Using with AMD's HIP on Frontier

Noel Chalmers, Damon McDougall, Paul Bauman, Nicholas Curtis,
Nicholas Malaya, Rene van Oostrum, Noah Wolfe

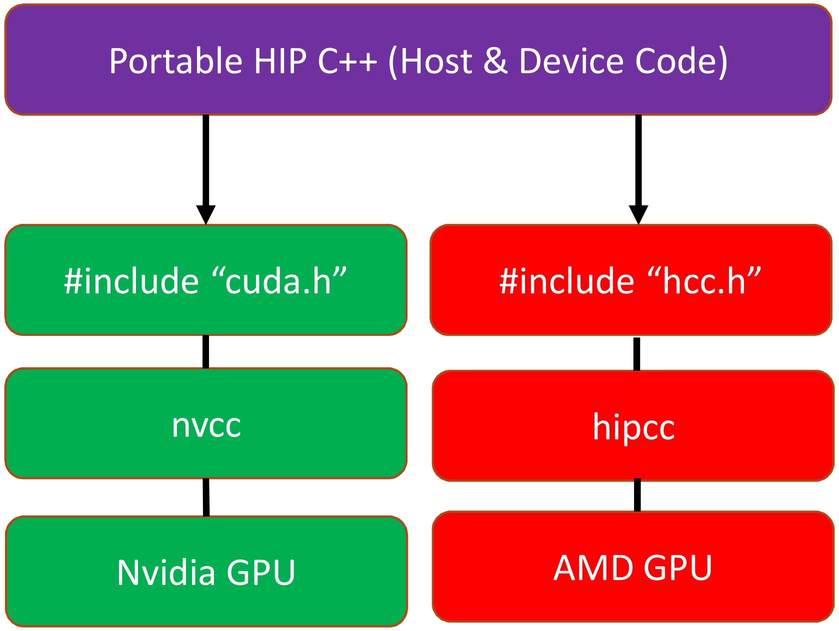
10/8/2019

Introduction to HIP

AMD's **H**eterogeneous-compute **I**nterface for **P**ortability, 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:

- 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
- Open-source
- **Currently available on Summit**



Getting started with HIP

CUDA VECTOR ADD

```
__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];
  }
}
```

HIP VECTOR ADD

```
__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

CUDA	HIP
cudaMalloc(&d_x, N*sizeof(double));	hipMalloc(&d_x, N*sizeof(double));
cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);	hipMemcpy(d_x, x, N*sizeof(double), hipMemcpyHostToDevice);
cudaDeviceSynchronize();	hipDeviceSynchronize();

Launching a kernel

CUDA KERNEL LAUNCH SYNTAX

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

HIP KERNEL LAUNCH SYNTAX

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

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

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

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);
```


AMD

Device Memory

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;
  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);
```


AMD

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

Despite differences, majority of CUDA code in applications can be simply translated.

Portability layers using HIP

Several portability layers are already supporting, or implementing, HIP

- RAJA
  - HIP kernel execution policies syntactically identical to CUDA
  - Official PRs under review
- Kokkos
  - HIP kernel execution policies syntactically identical to CUDA
  - Support is in Alpha and under development by Kokkos and AMD developers
- OCCA
  - OKL kernels can compile for HIP devices
  - Available in OCCA's master branch
- OpenMP 5.0
  - gcc and AMD's aomp compilers support target offload regions, interop with HIP

Tuning HIP Applications for Frontier

Device Management

Host can query number of devices visible to system:

