Introduction to Parallel & Distributed Computing

OpenCL: memory & threads

Lecture 12, Spring 2014

Instructor: 罗国杰

gluo@pku.edu.cn

In this Lecture …

- **Example: image rotation**
- **GPU threads and scheduling**
- **Understanding GPU memory**

Example: Image Rotation

- **Using OpenCL buffers**
  - **Declaring buffers**
  - **Enqueue reading and writing of buffers**
- **Simple but complete examples**
  - **Image Rotation**
  - **Non-blocking Matrix Multiplication**

Creating OpenCL Buffers

- Data used by OpenCL devices is stored in a "buffer" on the device
- An OpenCL buffer object is created using the following function

```
cl_mem  bufferobj = clCreateBuffer (
            cl_context context,          //Context name
            cl_mem_flags flags,          //Memory flags
            size_t size,                 //Memory size allocated in buffer
            void *host_ptr,              //Host data
            cl_int *errcode)             //Returned error code
```

- Data can implicitly be copied to the device using a host pointer parameter
  - In this case copy to device is invoked when kernel is enqueued

Memory Flags

- **Memory flag field in clCreateBuffer() allows us to define characteristics of the buffer object**

Memory Flag	Behavior
CL_MEM_READ_WRITE	Specifies memory read / write behavior
CL_MEM_WRITE_ONLY	
CL_MEM_READ_ONLY	
CL_MEM_USE_HOST_PTR	Implementations can cache the contents pointed to by host_ptr in device memory. This cached copy can be used when kernels are executed on a device.
CL_MEM_ALLOC_HOST_PTR	Specifies to the implementation to allocate memory from host accessible memory.
CL_MEM_COPY_HOST_PTR	Specifies to allocate memory for the object and copy the data from memory referenced by host_ptr.

Copying Buffers to Device

- **clEnqueueWriteBuffer() is used to write a buffer object to device memory (from the host)**
- **Provides more control over copy process than using host pointer functionality of clCreateBuffer()**
  - **Allows waiting for events and blocking**

```
cl_int clEnqueueWriteBuffer (
        cl_command_queue queue,                 //Command queue to device
        cl_mem buffer,                          //OpenCL Buffer Object
        cl_bool blocking_read,                  //Blocking/Non-Blocking Flag
        size_t offset,                          //Offset into buffer to write to
        size_t cb,                              //Size of data
        void *ptr,                              //Host pointer
        cl_uint num_in_wait_list,               //Number of events in wait list
        const cl_event *  event_wait_list,      //Array of events to wait for
        cl_event *event)                        //Event handler for this function
```

Copying Buffers to Host

- **clEnqueueReadBuffer() is used to read from a buffer object from device to host memory**
- **Similar to clEnqueueWriteBuffer()**

```
cl_int clEnqueueReadBuffer (
        cl_command_queue queue,                  //Command queue to device
        cl_mem buffer,                           //OpenCL Buffer Object
        cl_bool blocking_read,                   //Blocking/Non-Blocking Flag
        size_t offset,                           //Offset to copy from
        size_t cb,                               //Size of data
        void *ptr,                               //Host pointer
        cl_uint num_in_wait_list,                //Number of events in wait list
        const cl_event *  event_wait_list,       //Array of events to wait for
        cl_event *event)                         //Event handler for this function
```

Example: Image Rotation

- **A common image processing routine**
  - **Applications in matching, alignment, etc.**
- **New coordinates of point (x_1,y_1) when rotated by an angle Θ around (x_0,y_0)**

$$x_2 = \cos(\theta) * (x_1 - x_0) - \sin(\theta) * (y_1 - y_0) + x_0$$
$$y_2 = \sin(\theta) * (x_1 - x_0) + \cos(\theta) * (y_1 - y_0) + x_0$$

- **By rotating the image about the origin (0,0) we get**

$$x_2 = \cos(\theta) * (x_1) - \sin(\theta) * (y_1)$$
$$y_2 = \sin(\theta) * (x_1) + \cos(\theta) * (y_1)$$

- **Each coordinate for every point in the image can be calculated independently**

Original Image

Rotated Image (90º)

Image Rotation

- **Input: To copy to device**
  - Image (2D Matrix of floats)
  - Rotation parameters
  - Image dimensions
- **Output: From device**
  - Rotated Image
- **Main Steps**
  - Copy image to device by enqueueing a write to a buffer on the device from the host
  - Run the Image rotation kernel on input image
  - Copy output image to host by enqueueing a read from a buffer on the device

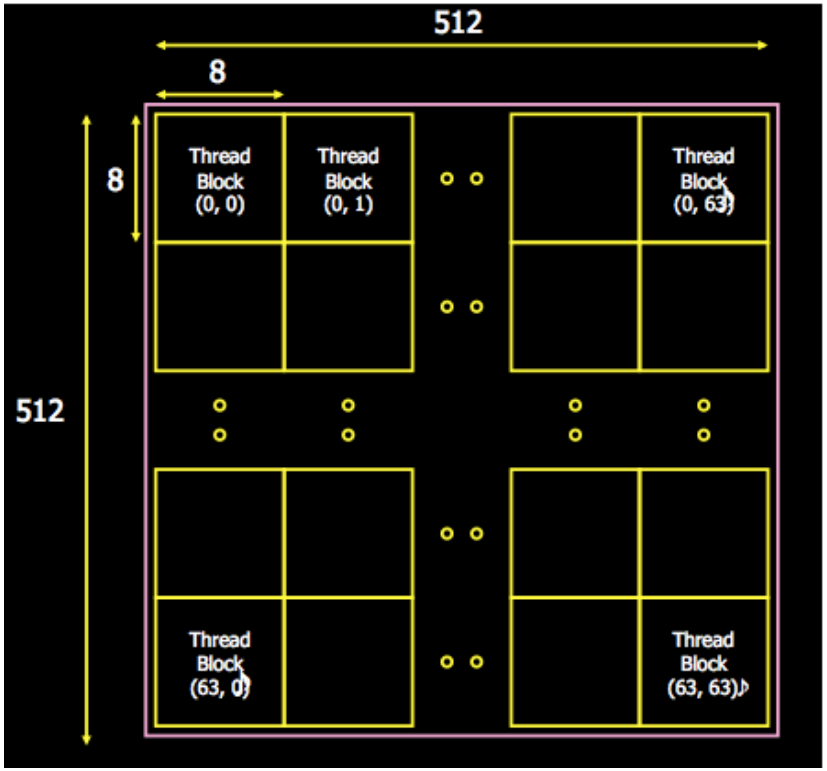


Image Rotation: The OpenCL Kernel

- **Parallel portion of the algorithm off-loaded to device**
  - **Most thought provoking part of coding process**
- **Steps to be done in Image Rotation kernel**
  - **Obtain coordinates of work item in work group**
  - **Read rotation parameters**
  - **Calculate destination coordinates**
  - **Read input and write rotated output at calculated coordinates**
- **Parallel kernel is not always this obvious.**
  - **Profiling of an application is often necessary to find the bottlenecks and locate the data parallelism**
- **In this example grid of output image decomposed into work items**
  -

Image Rotation: OpenCL Kernel

```
__kernel  void image_rotate(
            __global float * src_data, __global float * dest_data,    //Data in global
memory
            int W,    int H,                                          //Image Dimensions
            float sinTheta, float cosTheta )                          //Rotation Parameters
{
            //Thread gets its index within index space
            const int ix = get_global_id(0);
            const int iy = get_global_id(1);

            //Calculate location of data to move into ix and iy– Output decomposition as
mentioned
            float xpos = (  ((float) ix)*cosTheta + ((float)iy )*sinTheta);
            float ypos = (  ((float) iy)*cosTheta -  ((float)ix)*sinTheta);

            if ((          ((int)xpos>=0) && ((int)xpos< W)))          //Bound Checking
                        && (((int)ypos>=0) && ((int)ypos< H)))
            {
                        //Read (xpos,ypos) src_data and store at (ix,iy) in dest_data
                        dest_data[iy*W+ix]=
                                    src_data[(int)(floor(ypos*W+xpos))];
             }
}
```

