OpenCL
Parallel Reduction

21. November 2017 | Andreas Beckmann

Agenda

- Synchronization
- Parallel Reduction
  - CPU version
  - GPU variants
    - only one work-item performs the partial sum
    - previous version – memory optimized
    - introducing binary sum
    - previous version – optimize loop iterations
    - 2nd kernel for summing up final partial sums

Acknowledgements

This presentation is an excerpt of:

- Introduction to OpenCL, Training course June 2012, George Leaver, University of Manchester

Synchronization

- Synchronization can be performed between
  - work-items in a work-group
  - kernels (commands) in a command-queue
  - kernels (commands) in seperate queues within the same context
  - host and queues
- Synchronization can be enforced implicitly
  - in-order-queue: commands execute in order submitted
- Synchronization can be requested by user
  - out-of-order queue: commands scheduled by OpenCL
  - barriers in kernels and queues
  - events in queues

Work-item Synchronization

- Only possible within a work-group
  - Can't sync with work-items in other work-groups
  - Can't sync one work-group with another
- Use **barrier(type)** in kernel where type is
  - CLK_LOCAL_MEM_FENCE: ensure consistency in local memory
  - CLK_GLOBAL_MEM_FENCE: ensure consistency in global memory
- All work-items in work-group must issue the **barrier()** call and same number of calls

```
__kernel void BadKernel(...) {
  int i = get_global_id(0);
  ...
  // ERROR: Not all WIs reach barrier
  if ( i % 2 )
    barrier(CLK_GLOBAL_MEM_FENCE);
}
```

```
__kernel void BadKernel(...) {
  int i = get_local_id(0);
  ...
  // ERROR: WIs issue different number
  for ( j=0; j<=i; j++ )
    barrier(CLK_LOCAL_MEM_FENCE);
}
```

Use with __local memory

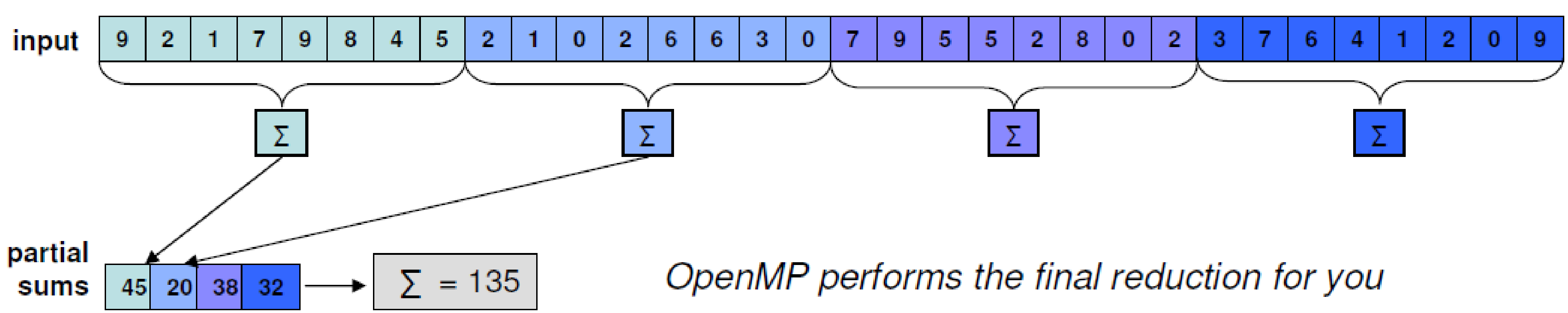
- Barrier often used when initializing __local memory
  - Kernel must initialize local memory

```
__kernel void kMat( const int n, __global float *A, __local float *tmp_arr )
{
  // Could also have some fixed size local array
  // __local float tmp_arr[64];

  int gbl_id = get_global_id(0);   // ID within entire index space
  int loc_id = get_local_id(0);    // ID within this work-group
  int loc_sz = get_local_size(0);  // Size of this work-group

  // For some reason we want to fill up the first half of the local array
  if ( loc_id < loc_sz/2 )
    tmp_arr[loc_id] = A[gbl_id];

  // All work-items must hit barrier. They'll all see a consistent tmp_arr[]
  barrier(CLK_LOCAL_MEM_FENCE);

  // Each work-item can now use the elements from tmp_arr[] safely.
  // Often used if we'd be repeatedly accessing the same A[] elements.
  for ( j=0; j<loc_sz; j++ )
    my_compute( gbl_id, tmp_arr[j], A );
```

