GPU Threads and Scheduling

A Collaboration Between
David Kaeli, Northeastern University
Benedict R. Gaster, AMD

Instructor Notes

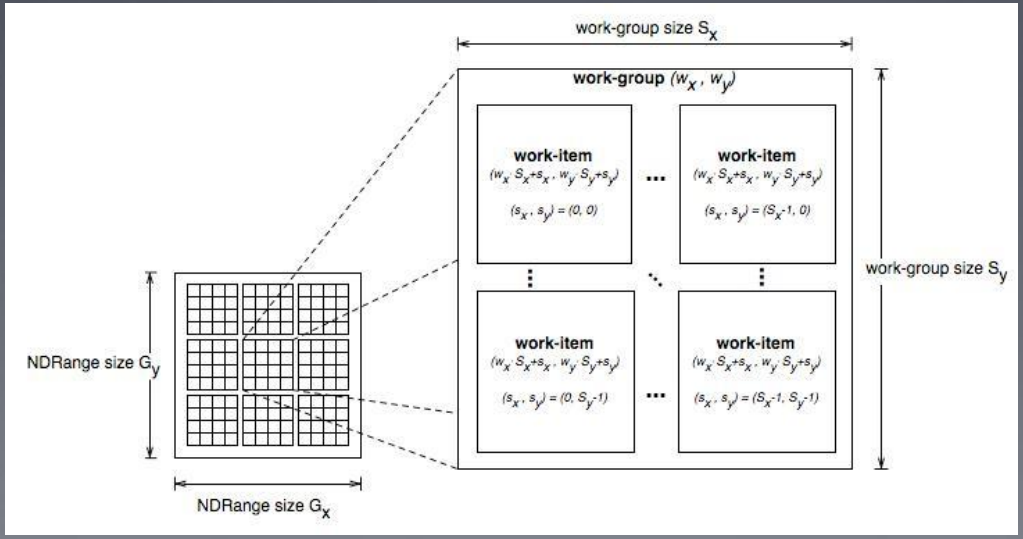
- This lecture deals with how work groups are scheduled for execution on the compute units of devices
- Also explain the effects of divergence of work items within a group and its negative effect on performance
- Reasons why we discuss warps and wavefronts because even though they are not part of the 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