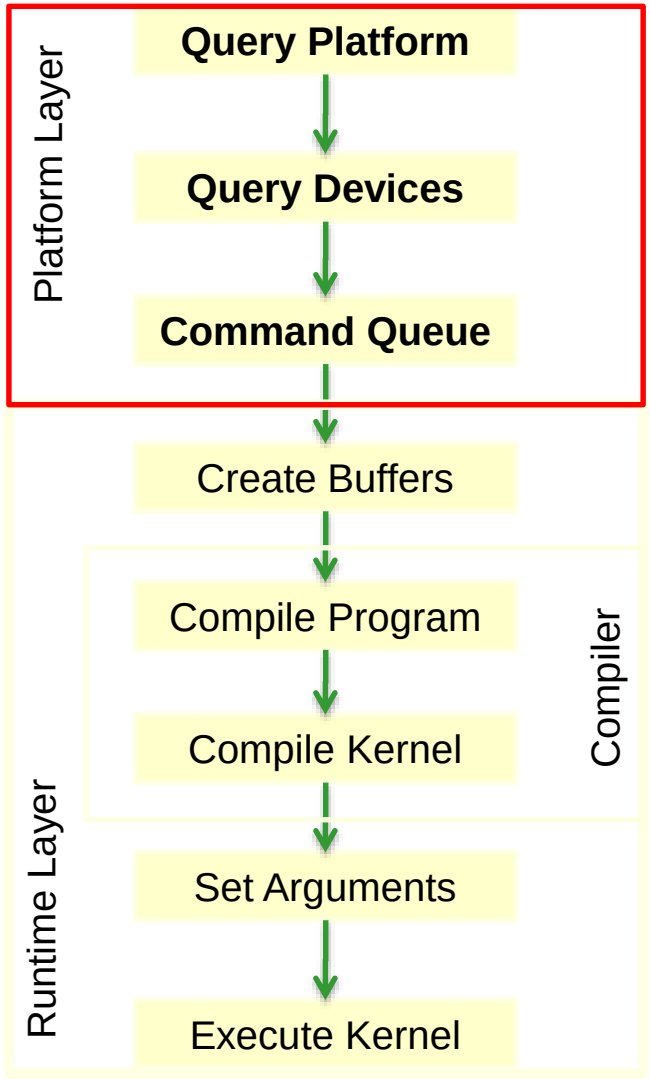
Step0: Initialize Device

- **Declare context**
- **Choose a device from context**
- **Using device and context create a command queue**

```
cl_context myctx = clCreateContextFromType (
                0, CL_DEVICE_TYPE_GPU,
                        NULL, NULL, &ciErrNum);
```

```
ciErrNum = clGetDeviceIDs (0,
                CL_DEVICE_TYPE_GPU,
                1, &device,  cl_uint *num_devices)
```

```
cl_commandqueue myqueue ;
myqueue = clCreateCommandQueue(
                        myctx, device, 0, &ciErrNum);
```



Step1: Create Buffers

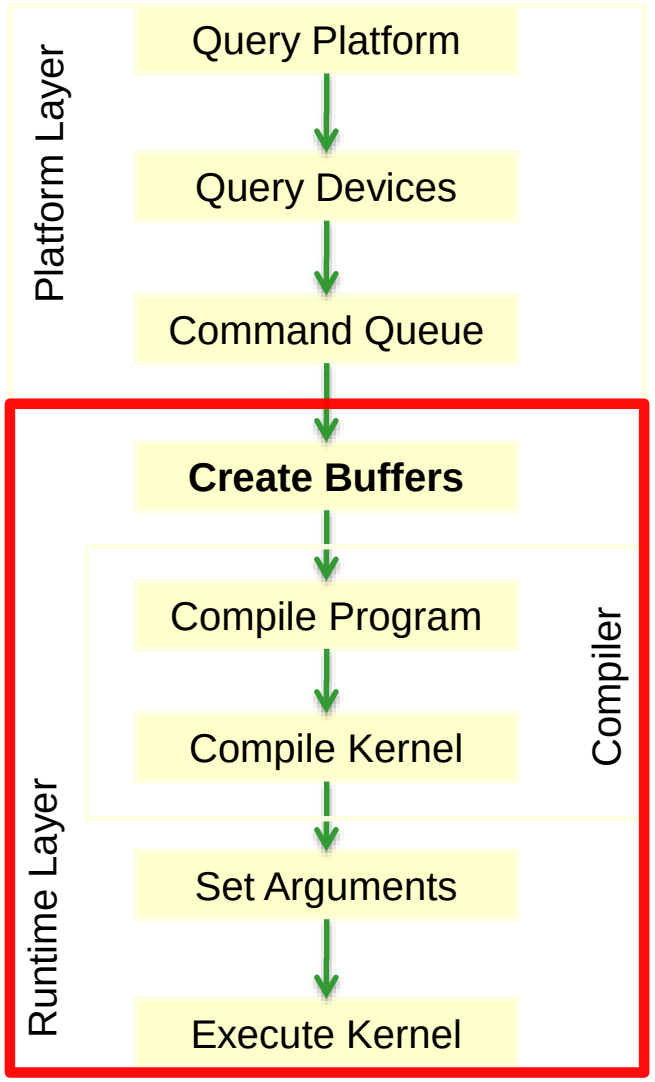
- **Create buffers on device**
  - **Input data is read-only**
  - **Output data is write-only**

```
cl_mem d_ip = clCreateBuffer(
              myctx, CL_MEM_READ_ONLY,
              mem_size,
              NULL, &ciErrNum);
```

```
cl_mem d_op = clCreateBuffer(
              myctx, CL_MEM_WRITE_ONLY,
              mem_size,
              NULL, &ciErrNum);
```

- **Transfer input data to the device**

```
ciErrNum = clEnqueueWriteBuffer (
              myqueue , d_ip, CL_TRUE,
              0, mem_size, (void *)src_image,
              0, NULL,  NULL)
```

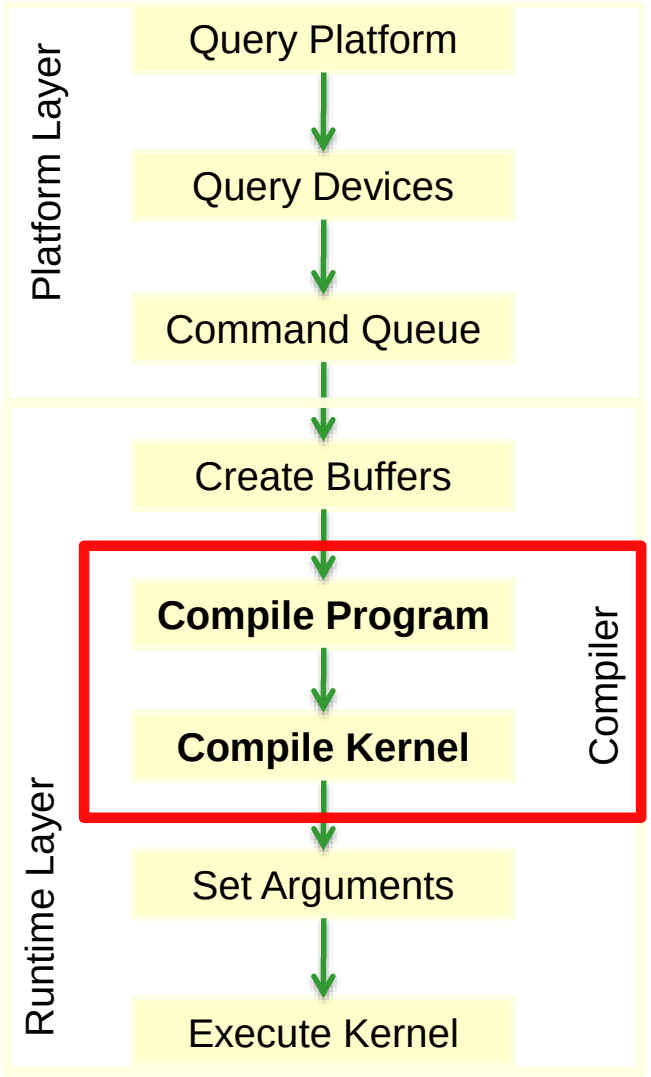


Step2: Build Program, Select Kernel

```
// create the program
cl_program myprog = clCreateProgramWithSource
                    ( myctx,1, (const char **)&source,
                    &program_length, &ciErrNum);
```

```
// build the program
ciErrNum = clBuildProgram( myprog, 0,
                           NULL, NULL, NULL, NULL);
```

```
//Use the “image_rotate” function as the kernel
cl_kernel mykernel = clCreateKernel (
                           myprog , “image_rotate” ,
                           error_code)
```

