OPENCL 2.0 FEATURES

BENJAMIN COQUELLE
MAY 2015

MAIN FEATURES

- **Shared virtual memory**
  – Allows to share complex structures between host and devices.
- **Pipes**
- **Nested parallelism**
  – Enqueue a kernel from a kernel
  – Similar to CUDA dynamic parallelism (compute capability 3.5)
- **Work group built-in functions (scan, reduce…)**
- **Generic address space**
  – avoid to duplicate code

SHARE VIRTUAL MEMORY (SVM)

AMD

- **clSVMAlloc – allocates a shared virtual memory buffer**
  - Specify size in bytes
  - Specify usage information
  - Optional alignment value

- **SVM pointer can be shared by the host and OpenCL device**

void* clSVMAlloc(cl_context ctx, cl_mem_flags flags, size_t size, unsigned int alignement)

- **Examples**

clSVMAlloc(ctx, CL_MEM_READ_WRITE, 1024 * sizeof(float), 0)

clSVMAlloc(ctx, CL_MEM_READ_ONLY, 1024 * 1024, sizeof(cl_float4))

- **Free SVM buffers**

- clEnqueueSVMFree, clSVMFree

SHARE VIRTUAL MEMORY (SVM)

- **clSetKernelArgSVMPointer**
  - SVM pointers as kernel arguments
  - A SVM pointer
  - A SVM pointer + offset

```
// allocating SVM pointers
cl_float *src = (cl_float *)clSVMAlloc(ctx, CL_MEM_READ_ONLY, size, 0);
cl_float *dst = (cl_float *)clSVMAlloc(ctx, CL_MEM_READ_WRITE, size, 0);

// Passing SVM pointers as arguments
clSetKernelArgSVMPointer(vec_add_kernel, 0, src);
clSetKernelArgSVMPointer(vec_add_kernel, 1, dst);

// Passing SVM pointer + offset as arguments
clSetKernelArgSVMPointer(vec_add_kernel, 0, src + offset);
clSetKernelArgSVMPointer(vec_add_kernel, 1, dst + offset);
```

SHARE VIRTUAL MEMORY (SVM)

AMD

- **clSetKernelExecInfo**
  - Passing SVM pointers in other SVM objects

```
// allocating SVM pointers
my_info_t *pA = (my_info_t *)clSVMAlloc(ctx,
CL_MEM_READ_ONLY, sizeof(my_info_t), 0);

pA->pB = (cl_float *)clSVMAlloc(ctx,
CL_MEM_READ_WRITE, size, 0);
```

```
typedef struct
{
 float *pB;
} my_info_t;

kernel void my_kernel(global my_info_t *pA,…)
{
 do_stuff(pA->pB, …);
}
```

```
// Passing SVM pointers
clSetKernelArgSVMPointer(my_kernel, 0, pA);
clSetKernelExecInfo (my_kernel, CL_KERNEL_EXEC_INFO_SVM_PTRS,1 * sizeof(void *), &pA->pB);
```

SVM
BINARY TREE EXAMPLE

```
typedef struct nodeStruct
{
  int value;
  struct nodeStruct* left;
  struct nodeStruct* right;
} node;


svmTreeBuf = clSVMAlloc(context,
                        CL_MEM_READ_WRITE,
                        numNodes*sizeof(node),
                        0);
```

SHARE VIRTUAL MEMORY (SVM)

- **Three types of sharing**
  - Coarse-grained buffer sharing
  - Fine-grained buffer sharing
  - System sharing

SHARE VIRTUAL MEMORY (SVM)
COARSE & FINE-GRAINED BUFFER SHARING

- **SVM buffers allocated using clSVMAlloc**
- **Coarse grained sharing**
  - Memory consistency only guaranteed at synchronization points
  - Host still needs to use synchronization APIs to update data
  - clEnqueueSVMMap / clEnqueueSVMUnmap or event callbacks
  - Memory consistency is at a buffer level
  - Allows sharing of pointers between host and OpenCL device
- **Fine grained sharing**
  - No synchronization needed between host and OpenCL device
  - Host and device can update data in buffer concurrently
  - Memory consistency using C11 atomics and synchronization operations
  - Optional Feature

SHARE VIRTUAL MEMORY (SVM)

SYSTEM SHARING

- **Can directly use any pointer allocated on the host**
  - No OpenCL APIs needed to allocate SVM buffers. Just use malloc/new
- **Both host and OpenCL device can update data using C11 atomics and synchronization functions**
- **Optional Feature**

SHARE VIRTUAL MEMORY (SVM)