Topics

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

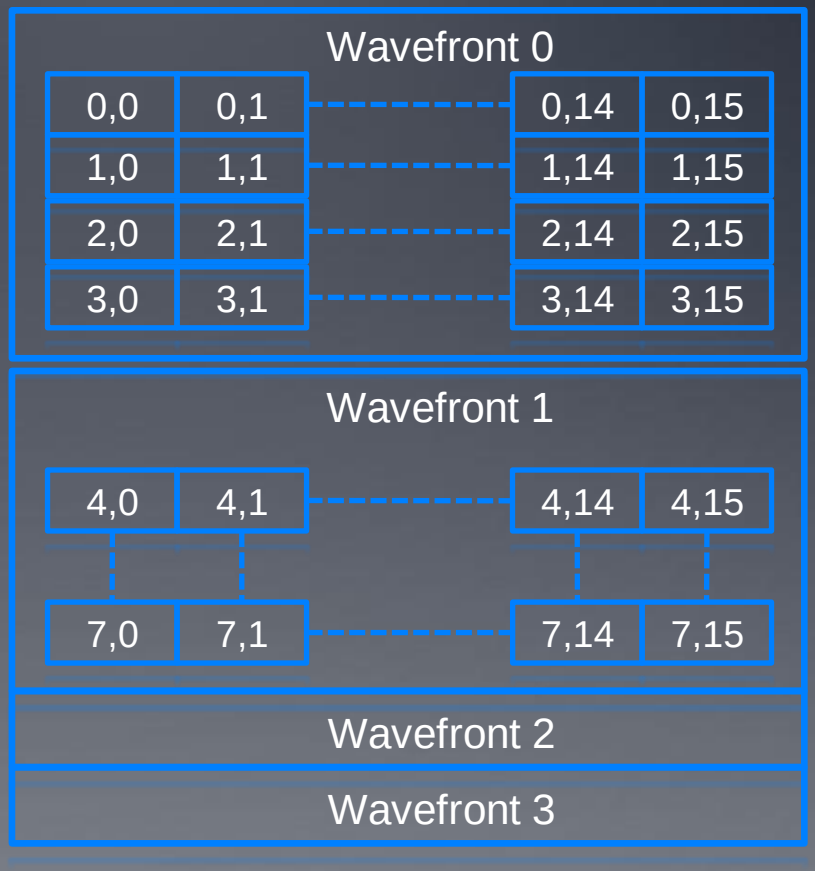
Work Groups to HW Threads

- 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

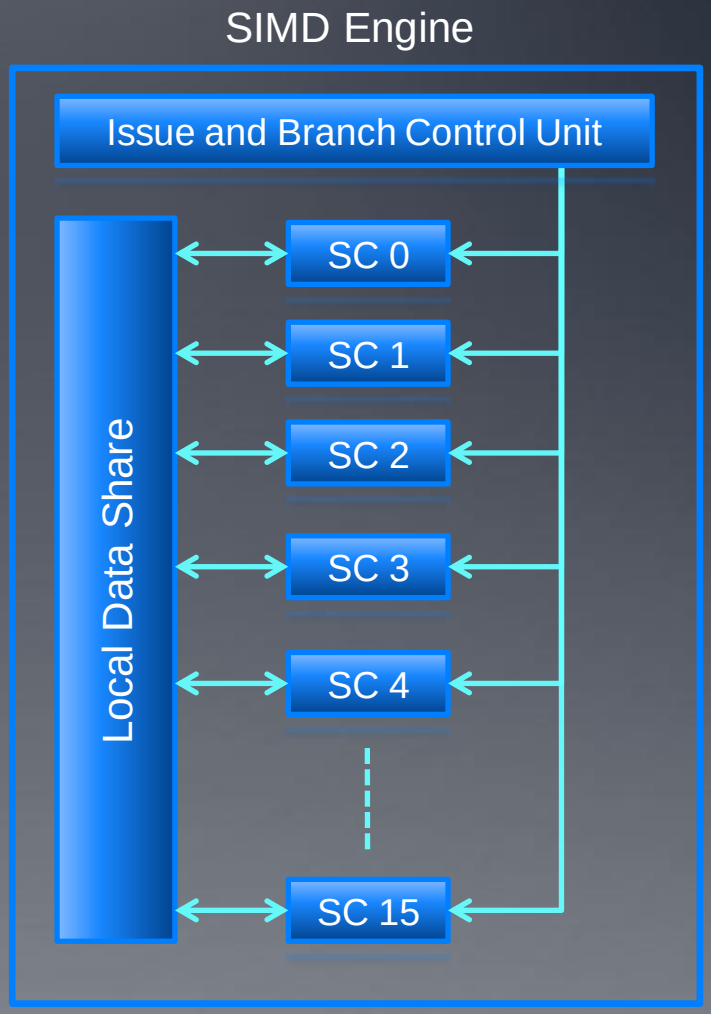
- Hardware creates wavefronts by grouping threads of a work group
  - Along the X dimension first
- All threads in a wavefront execute the same instruction
  - Threads within a wavefront move in lockstep
- Threads have their own register state and are free to execute different control paths
  - Thread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