Example: Parallel Reduction (CPU)

- Reduction of an array of elements to a single value
  - e.g. sum:
  - OpenMP on host performs partial in each thread
    - Linear (serial) sum within thread

```
void sumCPU( const int n, const float *x,
             float *res )
{
  float sum = 0.0;
#pragma omp parallel for reduction(+:sum)
  for ( int i=0; i<n; i++ )
    sum += x[i];
  *res = sum;
}
```

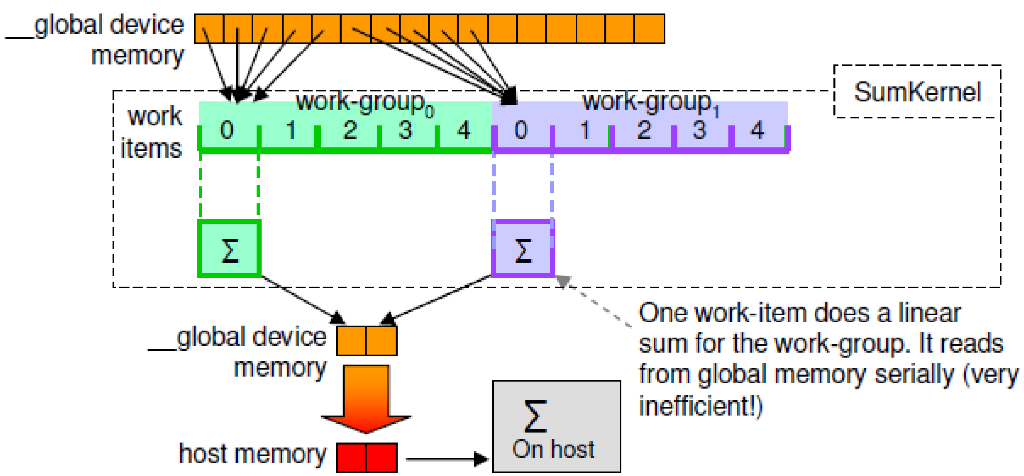


OpenMP performs the final reduction for you

- Can recreate in OpenCL using work-groups

Parallel Sum on GPU (I)

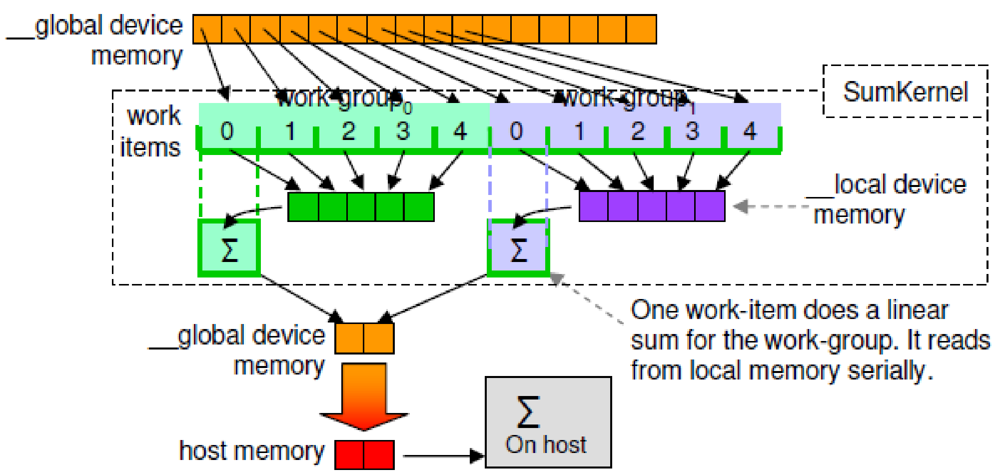
- Use one work-item to perform a sum within a work-group
  - Inefficient – only one work-item forms the partial sum




```
__kernel void sumGPU1( const uint n, __global const float *x,
                       __global float *partialSums ) {
  if ( get_local_id(0) == 0 ) {                          // Many idle work-items!
    float group_sum = x[get_global_id(0)];
    for ( int i=1; i<get_local_size(0); i++ )
      group_sum += x[get_global_id(0)+i];                // Should check (gid+i) < n
    partialSums[get_group_id(0)] = group_sum;            // Write sum to output array
  }
  // Add barrier(CLK_GLOBAL_MEM_FENCE) if doing other work in kernel
}
```

Parallel Sum on GPU (II)