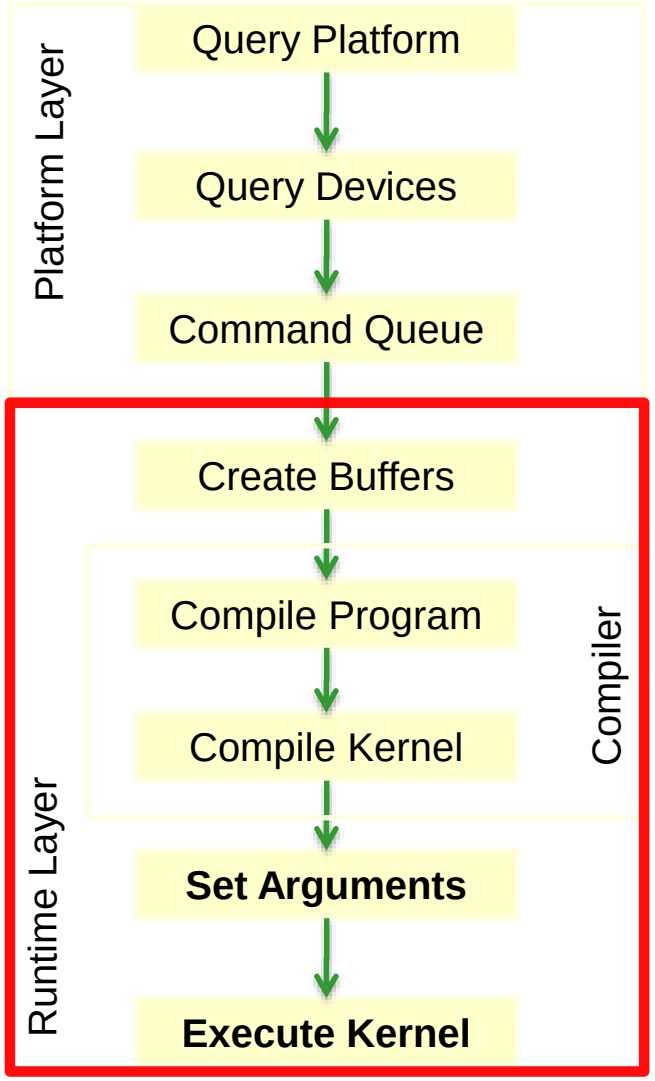


Step3: Set Arguments, Enqueue Kernel

```
// Set Arguments
clSetKernelArg(mykernel, 0, sizeof(cl_mem),
                          (void *)&d_ip);
clSetKernelArg(mykernel, 1, sizeof(cl_mem),
                          (void *)&d_op);
 clSetKernelArg(mykernel, 2, sizeof(cl_int),
                          (void *)&W);
...
```

```
//Set local and global workgroup sizes
size_t localws[2] = {16,16} ;
size_t globalws[2] = {W, H};//Assume divisible by 16
```

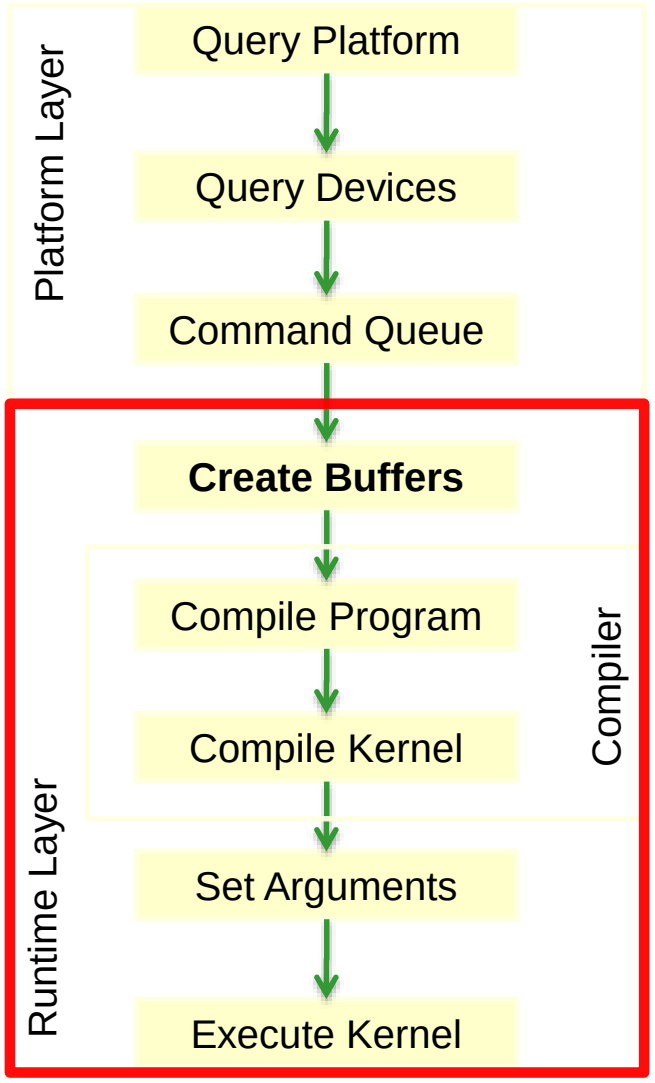
```
// execute kernel
clEnqueueNDRangeKernel(
            myqueue , myKernel,
            2, 0, globalws, localws,
            0, NULL, NULL);
```



Step4: Read Back Result

- **Only necessary for data required on the host**
- **Data output from one kernel can be reused for another kernel**
  - **Avoid redundant host-device IO**

```
// copy results from device back to host
clEnqueueReadBuffer(
                myctx, d_op,
                CL_TRUE,                //Blocking Read
Back
                0, mem_size,  (void *) op_data,
                 NULL, NULL, NULL);
```



OpenCL Timing

- OpenCL provides "events" which can be used for timing kernels
  - Events will be discussed in detail in Lecture 11
- We pass an event to the OpenCL enqueue kernel function to capture timestamps
- Code snippet provided can be used to time a kernel
  - Add profiling enable flag to create command queue
  - By taking differences of the start and end timestamps we discount overheads like time spent in the command queue

```
cl_event event_timer;
clEnqueueNDRangeKernel(
                myqueue , myKernel,
                2, 0, globalws, localws,
                0, NULL, &event_timer);
```

```
unsigned long starttime, endtime;
```

```
clGetEventProfilingInfo( event_time,
    CL_PROFILING_COMMAND_START,
    sizeof(cl_ulong), &starttime, NULL);
clGetEventProfilingInfo(event_time,
    CL_PROFILING_COMMAND_END,
    sizeof(cl_ulong), &endtime, NULL);
```

```
unsigned long elapsed =
(unsigned long)(endtime - starttime);
```

Example: Image Rotation

- We have studied the use of OpenCL buffer objects
- A complete program in OpenCL has been written
- We have understood how an OpenCL work-item can be used to work on a single output element (seen with rotation and matrix multiplication)
  - While the previously discussed examples are correct data parallel programs their performance can be drastically improved

GPU Threads and Scheduling

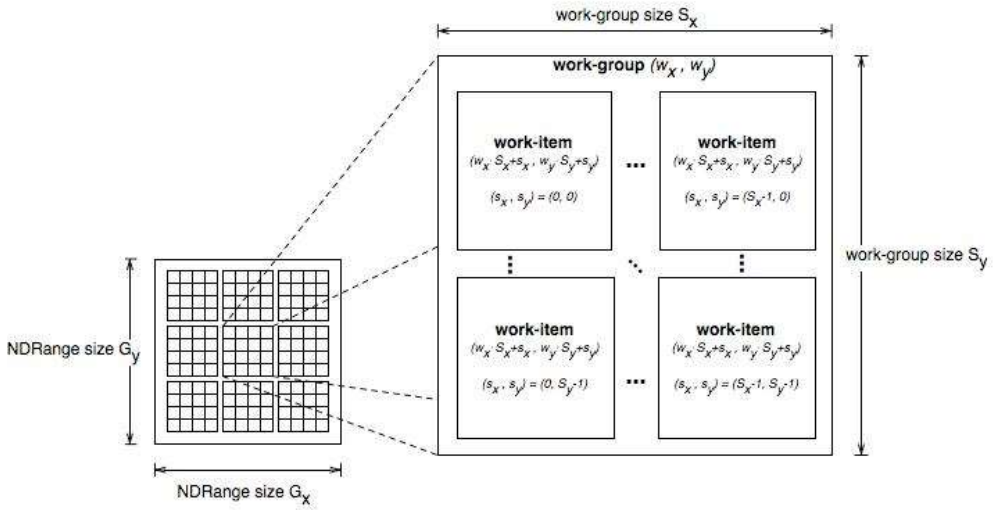
- How are work groups scheduled for execution on the compute units of devices?
- The effects of *divergence* of work items within a group and its negative effect on performance
- The concepts *warps* and *wavefronts* are discussed, even though they are not part of the OpenCL specification
  - Serve as another hierarchy of threads and their implicit synchronization enables interesting implementations of algorithms on GPUs
  - Implicit synchronization and write combining property in local memory used to implement warp voting
  - We discuss how predication is used for divergent work items even though all threads in a warp are issued in lockstep