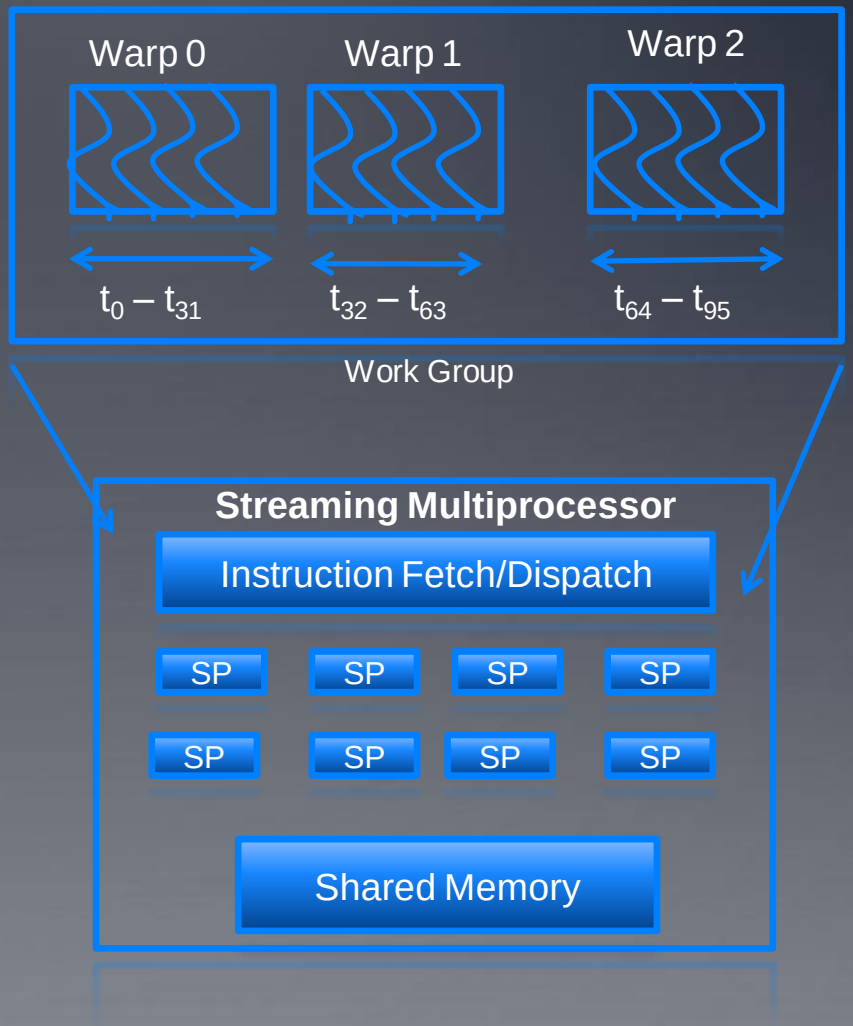


Wavefront Scheduling - AMD

- 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
- Occupancy will be discussed in detail in Lecture 08

