we have 4 small kernels to execute:

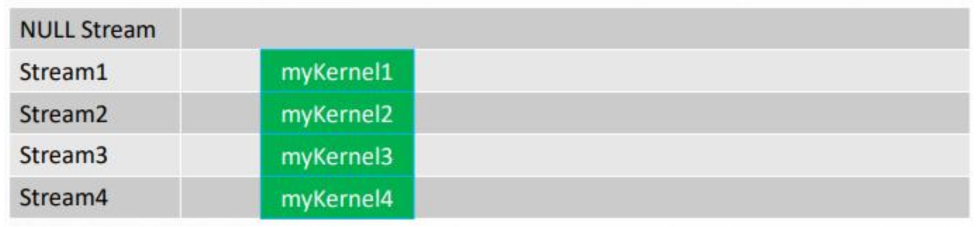
```
hipLaunchKernelGGL(myKernel1, dim3(1), dim3(256), 0, 0, 256, d_a1);
hipLaunchKernelGGL(myKernel2, dim3(1), dim3(256), 0, 0, 256, d_a2);
hipLaunchKernelGGL(myKernel3, dim3(1), dim3(256), 0, 0, 256, d_a3);
hipLaunchKernelGGL(myKernel4, dim3(1), dim3(256), 0, 0, 256, d_a4);
```

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



▪ With streams we can effectively share the GPU's compute resources:

```
hipLaunchKernelGGL(myKernel1, dim3(1), dim3(256), 0, stream1, 256, d_a1);
hipLaunchKernelGGL(myKernel2, dim3(1), dim3(256), 0, stream2, 256, d_a2);
hipLaunchKernelGGL(myKernel3, dim3(1), dim3(256), 0, stream3, 256, d_a3);
hipLaunchKernelGGL(myKernel4, dim3(1), dim3(256), 0, stream4, 256, d_a4);
```



Rem: Check that the kernels modify different parts of memory to avoid data races.

With large kernels, overlapping computations may not help performance.

Stream

- 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:
```
double *h_a = NULL;
hipHostMalloc(&h_a, Nbytes);
```

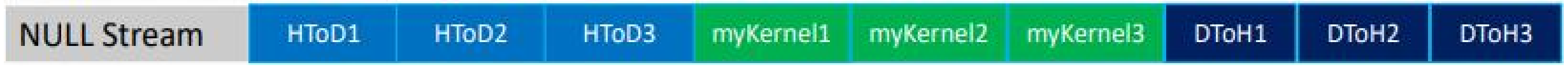
Free pinned host memory:
```
hipHostFree(h_a);
```

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.

Stream

Suppose we have 3 kernels which require moving data to and from the device:

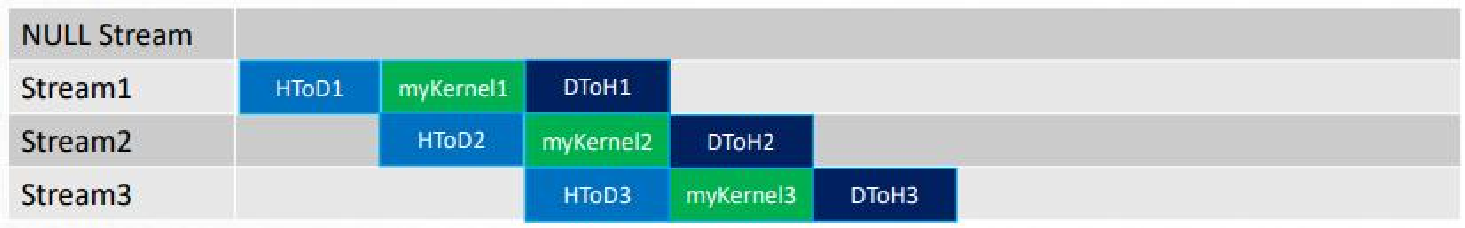
```
hipMemcpy(d_a1, h_a1, Nbytes, hipMemcpyHostToDevice));
hipMemcpy(d_a2, h_a2, Nbytes, hipMemcpyHostToDevice));
hipMemcpy(d_a3, h_a3, Nbytes, hipMemcpyHostToDevice));

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);

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



Stream

Changing to asynchronous memcpys and using streams:

```
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);
```



HIP: Example data transfer and compute

Serial

```
hipCheck( hipEventRecord(startEvent,0) );

hipCheck( hipMemcpy(d_a, a, bytes, hipMemcpyHostToDevice) );

hipLaunchKernelGGL(kernel, n/blockSize, blockSize, 0, 0, d_a, 0);

hipCheck( hipMemcpy(a, d_a, bytes, hipMemcpyDeviceToHost) );

hipCheck( hipEventRecord(stopEvent, 0) );
hipCheck( hipEventSynchronize(stopEvent) );
hipCheck( hipEventElapsedTime(&duration, startEvent, stopEvent) );
printf("Duration of sequential transfer and execute (ms): %f\n", duration);
```

HIP: How on improve the performance ?

- Use streams to overlap computation with communication

```
hipStream_t stream[nStreams];
for (int i = 0; i < nStreams; ++i) hipStreamCreate(&stream[i])
```

- Use Asynchronous data transfer

- Execute kernels on different streams