```
int numDevices = 0;
hipGetDeviceCount(&numDevices);
```

Each MPI rank can select a particular device on a node:

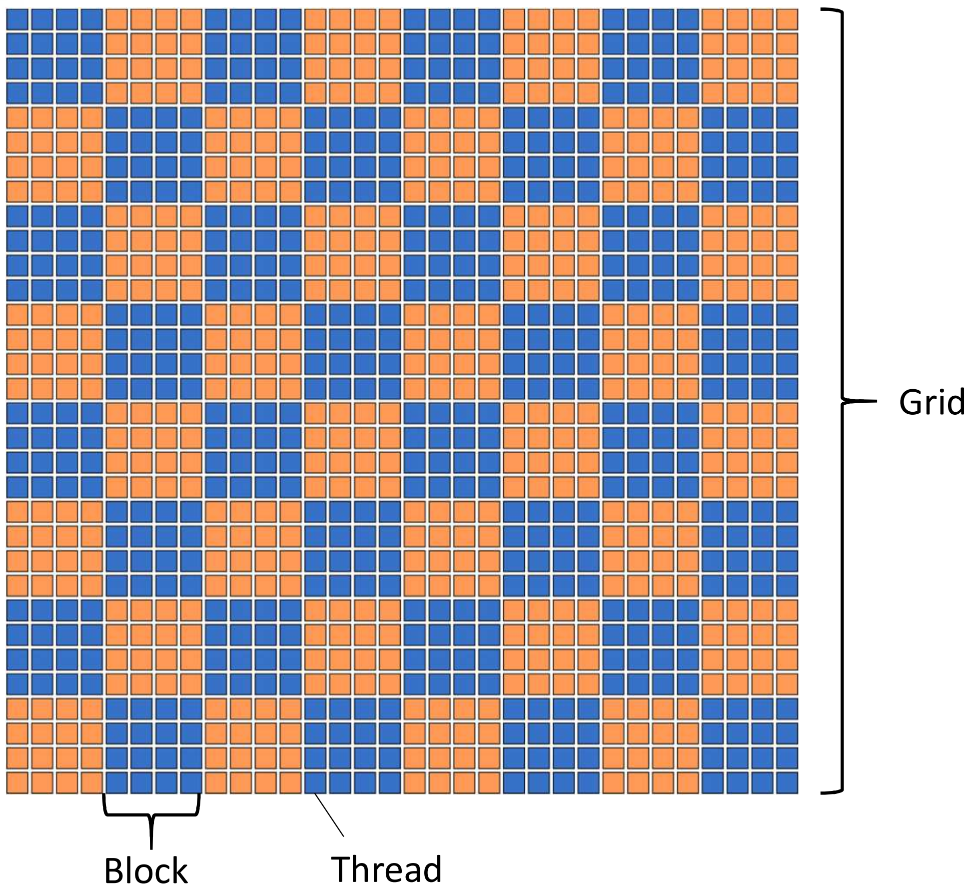
```
int rank;
MPI_Comm_rank(comm, &rank);
hipSetDevice(rank % numDevices);
```

The host can manage several devices by swapping the currently selected device during runtime.

Typical case is for each rank to manage its own GPU.

Device Kernels: The Grid

- In HIP, kernels are executed on a ”grid”
- The “grid” is what you will map your problem to
  - Your algorithm may not map to a grid, but it can be useful to think that way
- AMD devices (GPUs) support 1D, 2D, and 3D grids.
- 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 identical in HIP



SIMD operations

There is a natural mapping of blocks & threads to hardware:

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

Good practice is to make the block size a multiple of 64 and have several wavefronts (e.g. at least 256 threads)

When every CU on a GPU has many wavefronts executing, the kernel is said to have high ‘occupancy’.

SIMD Execution

After entering a kernel, all device code is executed on SIMD units.

- 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 significantly impact performance
- E.g. Both for loops are executed in order:

```
if (threadIdx.x % 2 == 0) {
  for (int i=0;i<1000;i++) d_a[id+i] *= 2.0;
else
  for (int i=0;i<1000;i++) d_a[id+i] /= 2.0;
```

Memory Hierarchy in Device Code

Several types of memory accessible in device code (Ordered generally slowest to fastest):

- Pinned Host Memory
- Unified Virtual Memory (UVM)
- Device Global Memory
- Local Data Share (LDS)
- Vector/Scalar Registers

Memory in Device Code

- Threads by default can dereference pinned host memory in device code:
  - Memory allocated by `hipHostMalloc()` (more details later)
  - Data travels over host<->device data fabric (e.g. PCIe®)
  - Access will likely be slow compared to other memory types.

- Threads can all access pointers to Unified Virtual Memory:
  - Memory allocated by `hipMallocManaged()`
  - Memory is automatically migrated between host and device by the HIP runtime
  - Can have significant overhead, even when memory is already resident on device
  - Sometimes useful to use UVM in porting process
  - **Highly recommended to migrate away from UVM usage for performance sensitive regions.**

- Threads can all access device global memory via device pointers:
  - Memory allocated by `hipMalloc()`
  - Access is slow compared to more local memory (registers and LDS)
  - Bandwidth can be significantly improved if the wavefront accesses memory in coalesced fashion (more later)

AMD

Memory in Device Code

- Stack variables declared in device code are allocated in vector registers, entries private to each thread:
  - Access is very fast
  - There is a limited amount of register space available per thread
  - If all threads in a wavefront access a common value, scalar register can be used instead

- Stack variables declared as `__shared__`:
  - Allocated in Local Data Share (LDS), a.k.a. shared memory
  - Variables are shared and accessible by all threads in the same block
  - Access is significantly (~10x) faster than device global memory (but slower than register)
  - LDS coherency often requires block-level synchronization (`__syncthreads()`)

AMD

Shared Memory Example