GPU Threads and Scheduling: Topics

- Wavefronts and warps
- Thread scheduling for both AMD and NVIDIA GPUs
- Predication
- Warp voting and synchronization
- Pitfalls of wavefront/warp specific implementations

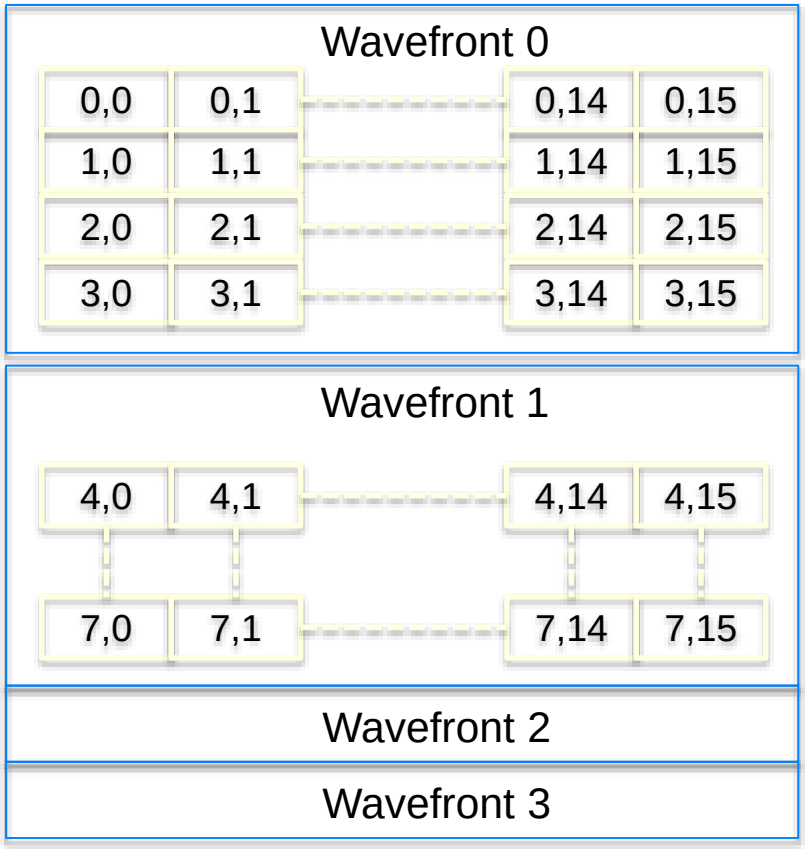
Work Groups to HW Threads

- OpenCL kernels are structured into work groups that map to device compute units
- Compute units on GPUs consist of SIMT processing elements
- Work groups automatically get broken down into hardware schedulable groups of threads for the SIMT hardware
  - This "schedulable unit" is known as a *warp* (NVIDIA) or a *wavefront* (AMD)



Work-Item Scheduling

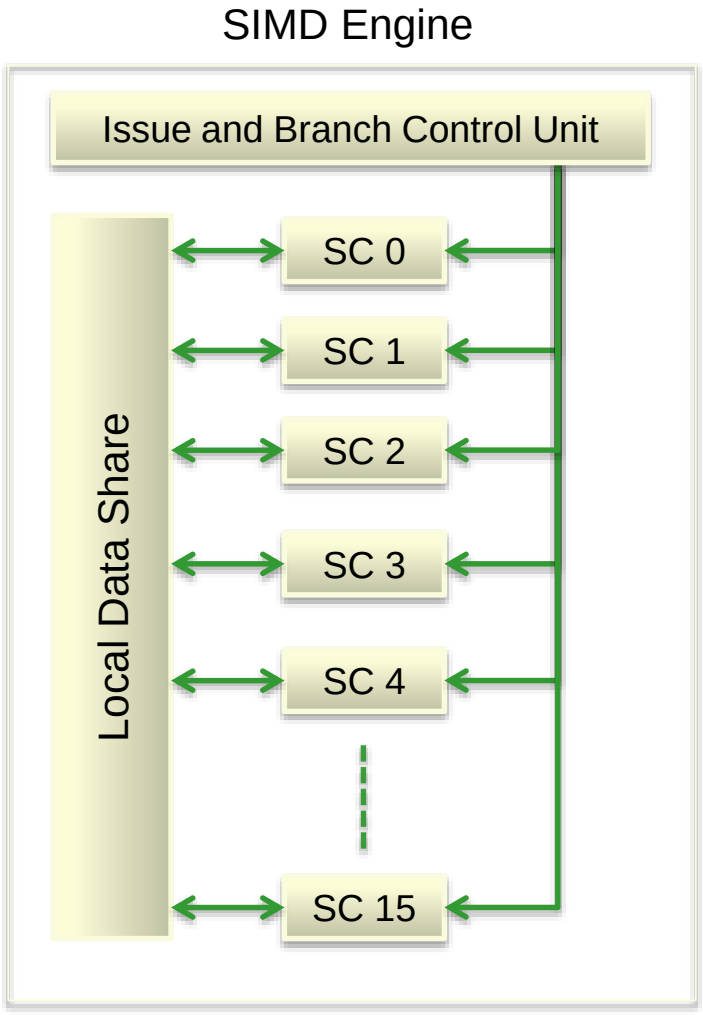
- Hardware creates wavefronts by grouping threads of a work group
  - Along the X dimension first
- All threads in a wavefront execute the same instruction
  - Threads within a wavefront move in lockstep
- Threads have their own register state and are free to execute different control paths
  - Thread masking used by HW
  - Predication can be set by compiler



Grouping of work-group into wavefronts

Wavefront Scheduling - AMD

- **Wavefront size is 64 threads**
  - **Each thread executes a 5 way VLIW instruction issued by the common issue unit**
- **A Stream Core (SC) executes one VLIW instruction**
  - **16 stream cores execute 16 VLIW instructions on each cycle**
- **A quarter wavefront is executed on each cycle, the entire wavefront is executed in four consecutive cycles**