```
hipLaunchKernelGGL(some_kernel, gridsize, blocksize, shared_mem_size, stream,arg0, arg1, ...);
```

HIP: Synchronization

- Synchronize everything, could be used after each kernel launch except if you know what you are doing
  - **hipDeviceSynchronize**()
    - Heavy-duty sync point.
    - Blocks host until all work in all device streams has reported complete.

- Synchronize a specific stream Blocks host until all HIP calls are completed on this stream
  - **hipStreamSynchronize**(stream)

- Synchronize using Events
  - Create event
    - **hipEvent_t** stopEvent
    - **hipEventCreate**(&stopEvent)

  - Record an event in a specific stream and wait until is recorded
    - **hipEventRecord**(stopEvent,0)
    - **hipEventSynchronize**(stopEvent)

  - Make a stream wait for a specific event
    - **hipStreamWaitEvent**(stream[i], stopEvent, unsigned int flags)

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.

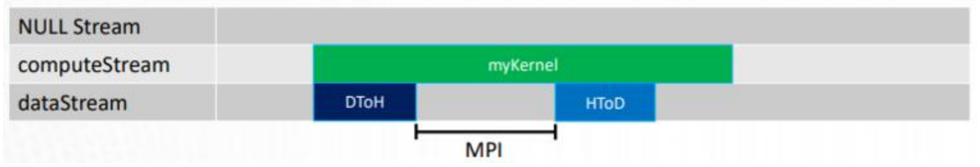
hipEventElapsedTime(&time, startEvent, endEvent);
- Returns the time in ms between when two events, startEvent and endEvent, completed
- Can be very useful for timing kernels/memcpys

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

Stream

A common use-case for streams is MPI traffic:

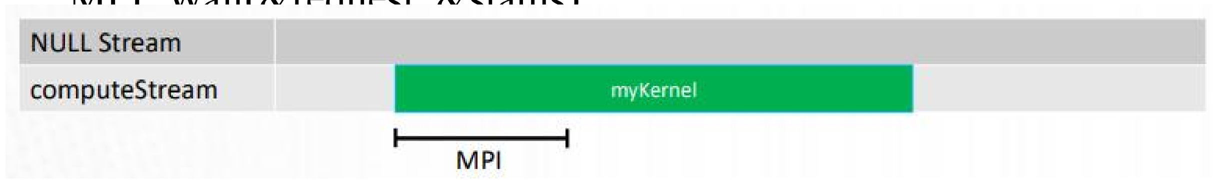
```
//Queue local compute kernel
hipLaunchKernelGGL(myKernel, blocks, threads, 0, computeStream, N, d_a);

//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
[illegible]tes, hipMemcpyHostToDevice, dataStream);
```



Stream

With a GPU-aware MPI stack, the Host<->Device traffic can be omitted:

```
//Some synchronization so that data on GPU and local compute are ready
hipDeviceSynchronize();

//Exchange data with MPI (with device pointer)
MPI_Data_Exchange(d_commBuffer, &request);

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

//Wait for MPI request to complete
MPI_Wait(&request, &status);
```



HIP: Synchronization in kernel

Code

```
__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.

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

HIP Memory allocations, access and unified memory

HIP: Memory

Memory model

Host and device have separate physical memories
It is generally not possible to call malloc() to allocate memory and access the data from the GPU
Memory management can be

- Explicit (user manages the movement of the data and makes sure CPU and GPU pointers are not mixed)
- Automatic, using Unified Memory (data movement is managed in thebackground by the Unified Memory driver)

Avoid moving data between CPU and GPU

Data copies between host and device are relatively slow
To achieve best performance, the host-device data traffic should be minimized regardless of the chosen memory management strategy

Initializing arrays on the GPU

Rather than just solving a linear equation on a GPU, also setting it up on the device

Not copying data back and forth between CPU and GPU every step or iteration can have a large performance impact!

HIP: Memory

Device memory hierarchy

- **Registers (per-thread-access)**

- Used automatically
- Size on the order of kilobytes
- Very fast access

- **Local memory (per-thread-access)**

- Used automatically if all registers are reserved
- Local memory resides in global memory
-Very slow access

- **Shared memory (per-blockaccess)**

- Usage must be explicitly programmed
- Size on the order of kilobytes
- Fast access

- **Global memory (per-deviceaccess)**

- Managed by the host through HIP API
- Size on the order of gigabytes
- Very slow access

- **There are more details in the memory hierarchy, some of which are architecture-dependent, eg,**

- Texture memory
- Constant memory

Complicates implementation
Should be considered only when a very high level of optimization is desirable

HIP: Memory

Important memory operations

- Allocate pinned device memory
  hipError_t hipMalloc(void **devPtr, size_t size)

- Allocate Unified Memory; The data is moved automatically between host/device
  hipError_t hipMallocManaged(void **devPtr, size_t size)

- Deallocate pinned device memory and Unified Memory
  hipError_t hipFree(void *devPtr)

- Copy data (host-host, host-device, device-host, device-device)
  hipError_t hipMemcpy(void *dst, const void *src, size_t count, enum hipMemcpyKind kind)