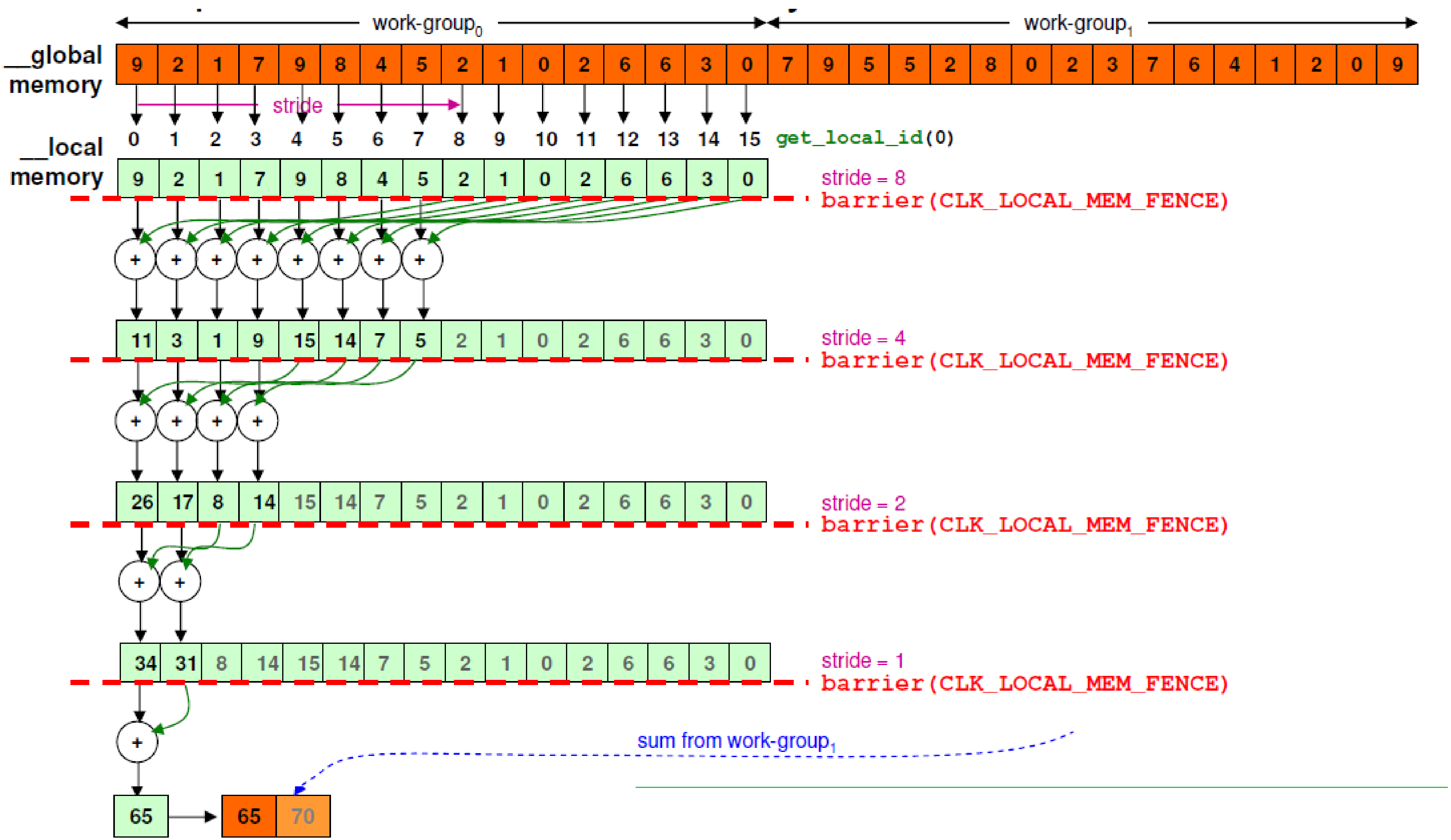
- Memory optimization – copy to __local memory in parallel
  - Still inefficient – only one work-item performs the partial sum




```
__kernel void sumGPU2( const uint n, __global const float *x,
                    __global float *partialSums, __local float *localCopy ) {
  localCopy[get_local_id(0)] = x[get_global_id(0)]; // Init the localCopy array
    barrier(CLK_LOCAL_MEM_FENCE);                   // All work-items must call
  if ( get_local_id(0) == 0 ) {                     // Many idle work-items!
    float group_sum = localCopy[0];
    for ( int i=1; i<get_local_size(0); i++ )
      group_sum += localCopy[i];                    // Sum up the local copy
    partialSums[get_group_id(0)] = group_sum;       // Write sum to output array
  } // Add barrier(CLK_GLOBAL_MEM_FENCE) if doing other work in kernel
}
```