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

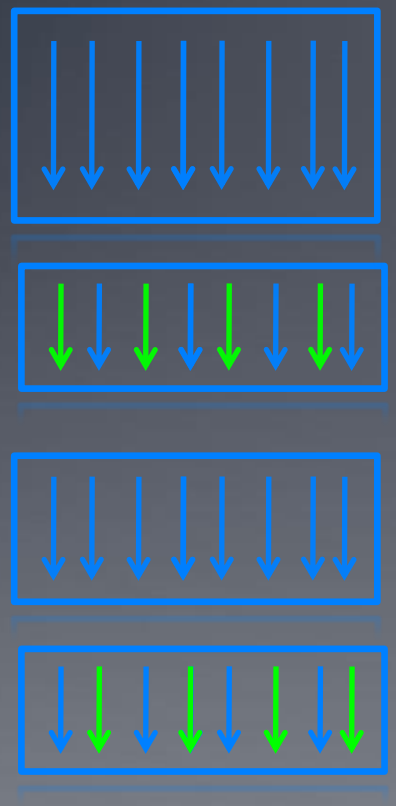
Case 2

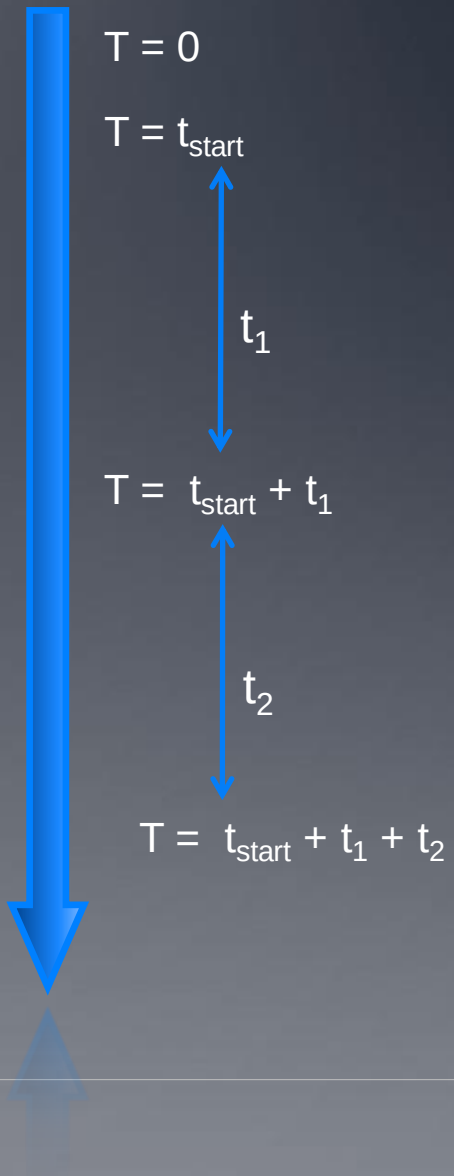
```
int tid = get_local_id(0)
if ( tid / 64 == 0)  //Full First Wavefront
        DoSomeWork()
else if (tid /64 == 1) //Full Second  Wavefront
        DoSomeWork2()
```

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)

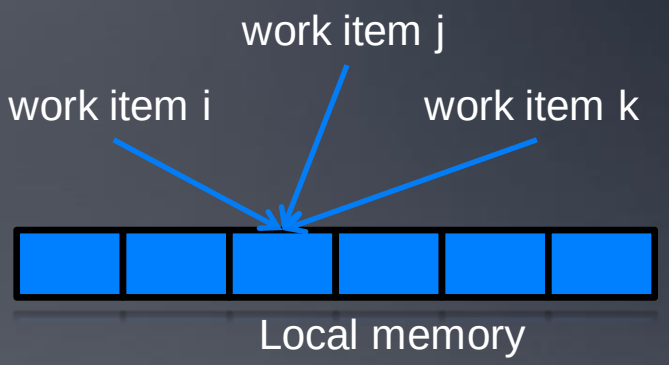




Total Time taken = t_{start} +t_1 + t_2

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



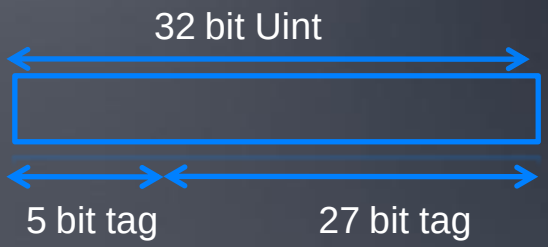
Shared memory write combining on allows **ONLY** one write from work-items i,j or k to succeed

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

Source: Nvidia GPU Computing SDK Examples




```
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

Warp-Based Implementation

- Implicit synchronization in warps at each instruction allows for expression of another thread hierarchy within work group
  - Warp specific implementations common in CUDA literature
- E.g.: 256 Bin Histogram
  - NVIDIA's implementation allows building histograms in local memory for devices without atomic operation support and limited shared memory
  - Synchronization in warps allows for implementing the voting discussed previously reducing local memory budget from N_THREADS*256 to N_WARPS_PER_BLOCK*256
- E.g.: CUDPP: CUDA Data Parallel Primitives
  - Utilizes an efficient warp scan to construct a block scan which works on one block in CUDA

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