HIP: Memory

Example of explicit memory management

```
int main() {
 int *A, *d_A;
 A = (int *) malloc(N*sizeof(int));
 hipMalloc((void**)&d_A, N*sizeof(int));
 ...
 /* Copy data to GPU and launch kernel */
 hipMemcpy(d_A, A, N*sizeof(int), hipMemcpyHostToDevice);
 hipLaunchKernelGGL(...);
 ...
 hipMemcpy(A, d_A, N*sizeof(int), hipMemcpyDeviceToHost);
 hipFree(d_A);
 printf("A[0]: %d\n", A[0]);
 free(A);
 return 0;
}
```

Example of Unified Memory

```
int main() {
 int *A;
 hipMallocManaged((void**)&A, N*sizeof(int));
 ...
 /* Launch GPU kernel */
 hipLaunchKernelGGL(...);
 hipStreamSynchronize(0);
 ...
 printf("A[0]: %d\n", A[0]);
 hipFree(A);
 return 0;
}
```

HIP: Memory

✓ Unified Memory pros

Allows incremental development
Can increase developer productivity significantly
    Especially large codebases with complex data structures
Supported by the latest NVIDIA + AMD architectures
Allows oversubscribing GPU memory on some architectures

✗ Unified Memory cons

Data transfers between host and device are initially slower, but can be optimized once the code works
    Through prefetches
    Through hints
Must still obey concurrency & coherency rules, not foolproof
The performance on the AMD cards is an open question

HIP: Memory

Unified Memory workflow for GPU offloading

- Allocate memory for the arrays accessed by the GPU with hipMallocManaged() instead of malloc()
  It is a good idea to have a wrapper function or use function overloading for memory allocations

- Offload compute kernels to GPUs

- Check profiler backtrace for GPU->CPU Unified Memory page-faults

(NVIDIA Visual Profiler, Nsight Systems, AMD profiler?)

- This indicates where the data residing on the GPU is accessed by the CPU very useful for large codebases, especially if the developer is new to the code)

HIP: Memory

Unified Memory workflow for GPU offloading

- Move operations from CPU to GPU if possible, or use hints / prefetching (hipMemAdvice() / hipMemPrefetchAsync())

- It is not necessary to eliminate all page faults, but eliminating the most frequently occurring ones can provide significant performance improvements

- Allocating GPU memory can have a much higher overhead than allocating standard host memory

If GPU memory is allocated and deallocated in a loop, consider using a GPU memory pool allocator for better performance

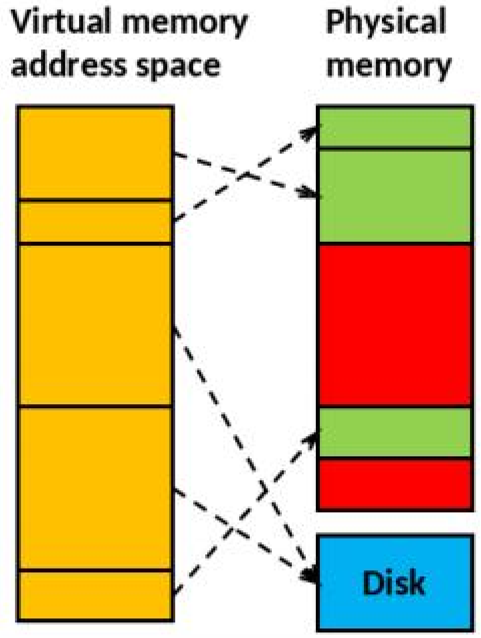
HIP: Memory

Virtual Memory addressing

- Modern operating systems utilize virtual memory

  - Memory is organized to memory pages

  - Memory pages can reside on swap area on the disk (or on the GPU with Unified Memory)



HIP: Memory

Page-locked (or pinned) memory

- Normal malloc() allows swapping and page faults
- User can page-lock an allocated memory block to a particular physical memory location
- Enables Direct Memory Access (DMA)
- Higher transfer speeds between host and device
- Copying can be interleaved with kernel execution
- Page-locking too much memory can degrade system performance due to paging problems

HIP: Memory

Allocating page-locked memory on host

- Allocated with hipHostMalloc() function instead of malloc()
- The allocation can be mapped to the device address space for device access (slow)

- On some architectures, the host pointer to device-mapped allocation can be directly used in device code (ie. it works similarly to Unified Memory pointer, but the access from the device is slow)

Deallocated using hipHostFree()

HIP: Memory

Asynchronous memcopies

- Normal hipMemcpy() calls are blocking (i.e. synchronizing)
- The execution of host code is blocked until copying is finished
- To overlap copying and program execution, asynchronous functions are required

- Such functions have Async suffix, eg. **hipMemcpyAsync()**

- User has to synchronize the program execution
- Requires page-locked memory

HIP: Memory

Global memory access in device code

- Global memory access from the device is slow
- Threads are executed in warps, memory operations are grouped in a similar fashion
- Memory access is optimized for coalesced access where threads read from and write to successive memory locations
- Exact alignment rules and performance issues depend on the architecture