Parallel Reduction (III)

- Improve on linear sum – binary sum



Parallel Reduction (III) Kernel

- Copy all work-group's values into __local memory
  - Repeatedly half the work-group, adding one half to the other

```
__kernel void sumGPU3( const uint n, __global const float *x,
                       __global float *partialSums, __local float *localSums )
{
  uint local_id   = get_local_id(0);
  uint group_size = get_local_size(0);

  // Copy from global mem in to local memory (should check for out of bounds)
  localSums[local_id] = x[get_global_id(0)];
  for (uint stride=group_size/2; stride>0; stride /= 2) { // stride halved at loop

    // Synchronize all work-items so we know all writes to localSums have occurred
    barrier(CLK_LOCAL_MEM_FENCE);

    // First n work-items read from second n work-items (n=stride)
    if ( local_id < stride )
      localSums[local_id] += localSums[local_id + stride]
  }
  // Write result to nth position in global output array (n=work-group-id)
  if ( local_id == 0 )
    partialSum[get_group_id(0)] = localSums[0];
}
```

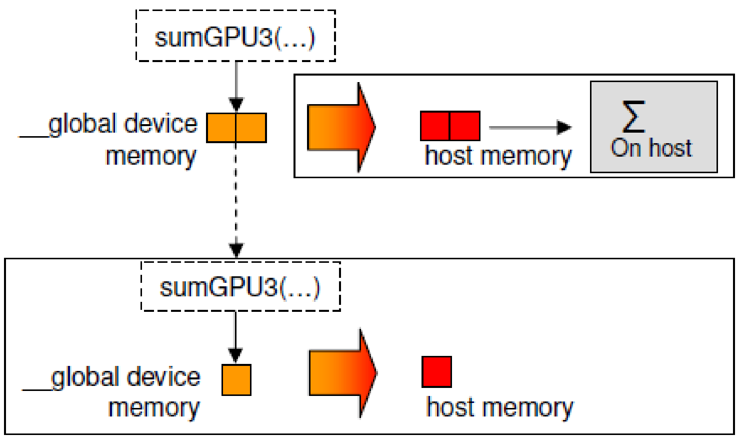
Improved Reduction (IV) Kernel

- Slight re-order to remove a couple of loop iterations
  - Set global_work_size to be half the input array length

```
__kernel void sumGPU4( const uint n, __global float *x,
                       __global float *partialSums, __local float *localSums ) {
  uint global_id   = get_global_id(0);     // Gives where to read from
  uint global_size = get_global_size(0);   // Used to calc where to read from
  uint local_id    = get_local_id(0);      // Gives where to read/write local mem
  uint group_size  = get_local_size(0);    // Used to calc initial stride

  // Copy from global mem in to local memory (doing first iteration)
  localSums[local_id] = x[global_id] + x[global_id + global_size];
  barrier(CLK_LOCAL_MEM_FENCE);
  for (uint stride=group_size/2; stride>1; stride>>=1 ) { // >>=1 does same as /=2
    // First n work-items read from second n work-items (n=stride)
    if ( local_id < stride )
      localSums[local_id] += localSums[local_id + stride];

    // Synchronize so we know all work-items have written to localSums
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  // Last iter: write result to nth position in global x array (n=work-group id)
  if ( local_id == 0 )
    x[get_group_id(0)] = localSums[0]+localSums[1];
```

Partial Sums

- Still have n partial sums (n=number of work-groups)
  - Sum on host
  - Sum on GPU
    - Linear sum (use one work-item)
    - Parallel reduction (use one work-group) provided n is small enough; iterate if not



- Both GPU options can be done with another kernel call
  - Data is still on the GPU (in the partialSums array)
    - Avoid a device-to-host transfer
    - Simply make another kernel passing in the device memory object
    - DO NOT transfer back to host then pass back to device!

Exercise Parallel_Reduction/parallel-reduction

- inspect the parallel reduction program and kernels
  - par_reduction.cc
  - SumGPU[1-4].cl
- and run it on different platforms:
  - make
  - ./par_reduction cpu|gpu|acc 1|2|3|4
- complete the device-only program by filling the TODO gap in
  - par_reduction_device_only.cc
- build and run it:
  - make
  - ./par_reduction_device_only cpu|gpu|acc