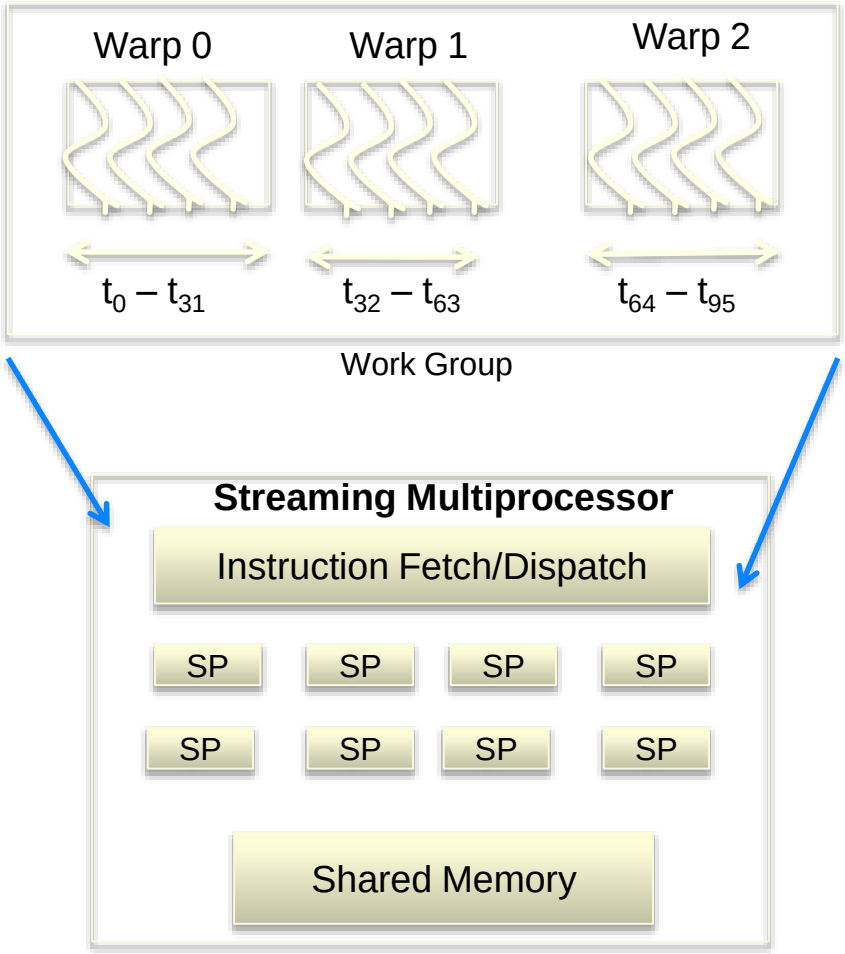


Wavefront Scheduling - AMD

- In the case of Read-After-Write (RAW) hazard, one wavefront will stall for four extra cycles
  - If another wavefront is available it can be scheduled to hide latency
  - After eight total cycles have elapsed, the ALU result from the first wavefront is ready, so the first wavefront can continue execution
- Two wavefronts (128 threads) completely hide a RAW latency
  - The first wavefront executes for four cycles
  - Another wavefront is scheduled for the next four cycles
  - The first wavefront can then run again
- Note that two wavefronts are needed just to hide RAW latency, the latency to global memory is much greater
  - During this time, the compute unit can process other independent wavefronts, if they are available

Warp Scheduling - Nvidia

- **Work groups are divided into 32-thread warps which are scheduled by a SM**
- **On Nvidia GPUs half warps are issued each time and they interleave their execution through the pipeline**
- **The number of warps available for scheduling is dependent on the resources used by each block**
- **Similar to wavefronts in AMD hardware except for size differences**



Occupancy - Tradeoffs

- Local memory and registers are persistent within compute unit when other work groups execute
  - Allows for lower overhead context switch
- The number of active wavefronts that can be supported per compute unit is limited
  - Decided by the local memory required per workgroup and register usage per thread
- The number of active wavefronts possible on a compute unit can be expressed using a metric called occupancy
- Larger numbers of active wavefronts allow for better latency hiding on both AMD and NVIDIA hardware

Divergent Control Flow

- Instructions are issued in lockstep in a wavefront /warp for both AMD and Nvidia
- However each work item can execute a different path from other threads in the wavefront
- If work items within a wavefront go on divergent paths of flow control, the invalid paths of a work-items are masked by hardware
- Branching should be limited to a wavefront granularity to prevent issuing of wasted instructions

Predication and Control Flow

- How do we handle threads going down different execution paths when the same instruction is issued to all the work-items in a wavefront ?
- Predication is a method for mitigating the costs associated with conditional branches
  - Beneficial in case of branches to short sections of code
  - Based on fact that executing an instruction and squashing its result may be as efficient as executing a conditional
  - Compilers may replace “switch” or “if then else” statements by using branch predication

Predication for GPUs

- **Predicate is a condition code that is set to true or false based on a conditional**
- **Both cases of conditional flow get scheduled for execution**
  - **Instructions with a true predicate are committed**
  - **Instructions with a false predicate do not write results or read operands**
- **Benefits performance only for very short conditionals**

```
__kernel
void test() {
      int  tid= get_local_id(0) ;
      if( tid %2 == 0)
            Do_Some_Work() ;
      else
            Do_Other_Work() ;
}
```

Predicate = True for threads 0,2,4….

Predicate = False for threads 1,3,5….

Predicates switched for the else condition

Divergent Control Flow

- **Case 1: All odd threads will execute if conditional while all even threads execute the else conditional. The if and else block need to be issued for each wavefront**
- **Case 2: All threads of the first wavefront will execute the if case while other wavefronts will execute the else case. In this case only one out of if or else is issued for each wavefront**

Case 1

```
int tid = get_local_id(0)
if ( tid % 2 == 0) //Even Work Items
                DoSomeWork()
else
                DoSomeWork2()
```

Conditional – With divergence

Case 2

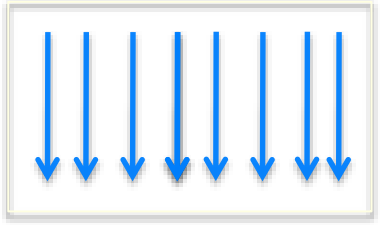
```
int tid = get_local_id(0)
if ( tid / 64 == 0)  //Full First Wavefront
                DoSomeWork()
else if (tid /64 == 1) //Full Second  Wavefront
                DoSomeWork2()
```

Conditional – No divergence

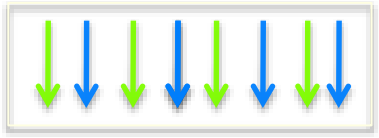
Effect of Predication on Performance

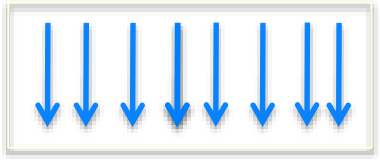
Time for Do_Some_Work = t_1 (if case)

Time for Do_Other _Work = t_2 (else case)

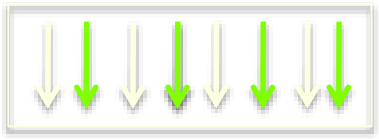


Green colored threads have valid results





Green colored threads have valid results



if(tid %2 == 0)

Do_Some_Work()

Squash invalid results, invert mask

Do_Other _Work()

Squash invalid results

T = 0

T = t_{start}

t_1

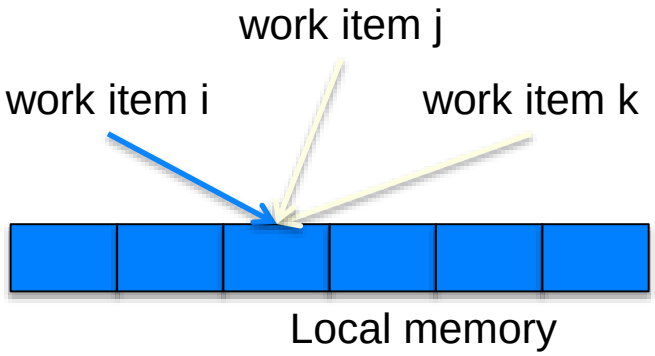
T = t_{start} + t_1

t_2

T = t_{start} + t_1 + t_2

Warp Voting

- **Implicit synchronization per instruction allows for techniques like warp voting**
  - **Useful for devices without atomic shared memory operations**
  - **We discuss warp voting with the 256-bin Histogram example**
- **For 64 bin histogram, we build a sub histogram per thread**
- **Local memory per work group for 256 bins**
  - **256 bins * 4Bytes * 64 threads / block = 64KB**
  - **G80 GPUs have only 16KB of shared memory**
- **Alternatively, build per warp subhistogram**
- **Local memory required per work group**
  - **256 bins * 4Bytes * 2 warps / block = 2KB**

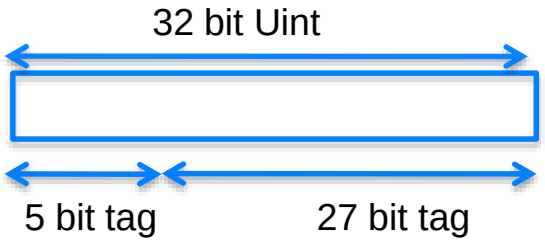


Shared memory write combining on allows **ONLY** one write from work-items i,j or k to succeed

By tagging bits in local memory and rechecking the value a work-item could know if its previously attempted write succeeded

Warp Voting for Histogram256

- **Build per warp subhistogram**
  - **Combine to per work group subhistogram**
  - **Local memory budget in per warp sub histogram technique allows us to have multiple work groups active**
- **Handle conflicting writes by threads within a warp using warp voting**
  - **Tag writes to per warp subhistogram with intra-warp thread ID**
  - **This allows the threads to check if their writes were successful in the next iteration of the while loop**
- **Worst case : 32 iterations done when all 32 threads write to the same bin**




```
void addData256(
        volatile __local uint * l_WarpHist,
        uint data, uint workitemTag) {

 unsigned int count;
 do {
        // Read the current value from
histogram
        count = l_WarpHist[data] &
0x07FFFFFU;
        // Add the tag and incremented data
to
        // the position in the histogram
        count = workitemTag | (count + 1);
         l_WarpHist[data] = count;
 }
 // Check if the value committed to local memory
 // If not go back in the loop and try again
 while(l_WarpHist[data] != count);
}
```