HIP: Memory

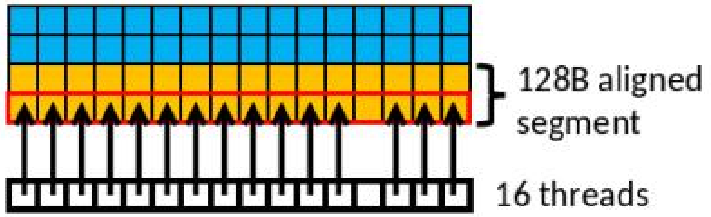
Coalesced memory access

- The global memory loads and stores consist of transactions of a certain size (eg. 32 bytes)
- If the threads within a warp access data within such a block 32 bytes, only one global memory transcation is needed
- Now, 32 threads within a warp can each read a different 4-byte integer value with just 4 transactions
- When the stride between each 4- byte integer is increased, more transactions are required (up to 32 for the worst case)!

HIP: Memory

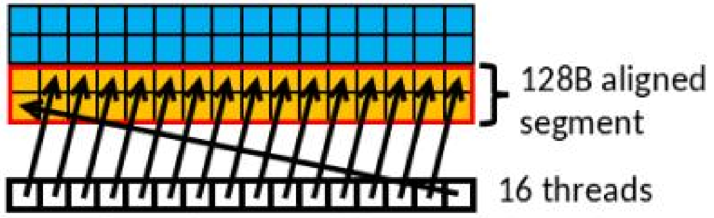
Coalesced memory access example

```
__global__ void memAccess(float *out, float *in)
{
 int tid = blockIdx.x*blockDim.x + threadIdx.x;
 if(tid != 12) out[tid + 16] = in[tid + 16];
}
```



Coalesced access

```
__global__ void memAccess(float *out, float *in)
{
 int tid = blockIdx.x*blockDim.x + threadIdx.x;
 out[tid + 1] = in[tid + 1];
}
```



Unaligned sequential access (+1 tx)

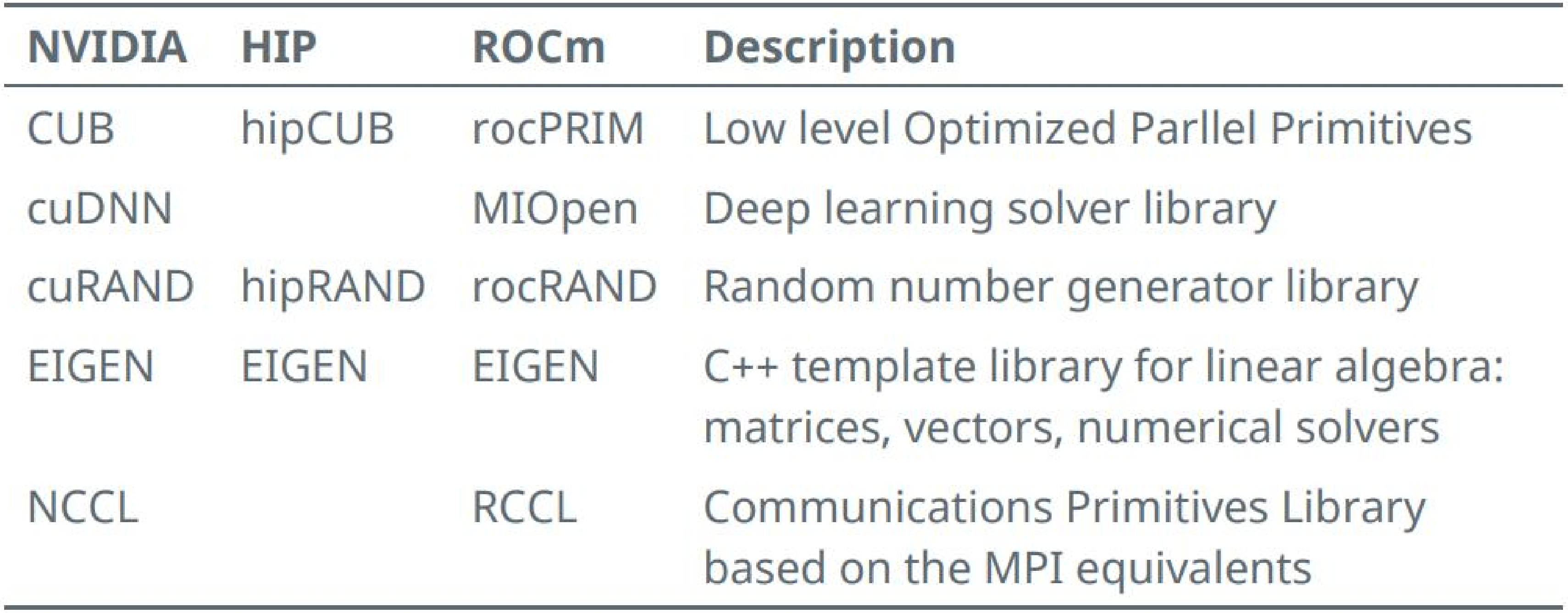
HIP Kernel optimization and profiling

HIP: Libraries

NVIDIA	HIP	ROCm	Description
cuBLAS	hipBLAS	rocBLAS	Basic Linear Algebra Subroutines
cuFFT	hipFFT	rocfft	Fast fourier Transform Library
cuSPARSE	hipSPARSE	rocSPARSE	Sparse BLAS + SMV
cuSOLVER	hipSOLVER	rocSOLVER	Lapack library
AMG-X		rocALUTION	Sparse iterative solvers and preconditioners with Geometric and Algebraic MultiGrid
Thrust	hipThrust	rocThrust	C++ parallel algorithms library