```
__global__ MatVec(const double *A, const double *x, double* Ax) {
  const int myrow = threadIdx.x;  //assume one block

  //Ax = A*x
  double r_Ax = 0.0; // accumulate answer in register
  for (int i=0; i<512; i++) {
    r_Ax += A[i+512*myrow]*x[i];
  }

  //write out result
  Ax[myrow] = r_Ax;
}
```

- Each thread streams through its row of the matrix
- Each thread uses all the values in the x vector
- If we put x in shared memory, can load it once and all threads can re-use it.

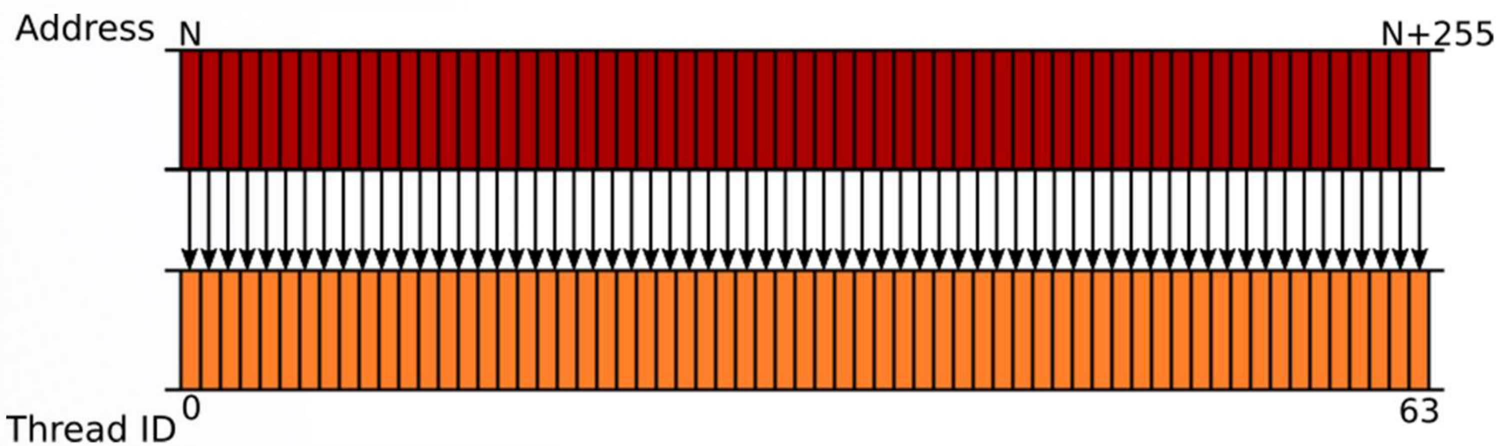
Shared Memory Example

```
__global__ MatVec(const double *A, const double *x, double* Ax) {
  const int myrow = threadIdx.x;  // assume one block

  __shared__ double s_x[512];
  if (myrow < 512) s_x[myrow] = x[myrow];

  __syncthreads(); // ensures all of s_x has been loaded

  //Ax = A*x
  double r_Ax = 0.0; // accumulate answer in register
  for (int i=0; i<512; i++) {
    r_Ax += A[i+512*myrow]*s_x[i];
  }