Pitfalls of using Wavefronts

- **OpenCL specification does not address warps/wavefronts or provide a means to query their size across platforms**
  - AMD GPUs (5870) have 64 threads per wavefront while NVIDIA has 32 threads per warp
  - NVIDIA's OpenCL extensions (discussed later) return warp size only for Nvidia hardware
- **Maintaining performance and correctness across devices becomes harder**
  - Code hardwired to 32 threads per warp when run on AMD hardware 64 threads will waste execution resources
  - Code hardwired to 64 threads per warp when run on Nvidia hardware can lead to races and affects the local memory budget
  - We have only discussed GPUs, the Cell doesn't have wavefronts
- **Maintaining portability – assign warp size at JIT time**
  - Check if running AMD / Nvidia and add a `-DWARP_SIZE` *`Size`* to build command

Threads: Summary

- Divergence within a work-group should be restricted to a wavefront/warp granularity for performance
- A tradeoff between schemes to avoid divergence and simple code which can quickly be predicated
  - Branches are usually highly biased and localized which leads to short predicated blocks
- The number of wavefronts active at any point in time should be maximized to allow latency hiding
  - Number of active wavefronts is determined by the requirements of resources like registers and local memory
- Wavefront specific implementations can enable more optimized implementations and enables more algorithms to GPUs
  - Maintaining performance and correctness may be hard due to the different wavefront sizes on AMD and NVIDIA hardware

Understanding GPU Memory

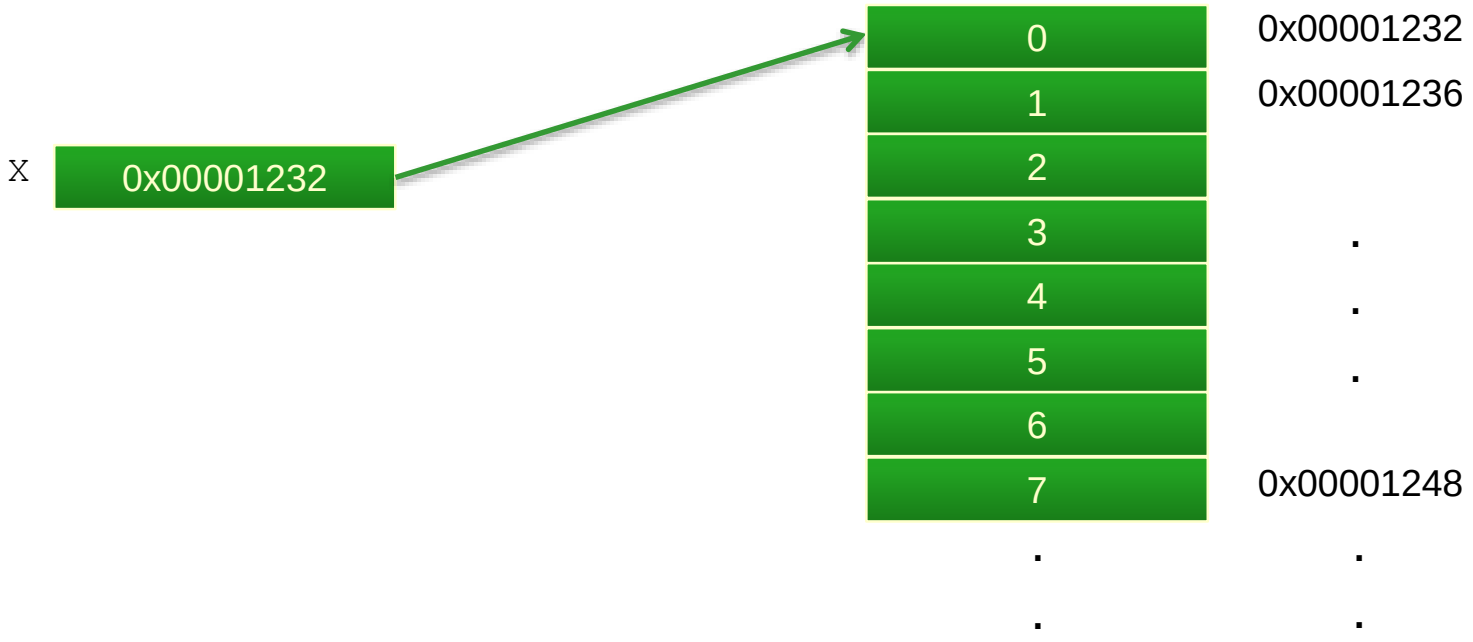
- Study the GPU memory subsystem to understand how data must be managed to obtain performance for data parallel programs
- Understand possible optimizations for programs running on data parallel hardware like GPUs

Memory: Topics

- **Global memory**
  - **GPU bus addressing**
  - **Coalescing accesses**
- **Local memory**
  - **Memory banks to enable simultaneous access**
    - **wavefront/warp level (e.g., half warp, quarter wavefront)**
  - **Memory bank conflicts**

Example

- Array `x` is a pointer to an array of integers (4-bytes each) located at address 0x00001232



- A thread wants to access the data at `x[0]`

```
int tmp = x[0];
```

Bus Addressing

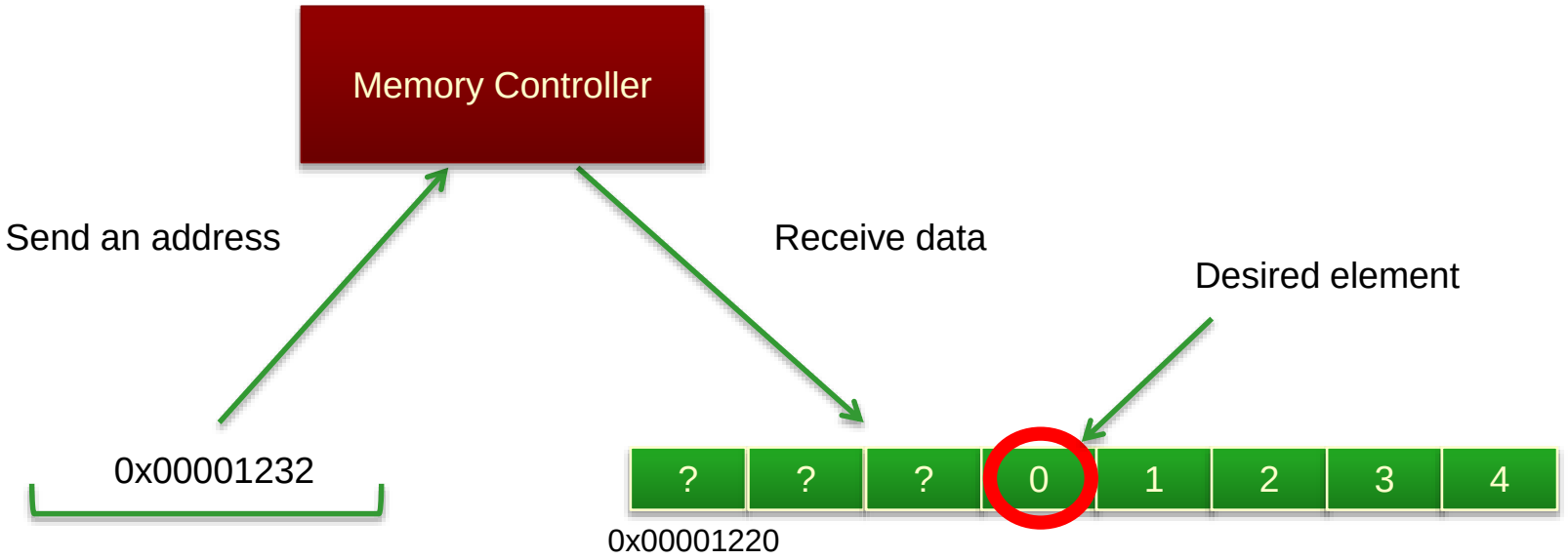
- **Assume that the memory bus is 32-bytes (256-bits) wide**
  - **This is the width on a Radeon 5870 GPU**
- **The byte-addressable bus must make accesses that are aligned to the bus width, so the bottom 5 bits are masked off**

Desired address:	`0x00001232`
Bus mask:	`0xFFFFFFE0`
Bus access:	`0x00001220`

- **Any access in the range 0x00001220 to 0x0000123F will produce the address 0x00001220**