HIP: Libraries

**Librarie
s**



NVIDIA	HIP	ROCm	Description
CUB	hipCUB	rocPRIM	Low level Optimized Parllel Primitives
cuDNN		MIOpen	Deep learning solver library
cuRAND	hipRAND	rocRAND	Random number generator library
EIGEN	EIGEN	EIGEN	C++ template library for linear algebra: matrices, vectors, numerical solvers
NCCL		RCCL	Communications Primitives Library based on the MPI equivalents

HIP: hipBLAS

CUDA

```
#include <cuda_runtime.h>
#include "cublas_v2.h"

 if (cudaSuccess != cudaMalloc((void **) &a_dev,
sizeof(*a) * n * n) ||
cudaSuccess != cudaMalloc((void **) &b_dev,
sizeof(*b) * n * n) ||
cudaSuccess != cudaMalloc((void **) &c_dev,
sizeof(*c) * n * n)) {
printf("error: memory allocation (CUDA)\n");
cudaFree(a_dev); cudaFree(b_dev);
cudaFree(c_dev);
     cudaDestroy(handle);
     exit(EXIT_FAILURE);
   }
```

HIP

```
#include <hip/hip_runtime.h>
#include "hipblas.h"

if (hipSuccess != hipMalloc((void **) &a_dev,
sizeof(*a) * n * n) ||
       hipSuccess != hipMalloc((void **) &b_dev,
sizeof(*b) * n * n) ||
       hipSuccess != hipMalloc((void **) &c_dev,
sizeof(*c) * n * n)) {
      printf("error: memory allocation (CUDA)\n");

      hipFree(a_dev); hipFree(b_dev); hipFree(c_dev);
      hipblasDestroy(handle);
      exit(EXIT_FAILURE);
    }
```

HIP: Kernels

You can call a kernel with the command:

```
hipLaunchKernelGGL(kernel_name, dim3(Blocks), dim3(Threads), 0, 0, arg1, arg2, ...);
```

or

```
kernel_name<<<dim3(Blocks), dim3(Threads),0,0>>>(arg1,arg2,...);
```

where blocks are for the 3D dimensions of the grid of blocks dimensions
threads for the 3D dimentions of a block of threads
0 for bytes of dynamic LDS space
0 for stream
kernel arguments

HIP: Metrics

Various useful metrics

GPUBusy: The percentage of time GPU was busy

Wavefronts: Total wavefronts

VALUInsts: The average number of vector ALU instructions executed per work-item (affected by flow control).

VALUUtilization: The percentage of active vector ALU threads in a wave. A lower number can mean either more thread divergence in a wave or that the work-group size is not a multiple of 64. Value range: 0% (bad), 100%
(ideal - no thread divergence)

VALUBusy: The percentage of GPUTime vector ALU instructions are processed. Value range: 0% (bad) to 100% (optimal).

L2CacheHit: The percentage of fetch, write, atomic, and other instructions that hit the data in L2 cache. Value range: 0% (no hit) to 100% (optimal).

LDSBankConflict: The percentage of GPUTime LDS is stalled by bank conflicts. Value range: 0% (optimal) to 100% (bad).