  //write out result
  Ax[myrow] = r_Ax;
}
```

Coalesced Memory Access

When accessing device global memory on AMD GPUs, bandwidth may be significantly increased if the access is **coalesced** across the wavefront.

Coalesced access means **consecutive threads in a wavefront access consecutive memory locations**

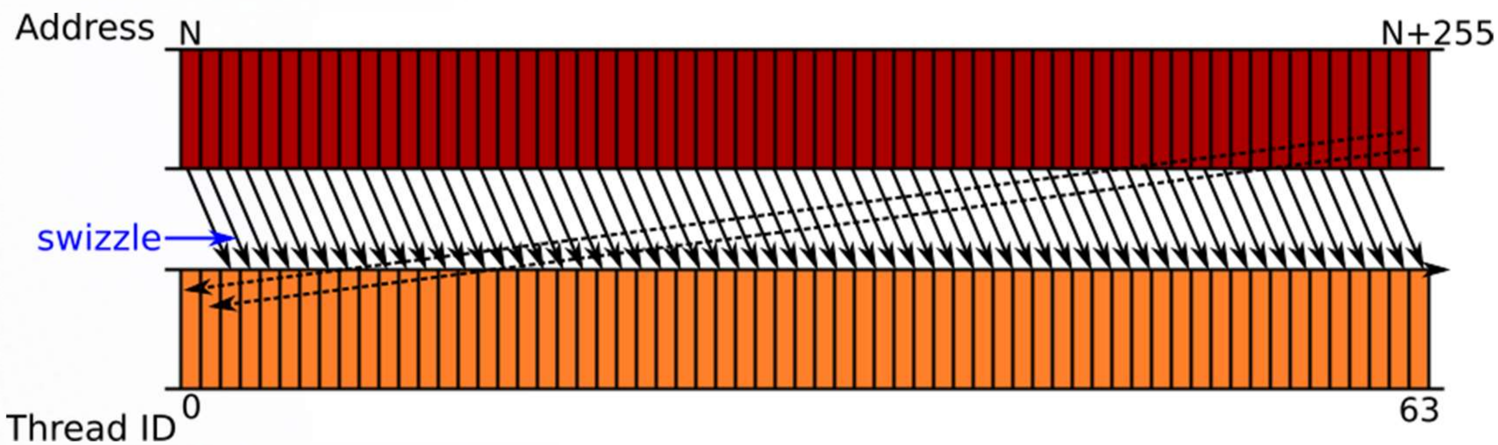


Coalesced Memory Access

Coalescing of a wavefront's memory access occurs at the hardware level.

Whenever possible, the memory controller turns the whole wavefront's request into a single coalesced memory request.

As a result, the wavefront's coalesced access could be swizzled, but still be coalesced by the memory controller.



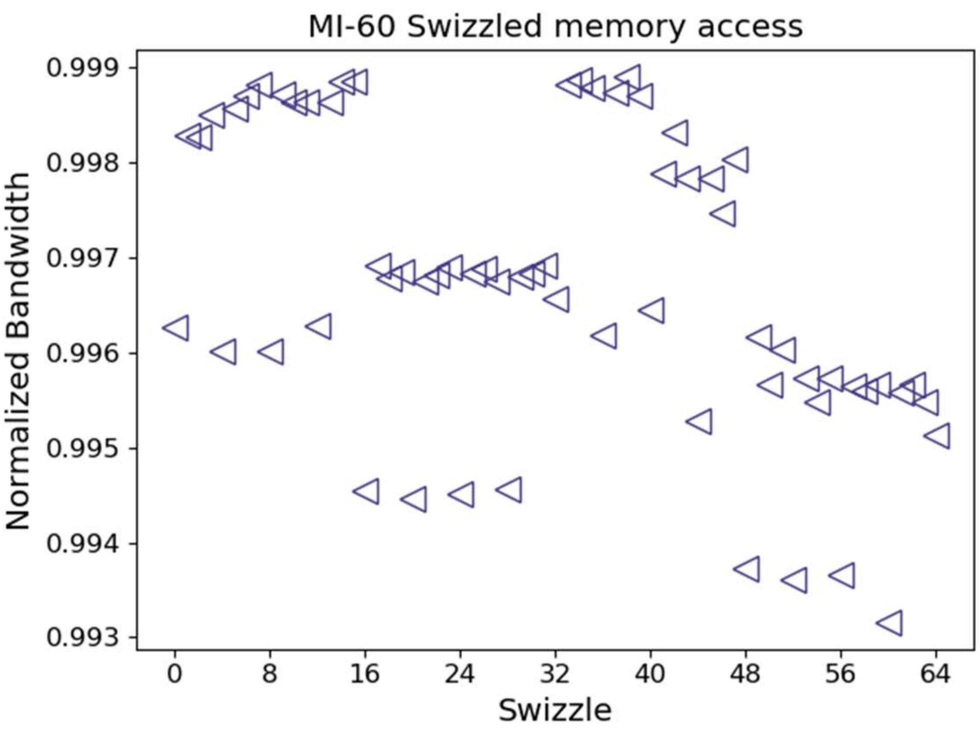
Coalesced Memory Access

Experiment on wavefront coalescing:

- Kernel doing loads/stores of 64 floats in each wavefront
- Order of access is swizzled at the wavefront level