COARSE GRAIN BUFFER SVM VS CL1.2

```
//by default the buffer is allocated as coarse grain
float* Buffer = (float*)clSVMAlloc(ctx, CL_MEM_READ_WRITE,
                    1024 * sizeof(float), 0);


//map and fill the buffer from host
status = clEnqueueSVMMap(commandQueue, CL_TRUE, CL_MAP_WRITE,
                    Buffer, 1024*sizeof(float)), 0,
                    NULL, NULL);
for (int i=0; i<1024; i++)
          Buffer[i] = ….;
//data transfer will happen here
clEnqueueSVMUnmap(commandQueue, Buffer, 0, NULL, NULL);


// use your SVM buffer in you OpenCL kernel
clSetKernelArgSVMPointer(my_kernel, 0, Buffer);


clEnqueueNDRangeKernel(queue, my_kernel,…)
```

```
//create device buffer
cl_mem DeviceBuffer =  clCreateBuffer(ctx,
          CL_MEM_READ_WRITE, 1024*sizeof(float), NULL, &err
);


//create host buffer
float* hostBuffer = new float[1024];
for (int i=0; i<1024; i++)
           hostBuffer ] = ….;
//data transfer happens here
clEnqueueWriteBuffer(queue, DeviceBuffer,… , hostBuffer);


//use our device buffer on device
clSetKernelArg(my_kernel,0,sizeof(cl_mem), &DeviceBuffer );


clEnqueueNDRangeKernel(queue, my_kernel,…)
```

SHARE VIRTUAL MEMORY (SVM)

FINE GRAIN BUFFER SVM VS CL1.2

AMD

```
//CL_MEM_SVM_FINE_GRAIN_BUFFER means host and device can
//concurrently access the buffer


float* Buffer = (float*)clSVMAlloc(ctx, CL_MEM_READ_WRITE |
                                   CL_MEM_SVM_FINE_GRAIN_BUFFER,
                                   1024 * sizeof(float), 0);


//fill the buffer from host

for (int i=0; i<1024; i++)

          Buffer[i] = ….;


// use your SVM buffer in you OpenCL kernel on device
directly

clSetKernelArgSVMPointer(my_kernel, 0, Buffer);


clEnqueueNDRangeKernel(queue, my_kernel,…)
```

```
//create device buffer


cl_mem DeviceBuffer =  clCreateBuffer(ctx,
          CL_MEM_READ_WRITE, 1024*sizeof(float), NULL, &err
);


//create host buffer

float* hostBuffer = new float[1024];

for (int i=0; i<1024; i++)

          hostBuffer ] = ….;

//data transfer happens here

clEnqueueWriteBuffer(queue, DeviceBuffer,… , hostBuffer);



//use our device buffer on device

clSetKernelArg(my_kernel,0,sizeof(cl_mem), &DeviceBuffer );


clEnqueueNDRangeKernel(queue, my_kernel,…)
```

SHARE VIRTUAL MEMORY (SVM)

FINE GRAIN SYSTEM

```
//no more OpenCL API needed to allocate data, simply use your favorite memory allocation function : new, malloc…
float* Buffer = (float*)malloc(1024*sizeof(float))

//fill the buffer from host
for (int i=0; i<1024; i++)
          Buffer[i] = ….;

// use your SVM buffer in you OpenCL kernel on device directly
clSetKernelArgSVMPointer(my_kernel, 0, Buffer);

clEnqueueNDRangeKernel(queue, my_kernel,…)
```

SHARE VIRTUAL MEMORY (SVM)

- https://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf
- http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/
  - Samples : GlobalMemoryBandwidth, DeviceEnqueueBFS, SVMBinaryTreeSearch, RangeMinimumQuery, SVMAtomicsBinaryTreeInsert (APU only), FineGrainSVM (APU only)

PIPES

- **Act like a queue object (FIFO) between kernels.**
- **Pipes objects are created on host….**
  - clCreatePipe(cl_context ctx, cl_mem_flags flags, cl_uint packet_size, cl_uint max_packets, cl_pipe_properties*, cl_int*)
- **….But they cannot be accessed from host (read and write)**
  - The only valid memory flag for clCreatePipe is CL_MEM_HOST_NO_ACCESS
- **Pipes can either be read_only or write_only within a kernel**
- **Pipes can only be coming from a kernel/functions arguments**
  - Pipes can't be created locally in a function/kernel
- **Pipes can only be used through built-in CL2.0 functions**
  - read_pipe (pipe p, reserve_id_t reserve_id, uint index, gentype *ptr): for reading 1 packet from pipe p into ptr.
  - write_pipe (pipe p, reserve_id_t reserve_id, uint index, gentype *ptr): for writing 1 packet
- **Pipes don't define any ordering for read/write operations amongst all the threads running. It is up to the developers to control this if needed**