Device Code, Shared Memory, and Thread Synchronization

HIP: Function Qualifiers

hipcc makes 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

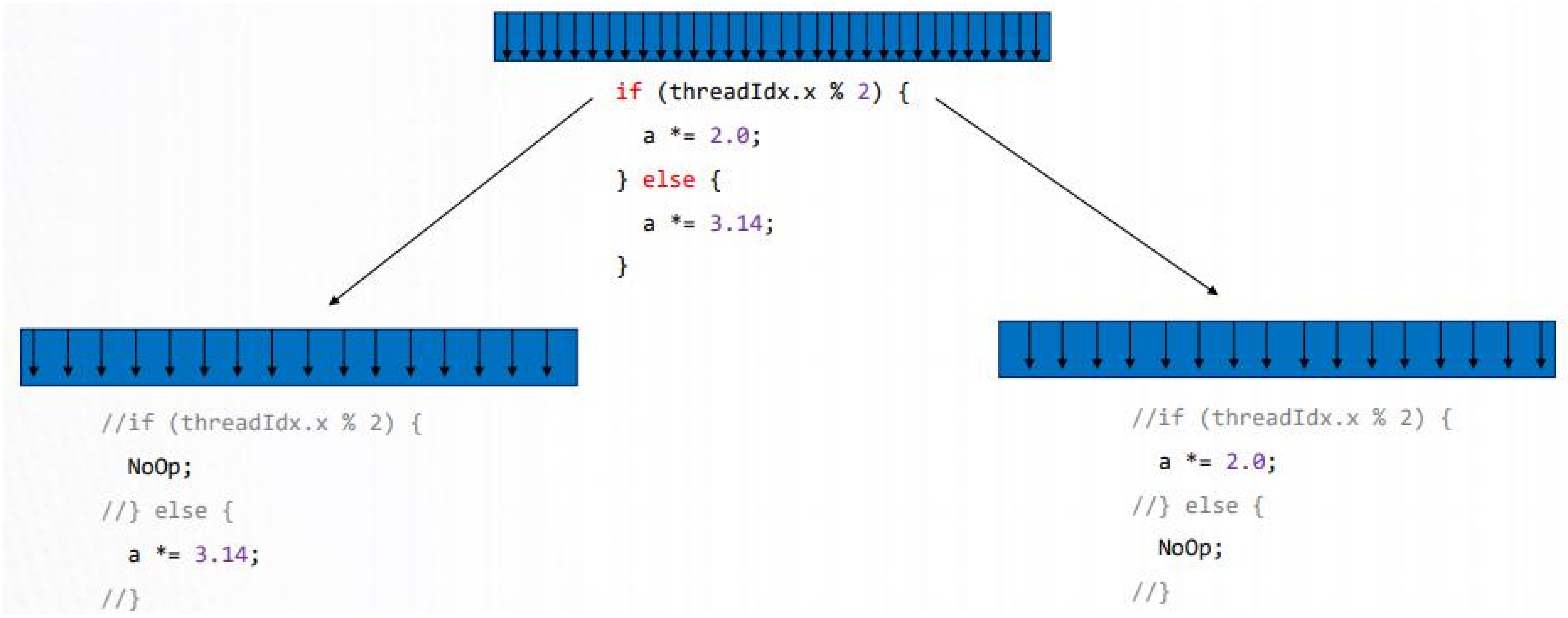
- On SIMD units, be aware of divergence.

- 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.

HIP: SIMD Execution

SIMD Execution



```
if (threadIdx.x % 2) {
  a *= 2.0;
} else {
  a *= 3.14;
}
```

```
//if (threadIdx.x % 2) {
  NoOp;
//} else {
  a *= 3.14;
//}
```

```
//if (threadIdx.x % 2) {
  a *= 2.0;
//} else {
  NoOp;
//}
```

HIP: Memory

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

HIP: Shared Memory

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.
__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
…
}
```

HIP: Thread Syncrhonization

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.

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

Rem 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

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()

HIP Multi-GPU programming and HIP+MPI

HIP: GPU Context

- Context is established when the first HIP function requiring an active context is called **hipMalloc()**

- Several processes can create contexts for a single device

- Resources are allocated per context

- By default, one context per device per process (since CUDA 4.0)
  -Threads of the same process share the primary context (for each device)

- Driver associates a number for each HIP-capable GPU starting from 0

- The function hipSetDevice() is used for selecting the desired device

HIP: Device Managment

- Return the number of hip capable devices in *count
  hipError_t hipGetDeviceCount(int *count)

- Set device as the current device for the calling host thread
  hipError_t hipSetDevice(int device)

- Return the current device for the calling host thread in *device
  hipError_t hipGetDevice(int *device)

- Reset and explicitly destroy all resources associated with the current device
  hipError_t hipDeviceReset(void)

HIP: Quering devices properties

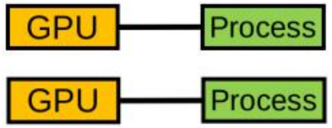
- One can query the properties of different devices in the system using hipGetDeviceProperties() function
  - No context needed
  - Provides e.g. name, amount of memory, warp size, support for unified virtual addressing, etc.
  - Useful for code portability

- Return the properties of a HIP capable device in *prop

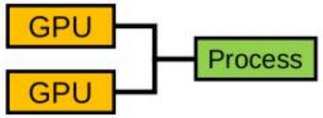
```
hipError_t hipGetDeviceProperties(struct hipDeviceProp *prop, int device)
```

HIP: Multi-GPU programsmming models

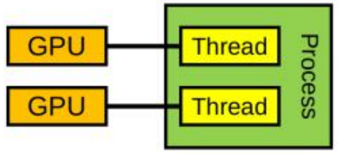
One GPU per process
- Syncing is handled through message passing (eg. MPI)



Many GPUs per process
- Process manages all context switching and syncing explicitly



One GPU per thread
- Syncing is handled through thread synchronization requirements



HIP: Multi-GPU, one GPU per process

- Recommended for multi-process applications using a message passing library
- Message passing library takes care of all GPU-GPU communication
- Each process interacts with only one GPU which makes the implementation easier and less invasive (if MPI is used anyway)
  - Apart from each process selecting a different device, the implementation looks much like a single-GPU program
- Multi-GPU implementation using MPI is discussed at the end!

HIP: Multi-GPU, many GPUs per process

- Process switches the active GPU using hipSetDevice() function

- After setting the device, HIP-calls such as the following are effective only on the selected GPU:
  - Memory allocations and copies
  - Streams and events
  - Kernel calls

- Asynchronous calls are required to overlap work across all devices

HIP: Multi-GPU, one GPU per process

- One GPU per CPU thread
  - I.e one OpenMP thread per GPU being used
- HIP API is threadsafe
  - Multiple threads can call the functions at the same time
- Each thread can create its own context on a different GPU
  - hipSetDevice() sets the device and creates a context per thread
  - Easy device management with no changing of device
- Communication between threads becomes a bit more tricky