```
a[id + threadIdx.x] = b[id + (threadIdx.x
                        + swizzle)%64];
```

- Results show bandwidth insensitive to swizzling (e.g., changing the order of access along the wavefront)
- Max 1% performance drop in relative bandwidth.
- Performance drop even less noticeable with more data movement in kernel (e.g. repeated reads/writes).

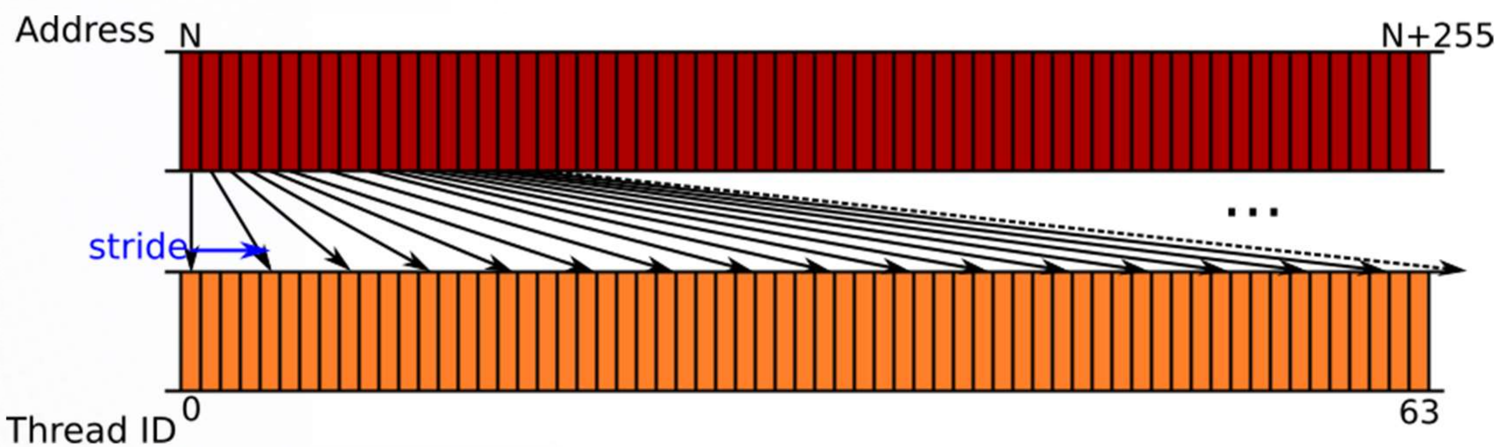


AMD

Strided Memory Access

A common access pattern is a strided memory access within a wavefront

- Thread 0 loads a value from address A, thread 1 from address A+1*stride, thread 2 from A+2*stride, etc.
- Common in structured grid problems
- Very common when using array-of-structures, rather than structures-of-arrays
- Can have **severe** impact on achieved bandwidth



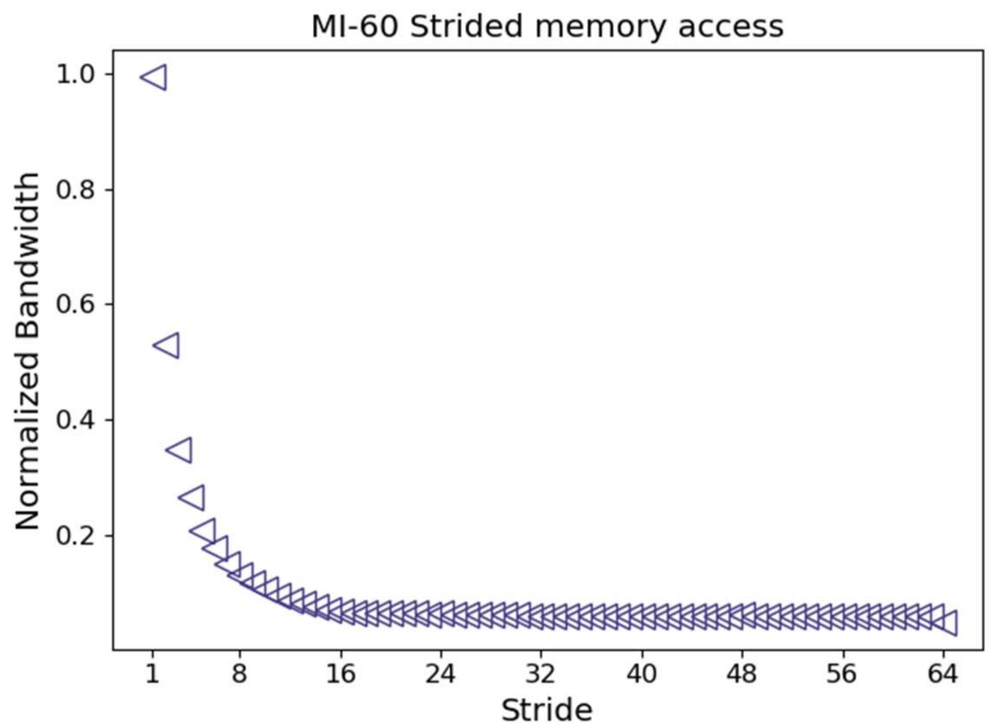
Strided Memory Access

Experiment on strided access:

- Kernel doing loads/stores of 64 floats in each wavefront
- Access is strided