Bus Addressing

- All data in the range 0x00001220 to 0x0000123F is returned on the bus
- In this case, 4 bytes are useful and 28 bytes are wasted



Coalescing Memory Accesses

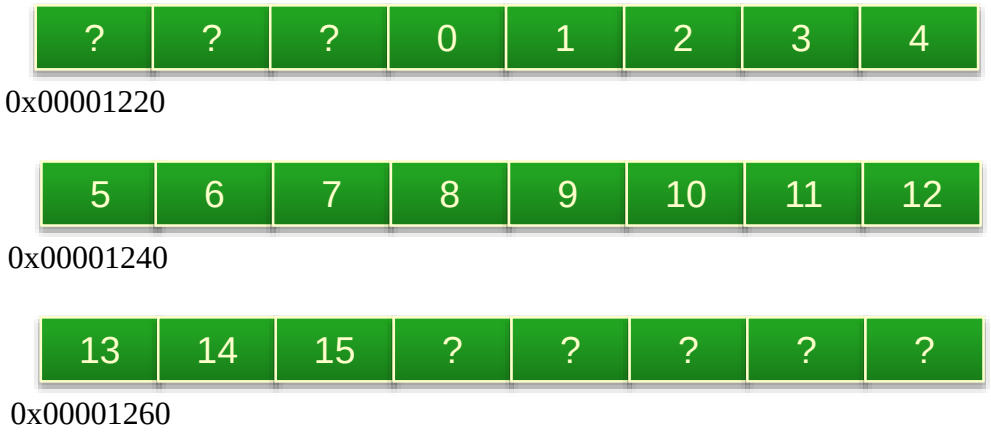
- To fully utilize the bus, GPUs combine the accesses of multiple threads into fewer requests when possible
- Consider the following OpenCL kernel code:

```
int tmp = X[get_global_id(0)];
```

- Assuming that array `X` is the same array from the example, the first 16 threads will access addresses 0x00001232 through 0x00001272
- If each request was sent out individually, there would be 16 accesses total, with 64 useful bytes of data and 448 wasted bytes
  - Notice that each access in the same 32-byte range would return exactly the same data

Coalescing Memory Accesses

- When GPU threads access data in the same 32-byte range, multiple accesses are combined so that each range is only accessed once
  - Combining accesses is called *coalescing*
- For this example, only 3 accesses are required
  - If the start of the array was 256-bit aligned, only two accesses would be required

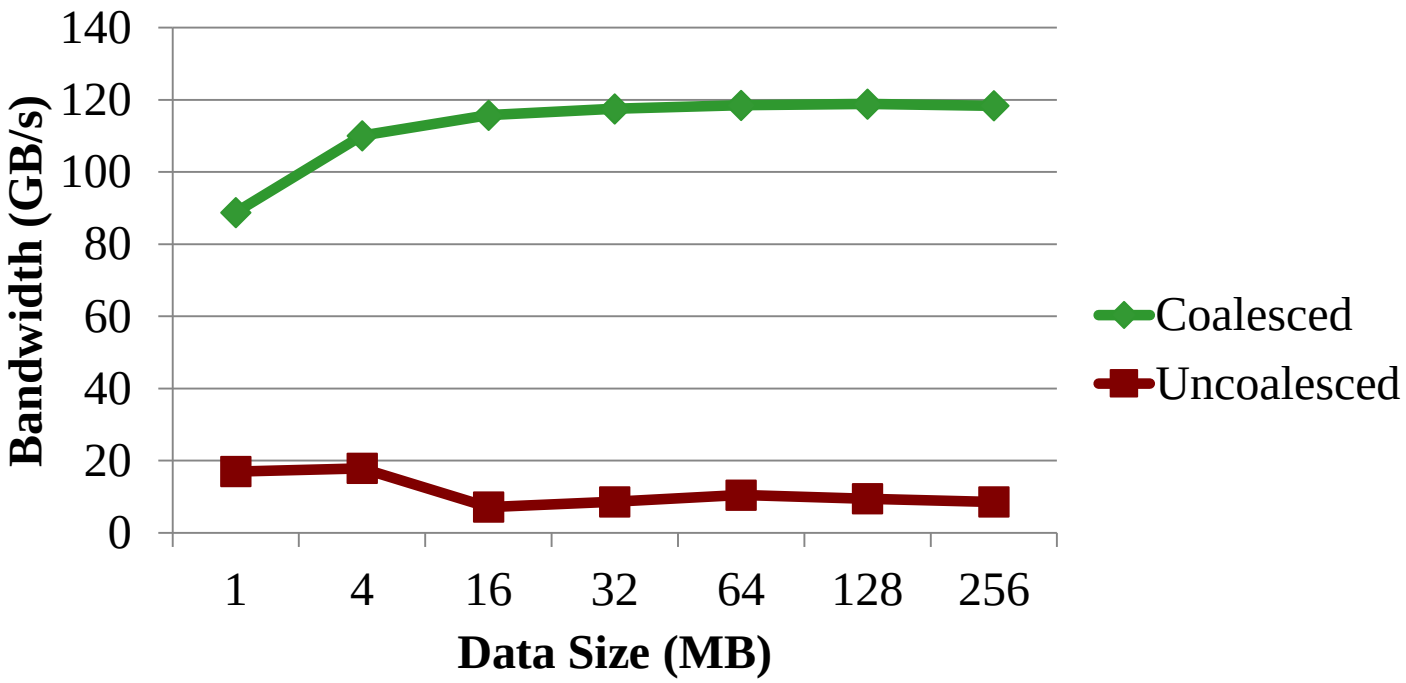


Coalescing Memory Accesses

- Recall that for AMD hardware, 64 threads are part of a wavefront and must execute the same instruction in a SIMD manner
- For the AMD 5870 GPU, memory accesses of 16 consecutive threads are evaluated together and can be coalesced to fully utilize the bus
  - This unit is called a quarter-wavefront and is the important hardware scheduling unit for memory accesses

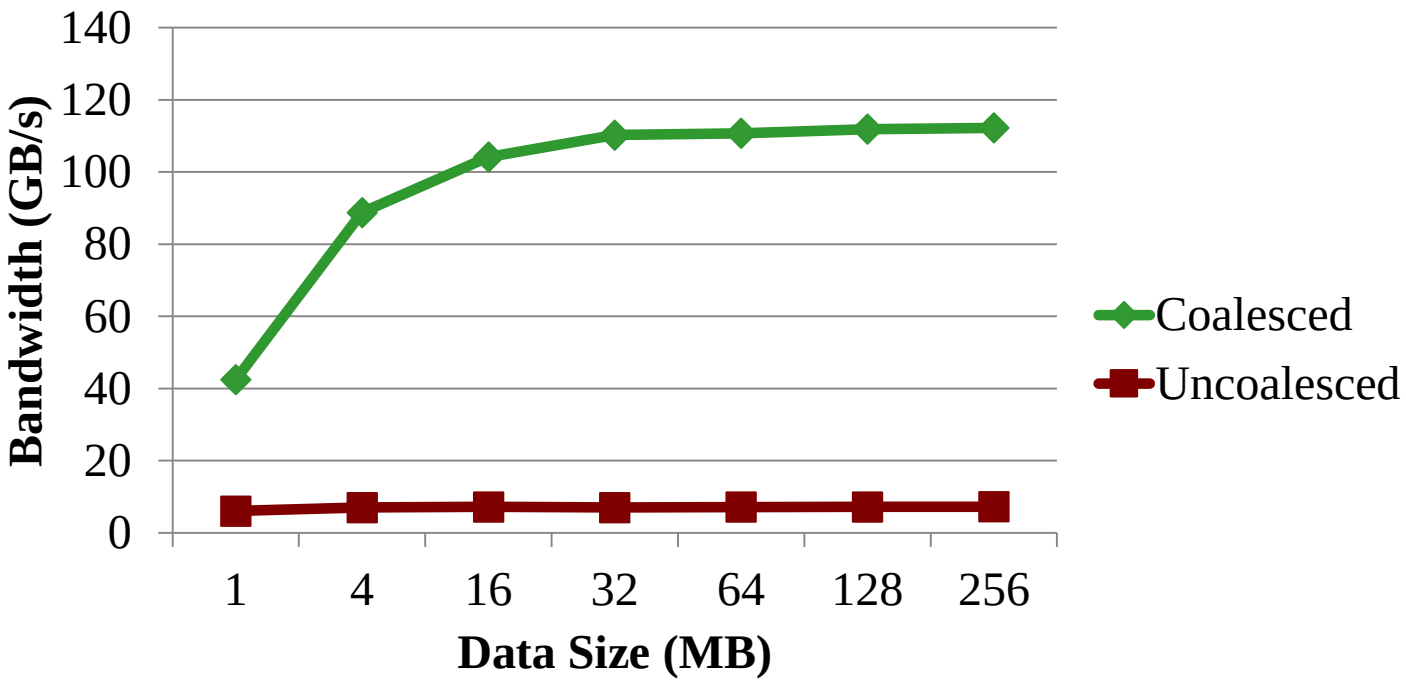
Coalescing Memory Accesses

- **Global memory performance for a simple data copying kernel of entirely coalesced and entirely non-coalesced accesses on an ATI Radeon 5870**