```
#pragma omp parallel for
for(unsigned int i = 0; i < deviceCount; i++)
{
hipSetDevice(i);
kernel<<<blocks[i],threads[i]>>>(arg1[i], arg2[i]);
}
```

HIP: Peer access

Access peer GPU memory directly from another GPU
- Pass a pointer to data on GPU 1 to a kernel running on GPU 0
- Transfer data between GPUs without going through host memory
- Lower latency, higher bandwidth

Check peer accessibility

```
hipError_t hipDeviceCanAccessPeer(int* canAccessPeer, int device, int peerDevice)
```

Enable peer access

```
hipError_t hipDeviceEnablePeerAccess(int peerDevice, unsigned int flags)
```

Disable peer access

```
hipError_t hipDeviceDisablePeerAccess(int peerDevice)
```

HIP: Peer to peer communication

- Devices have separate memories
- With devices supporting unified virtual addressing, hipMemCpy() with kind=hipMemcpyDefault, works:

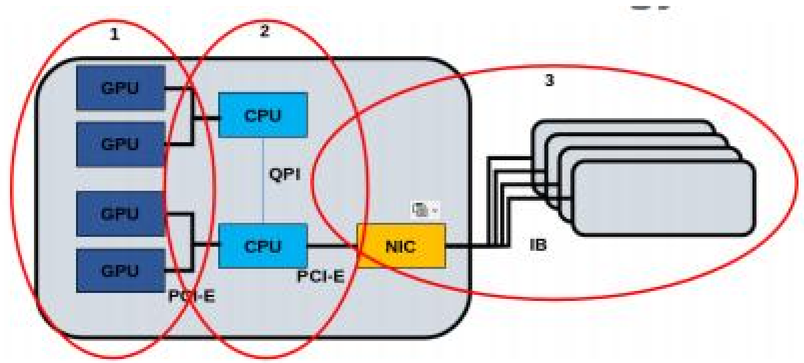
```
hipError_t hipMemcpy(void* dst, void* src, size_t count, hipMemcpyKind kind)
```

- Other option which does not require unified virtual addressing

```
hipError_t hipMemcpyPeer(void* dst, int dstDev, void* src, int srcDev, size_t count)
```

- If peer to peer access is not available, the functions result in a normal copy through host memory

HIP: Three levels of parallelism



1. GPU - GPU threads on the multiprocessors
   Parallelization strategy: HIP, SYCL, Kokkos, OpenMP
2. Node - Multiple GPUs and CPUs
   Parallelization strategy: MPI, Threads, OpenMP
3. Supercomputer - Many nodes connected with interconnect
   Parallelization strategy: MPI between nodes

HIP: Strategies

MPI+HIP strategies

1. One MPI process per node

2. One MPI process per GPU

3. Many MPI processes per GPU, only one uses it

4. Many MPI processes sharing a GPU

2 is recommended (also allows using 4 with services such as CUDA MPS)
- Typically results in most productive and least invasive implementation for an MPI program
- No need to implement GPU-GPU transfers explicitly (MPI handles all this)
- It is further possible to utilize remaining CPU cores with OpenMP (but this is not always worth the effort/increased complexity)

HIP: Strategies

Selecting the correct GPU

Typically all processes on the node can access all GPUs of that node.
The following implementation allows utilizing all GPUs using one or more processes per GPU.
-Use CUDA MPS when launching more processes than GPUs