```
a[id + threadIdx.x] = b[(id + threadIdx.x)
                         *stride];
```

- Stride = 1 corresponds to coalesced access (peak bandwidth)
- Stride = 2 immediate degrades bandwidth to near 50% of peak.
- By stride = 16, **a separate cache line must be loaded for each thread's memory request**



AMD

Coalesced Memory Example

```
__global__ MatVec(const double *A, const double *x, double* Ax) {
  const int myrow = threadIdx.x;  // assume one block

  __shared__ double s_x[512];
  if (myrow < 512) s_x[myrow] = x[myrow];

  __syncthreads(); // ensures all of s_x has been loaded

  //Ax = A*x
  double r_Ax = 0.0; // accumulate answer in register
  for (int i=0; i<512; i++) {
    r_Ax += A[i+512*myrow]*s_x[i];
  }

  //write out result
  Ax[myrow] = r_Ax;
}
```

- Strided access for the matrix entries
- Better to store A in column-major format

Coalesced Memory Example

```
__global__ MatVec(const double *A, const double *x, double* Ax) {
  const int myrow = threadIdx.x;  // assume one block

  __shared__ double s_x[512];
  if (myrow < 512) s_x[myrow] = x[myrow];

  __syncthreads(); // ensures all of s_x has been loaded

  //Ax = A*x
  double r_Ax = 0.0; // accumulate answer in register
  for (int i=0; i<512; i++) {
    r_Ax += A[i*numRows+myrow]*s_x[i];
  }

  //write out result
  Ax[myrow] = r_Ax;
}
```

Asynchronous computing with HIP

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 is safe to access/modify 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.

AMD

Streams

- A stream in HIP is a queue of tasks (e.g. kernels, memcpys, events).
  - Tasks enqueued in a stream **must 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 special stream called the 'NULL Stream':
  - This stream is special
  - No task on the NULL stream will begin until **all previously enqueued tasks in all other streams have completed**.
  - Blocking calls like hipMemcpy always run on the NULL stream.

Streams

- 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);
```

NULL Stream			
Stream1		myKernel1	
Stream2		myKernel2	
Stream3		myKernel3	
Stream4		myKernel4	

Note 1: Be sure 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
  - Device->Device memcpys
  - Compute kernels.

- These different operations can overlap without dividing the GPU's resources.
  - The overlapping operations must be in separate, non-NULL, streams.
  - Any host memory must be **pinned.**
    - Malloc'd with `hipHostMalloc()`
    - This also significantly increases regular Host<->Device memcpy bandwidth

Pinned Host 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.

Streams

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

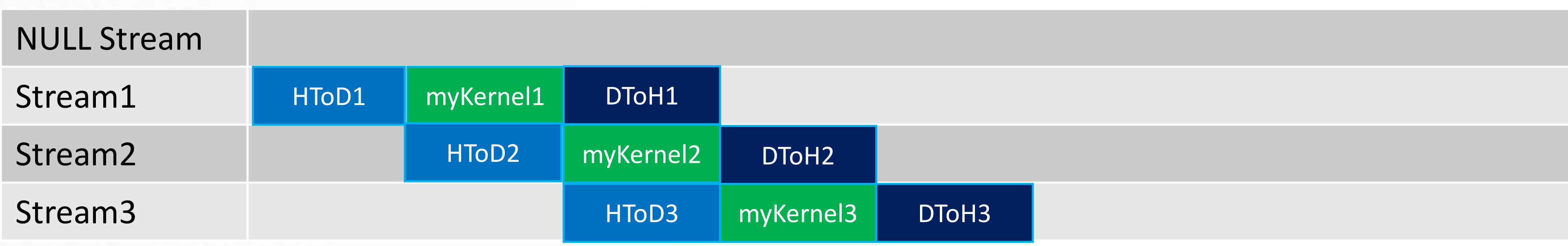
```
hipMemcpy(d_a1, h_a1, Nbytes, hipMemcpyHostToDevice));
hipMemcpy(d_a2, h_a2, Nbytes, hipMemcpyHostToDevice));
hipMemcpy(d_a3, h_a3, Nbytes, hipMemcpyHostToDevice));

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);
```

NULL Stream	HToD1	HToD2	HToD3	myKernel1	myKernel2	myKernel3	DToH1	DToH2	DToH3

Streams

Changing to asynchronous memcpys and using streams:

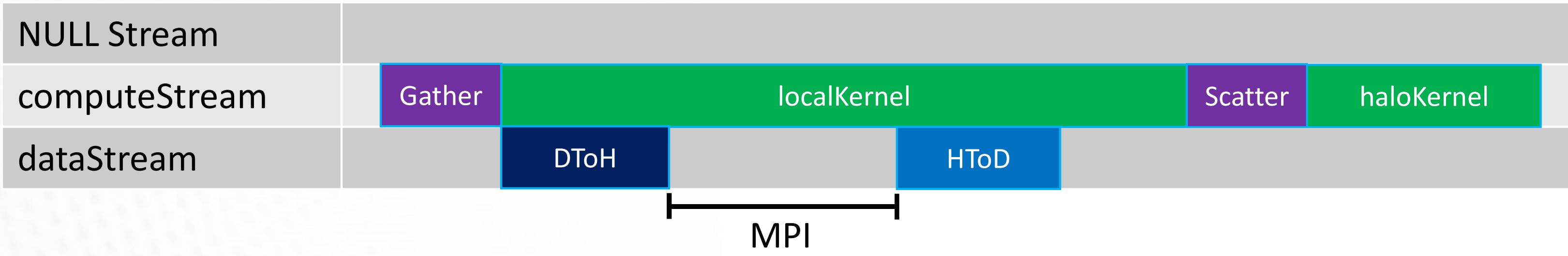
```
hipMemcpyAsync(d_a1, h_a1, Nbytes, hipMemcpyHostToDevice, stream1);
hipMemcpyAsync(d_a2, h_a2, Nbytes, hipMemcpyHostToDevice, stream2);
hipMemcpyAsync(d_a3, h_a3, Nbytes, hipMemcpyHostToDevice, stream3);

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);
```



Streams

A common use-case for streams is MPI traffic:

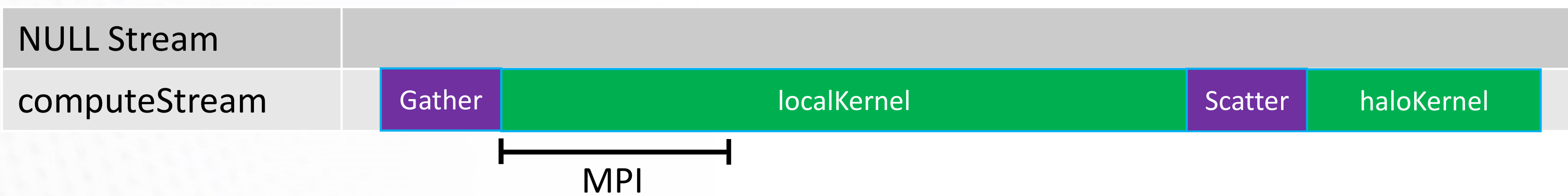
```
hipLaunchKernelGGL(haloGather, blocks, threads, 0, computeStream, N, d_a, d_commBuffer); //Gather halo data
hipStreamSynchronize(computeStream); //Wait for gather to complete

hipLaunchKernelGGL(localKernel, blocks, threads, 0, computeStream, N, d_a); //Local computation
hipMemcpyAsync(d_commBuffer, h_commBuffer, Nbytes, hipMemcpyDeviceToHost, dataStream); //copy to host
hipStreamSynchronize(dataStream); //Wait for data to arrive

MPI_Data_Exchange(h_commBuffer); //Exchange data with MPI

hipMemcpyAsync(h_commBuffer, d_commBuffer, Nbytes, hipMemcpyHostToDevice, dataStream); //copy back to device
hipStreamSynchronize(dataStream); //Wait for data to arrive

hipLaunchKernelGGL(haloScatter, blocks, threads, 0, computeStream, N, d_a, d_commBuffer); //Scatter halo data
hipLaunchKernelGGL(haloKernel, blocks, threads, 0, computeStream, N, d_a); //Halo computation
```



Streams

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

```
hipLaunchKernelGGL(haloGather, blocks, threads, 0, computeStream, N, d_a, d_commBuffer); //Gather halo data
hipEventRecord(gatherEvent, computeStream); //Record end of gather

hipLaunchKernelGGL(localKernel, blocks, threads, 0, computeStream, N, d_a); //Queue Local computation

hipEventSynchronize(gatherEvent); //Wait for gather kernel to complete

MPI_Data_Exchange(d_commBuffer); //Exchange data with MPI (using device buffer)

hipLaunchKernelGGL(haloScatter, blocks, threads, 0, computeStream, N, d_a, d_commBuffer); //Scatter halo data
hipLaunchKernelGGL(haloKernel, blocks, threads, 0, computeStream, N, d_a); //Halo computation
```



AMD

Host/Device Synchronization

To avoid idle time on host and/or device, be aware of how and when the host is synchronizing with the devices' streams:

- `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.