Coalescing Memory Accesses

- **Global memory performance for a simple data copying kernel of entirely coalesced and entirely non-coalesced accesses on an NVIDIA GTX 285**



Memory Banks

- **Memory is made up of *banks***
  - **Memory banks are the hardware units that actually store data**
- **The memory banks targeted by a memory access depend on the address of the data to be read/written**
  - **Note that on current GPUs, there are more memory banks than can be addressed at once by the global memory bus, so it is possible for different accesses to target different banks**
    - **Bank response time, not access requests, is the bottleneck**
- **Successive data are stored in successive banks (strides of 32-bit words on GPUs) so that a group of threads accessing successive elements will produce no bank conflicts**

Bank Conflicts – Local Memory

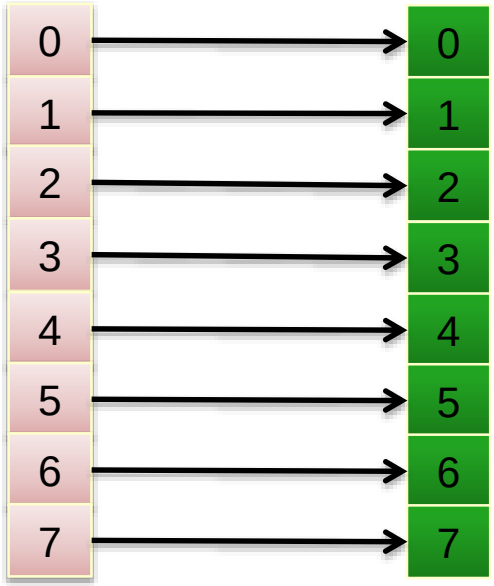
- **Bank conflicts have the largest negative effect on local memory operations**
  - **Local memory does not require that accesses are to sequentially increasing elements**
- **Accesses from successive threads should target different memory banks**
  - **Threads accessing sequentially increasing data will fall into this category**

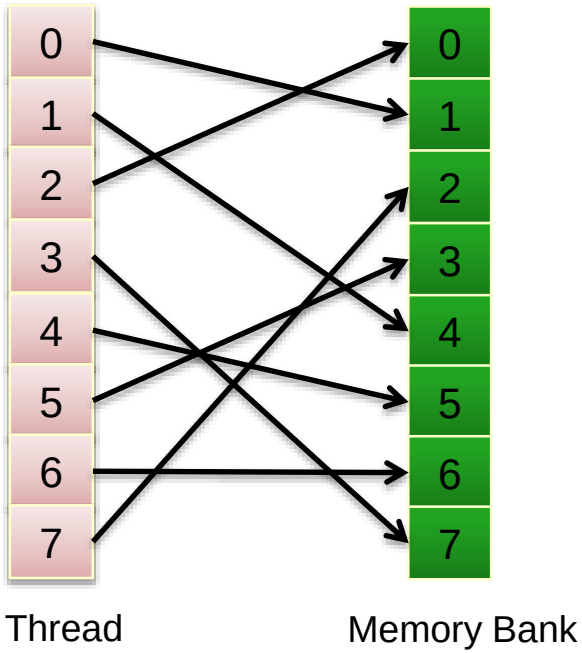
Bank Conflicts – Local Memory

- On AMD, a wavefront that generates bank conflicts stalls until all local memory operations complete
  - The hardware does not hide the stall by switching to another wavefront
- The following examples show local memory access patterns and whether conflicts are generated
  - For readability, only 8 memory banks are shown

Bank Conflicts – Local Memory

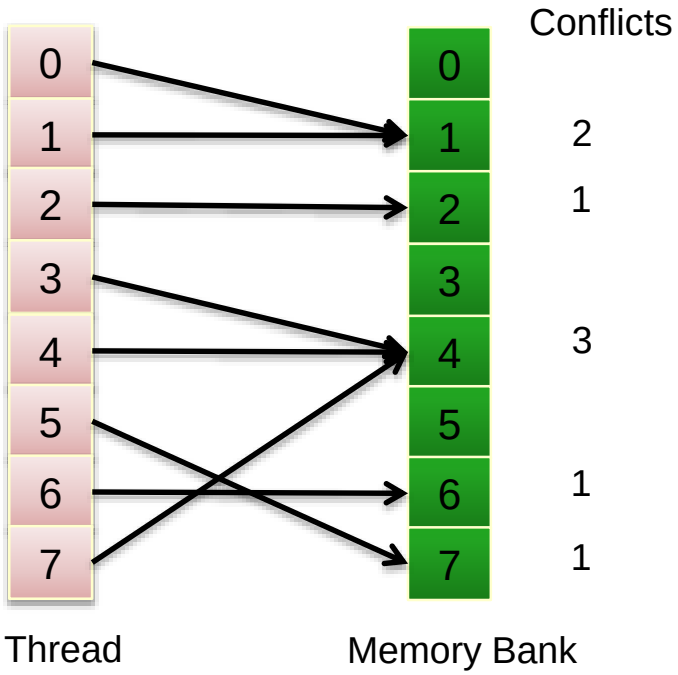
- If there are no bank conflicts, each bank can return an element without any delays
  - Both of the following patterns will complete without stalls on current GPU hardware





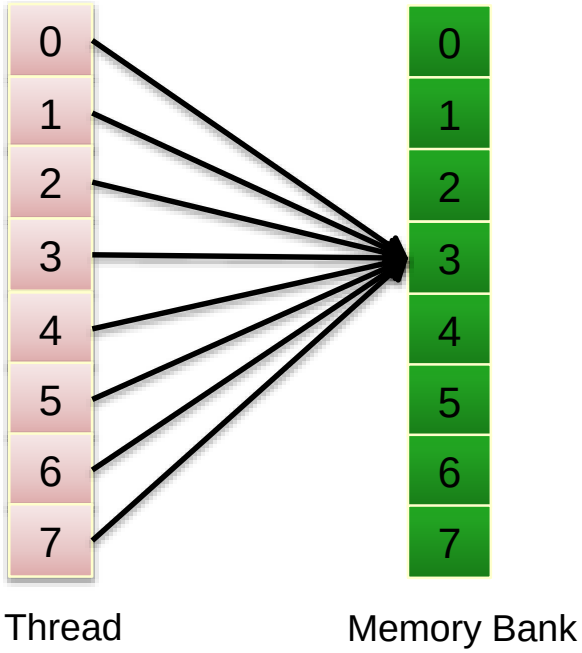
Bank Conflicts – Local Memory

- If multiple accesses occur to the same bank, then the bank with the most conflicts will determine the latency
  - The following pattern will take 3 times the access latency to complete



Bank Conflicts – Local Memory

- If all accesses are to the same address, then the bank can perform a broadcast and no delay is incurred
  - The following will only take one access to complete assuming the same data element is accessed



Bank Conflicts – Global Memory

- **Bank conflicts in global memory rely on the same principles, however the global memory bus makes the impact of conflicts more subtle**
  - **Since accessing data in global memory requires that an entire bus-line be read, bank conflicts within a work-group have a similar effect as non-coalesced accesses**
    - **If threads reading from global memory had a bank conflict then by definition it manifest as a non-coalesced access**
    - **Not all non-coalesced accesses are bank conflicts, however**
- **The ideal case for global memory is when different work-groups read from different banks**
  - **In reality, this is a very low-level optimization and should not be prioritized when first writing a program**

Summary

- GPU memory is different than CPU memory
  - The goal is high throughput instead of low-latency
- Memory access patterns have a huge impact on bus utilization
  - Low utilization means low performance
- Having coalesced memory accesses and avoiding bank conflicts are required for high performance code
- Specific hardware information (such as bus width, number of memory banks, and number of threads that coalesce memory requests) is GPU-specific and can be found in vendor documentation