```
int deviceCount, nodeRank;
MPI_Comm commNode;
MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL,
&commNode);
MPI_Comm_rank(commNode, &nodeRank);
hipGetDeviceCount(&deviceCount);
hipSetDevice(nodeRank % deviceCount);
```

HIP: Strategies

GPU-GPU communication through MPI

CUDA/ROCm aware MPI libraries support direct GPU-GPU transfers
- Can take a pointer to device buffer (avoids host/device data copies)

Unfortunately, currently no GPU support for custom MPI datatypes

(must use a datatype representing a contiguous block of memory)
- Data packing/unpacking must be implemented application-side on GPU

ROCm aware MPI libraries are under development and there may be problems
- It is a good idea to have a fallback option to use pinned host staging buffers

HIP: Strategies

Many options to write a multi-GPU program:

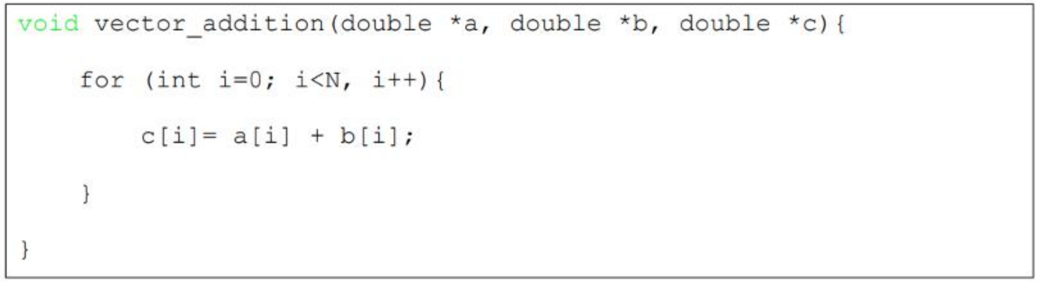
- Use hipSetDevice() to select the device, and the subsequent HIP calls operate on that device
- If you have an MPI program, it is often best to use one GPU per process, and let MPI handle data transfers between GPUs
- There is still little experience from ROCm aware MPIs, there may be issues
- Note that a CUDA/ROCm aware MPI is only required when passing device pointers to the MPI, passing only host pointers does not require any CUDA/ROCm awareness

HIP Examples

HIP: Vector Addition

Embarrassingly Parallel; each element-wise addition is completely independent from all others,
⇒ so all elements can be computed at the same time.

Serial function

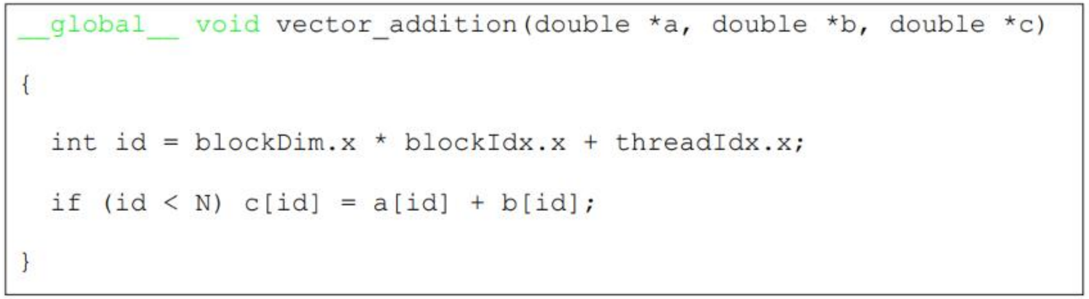


```
void vector_addition(double *a, double *b, double *c){

    for (int i=0; i<N, i++){

        c[i]= a[i] + b[i];

    }

}
```

A single process

iterates through the loop and adds the vectors element by element (sequentially)

GPU kernel



```
__global__ void vector_addition(double *a, double *b, double *c)

{

  int id = blockDim.x * blockIdx.x + threadIdx.x;

  if (id < N) c[id] = a[id] + b[id];

}
```

GPU kernel

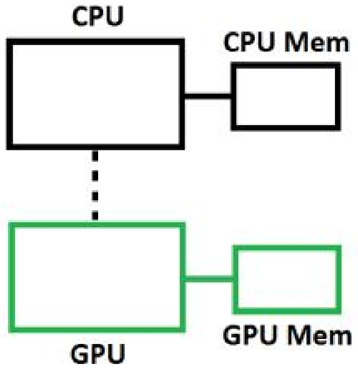
All GPU threads run same kernel function,
but each thread is assigned a unique global ID to know which element(s) to calculate.

__global__ : Indicates the function is a HIP kernel function – called by the host (CPU) and executed on the device (GPU).

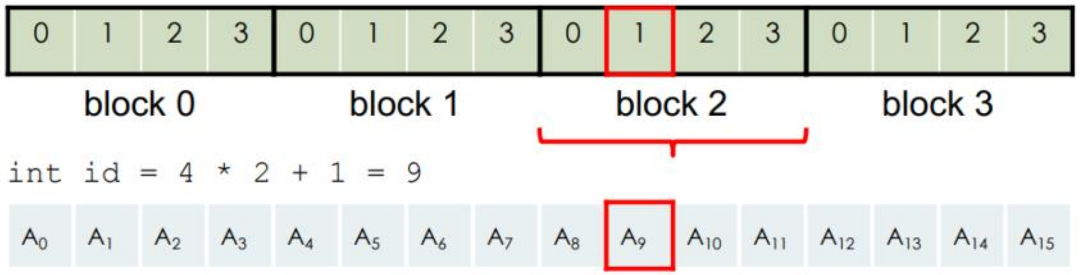
HIP: Vector Addition

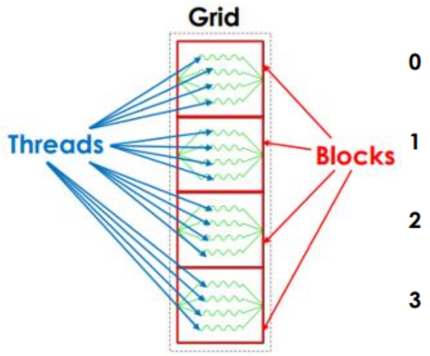
GPU kernel

```
__global__ void vector_addition(double *a, double *b, double *c)
{
                4             2              1
  int id = blockDim.x * blockIdx.x + threadIdx.x;

  if (id < N) c[id] = a[id] + b[id];
}
```



For example, with blockIdx.x = 2 and threadIdx.x = 1…





Thank you for your attention !