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

- `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.

Summary of Optimization Tips

- Multiple wavefronts per CU (i.e. high occupancy) important to latency hiding and instruction throughput
  - High register usage and/or LDS usage can reduce CU occupancy
  - LDS access is O(10) times faster than global memory
  - LDS usage can improve overall bandwidth, often worth the occupancy reduction
  - High occupancy is not a silver bullet
- Unified virtual memory is useful for ease of porting, but should be phased out ASAP for performance
- Memory coalescing dramatically increases bandwidth of load/store to LDS and global memory
- Reordering instructions to prefetch data to registers can help the scheduler issue loads earlier
- Unrolling loops allows compiler and scheduler to issue many loads/stores at once
  - May reduce occupancy
  - Register space spills to L1 cache, then to L2 cache, then to global device memory
- Important to issue enough work to fill all CUs
  - Many small kernels can suffer launch latency overheads

- **Important to shift application from being GPU-*accelerated* to GPU-*resident***

Optimization Tips (Advanced)

- AMD's GCN assembly code (ISA) is completely open
  - https://developer.amd.com/resources/developer-guides-manuals/
- To inspect GPU kernel assembly code, you can run `extractkernel` on your binary
  - Should obtain a .isa file
  - Can also set `KMDUMPISA=1` at link time to extract the .isa automatically

  - s_* : Scalar unit instructions
  - v_* : SIMD unit instructions
  - global_* : Global memory load/store
  - ds_* : LDS memory load/store
- Lots of preamble data to check register use
- Can check things like #pragma unroll effects in your kernel assembly

```
; %bb.0:                                          ; %entry
    s_load_dword s9, s[4:5], 0x4
    s_load_dword s4, s[4:5], 0xc
    s_load_dwordx2 s[0:1], s[6:7], 0x8
    s_load_dwordx2 s[2:3], s[6:7], 0x10
    s_waitcnt lgkmcnt(0)
    s_and_b32 s5, s9, 0xffff
    s_mul_i32 s6, s8, s5
    s_sub_i32 s4, s4, s6
    s_min_u32 s4, s4, s5
    s_mul_i32 s4, s4, s8
    v_add_u32_e32 v0, s4, v0
    v_ashrrev_i32_e32 v1, 31, v0
    v_lshlrev_b64 v[0:1], 3, v[0:1]
    v_mov_b32_e32 v3, s1
    v_add_co_u32_e32 v2, vcc, s0, v0
    v_addc_co_u32_e32 v3, vcc, v3, v1, vcc
    global_load_dwordx2 v[2:3], v[2:3], off
    v_mov_b32_e32 v4, s3
    v_add_co_u32_e32 v0, vcc, s2, v0
    v_addc_co_u32_e32 v1, vcc, v4, v1, vcc
    s_waitcnt vmcnt(0)
    global_store_dwordx2 v[0:1], v[2:3], off
    s_endpgm
```

QUESTIONS?

DISCLAIMER

Kernel time with events

Finally, another useful feature of streams is kernel timing with events:

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

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

And queued into a stream via:

```
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’.

Once completed, we can measure the time between two events:

```
hipEventElapsedTime(&time, startEvent, endEvent);
```

- Returns the time in ms between when two events, startEvent and endEvent, completed
- Very useful for timing kernels/memcpys

Note on 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
- Guaranteed to be performed in a conflict-free manner

- AMD GPUs support atomic operations on 32-bit integers in hardware
  - Float /double atomics are currently 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__`

AMD

Atomic Operations

Currently supported atomic operations in HIP:

Operation	Type, T	Notes
`T atomicAdd(T* address, T val)`	int, long long int, float, double	Adds val to *address
`T atomicExch(T* address, T val)`	int, long long int, float	Replace *address with val and return old value
`T atomicMin(T* address, T val)`	int, long long int	Replaces *address if val is smaller
`T atomicMax(T* address, T val)`	int, long long int	Replaces *address if val is larger
`T atomicAnd(T* address, T val)`	int, long long int	Bitwise AND between *address and val
`T atomicOr(T* address, T val)`	int, long long int	Bitwise OR between *address and val
`T atomicXor(T* address, T val)`	int, long long int	Bitwise XOR between *address and val