PIPES

AMD

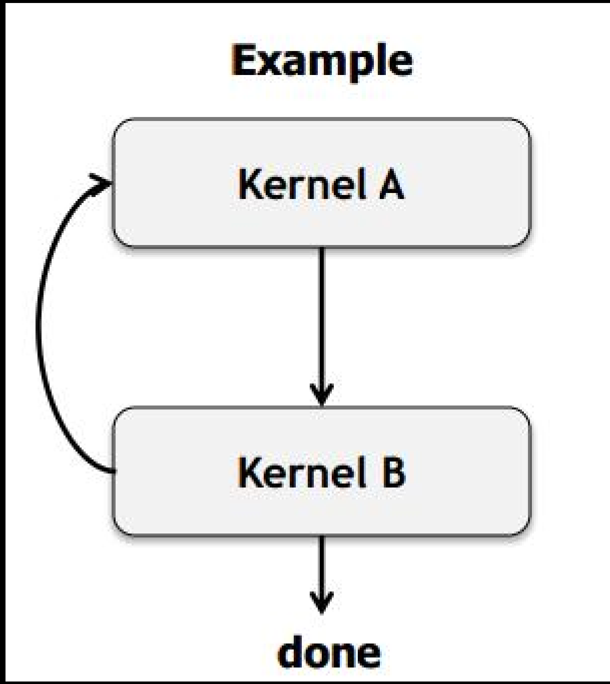
```
__kernel void pipeWrite(__global int *src, __write_only pipe int out_pipe)
{
  int gid = get_global_id(0);
  reserve_id_t res_id;
  res_id = reserve_write_pipe (out_pipe, 1);

  if( is_valid_reserve_id (res_id))
  {
    if( write_pipe (out_pipe, res_id, 0, &src[gid]) != 0)
    {
      return;
    }
    commit_write_pipe (out_pipe, res_id);
  }
}
```

PIPES
REFERENCES

- https://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf
- http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/
  - Samples simplePipe and DeviceEnqueueBFS

NESTED PARALLELISM

- **In OpenCL 1.2 only the host can enqueue kernels**
- **Iterative algorithm example**
  - kernel A queues kernel B
  - kernel B decides to queue kernel A again
- **A very simple but extremely common nested parallelism example**



NESTED PARALLELISM

AMD

- **Allow a device to queue kernels to itself**
  - Allow a work-item(s) to queue kernels
- **Use similar approach to how host queues commands**
  - Queues and Events
  - Event and Profiling functions

NESTED PARALLELISM

- Use clang Blocks to describe kernel to queue

```
kernel void my_func(global int *a, global int *b)
{
 …
 void (^my_block_A)(void) =
 ^
 {
   size_t id = get_global_id(0);
   b[id] += a[id];
 };

 enqueue_kernel(get_default_queue(),
  CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
  ndrange_1D(…),
  my_block_A);
}
```

NESTED PARALLELISM

2 API

```
int enqueue_kernel(queue_t queue,
                   kernel_enqueue_flags_t flags,
                   const ndrange_t ndrange,
                   void (^block)())

int enqueue_kernel(queue_t queue,
                   kernel_enqueue_flags_t flags,
                   const ndrange_t ndrange,
                   uint num_events_in_wait_list,
                   const clk_event_t *event_wait_list,
                   clk_event_t *event_ret,
                   void (^block)())
```

NESTED PARALLELISM

QUEUING KERNELS WITH POINTERS TO LOCAL ADDRESS SPACE AS ARGUMENTS

```
int enqueue_kernel(queue_t queue,
                   kernel_enqueue_flags_t flags,
                   const ndrange_t ndrange,
                   void (^block)(local void *, …), uint size0, …)
```

```
int enqueue_kernel(queue_t queue,
                   kernel_enqueue_flags_t flags,
                   const ndrange_t ndrange,
                   uint num_events_in_wait_list,
                   const clk_event_t *event_wait_list,
                   clk_event_t *event_ret,
                   void (^block)(local void *, …), uint size0, …)
```

NESTED PARALLELISM

```
void my_func_local_arg (global int *a, local int *lptr, …) { … }

kernel void my_func(global int *a, …)
{
 …
 uint local_mem_size = compute_local_mem_size(…);
 enqueue_kernel(get_default_queue(),
                CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                ndrange_1D(…),
                ^(local int *p){my_func_local_arg(a, p, …);},
                 local_mem_size);
}
```

NESTED PARALLELISM

- **Specify when a child kernel can begin execution (pick one)**
  - Don’t wait on parent
  - Wait for kernel to finish execution
  - Wait for work-group to finish execution
- **A kernel’s execution status is complete**
  - when it has finished execution
  - and all its child kernels have finished execution

NESTED PARALLEISM

AMD

- **Other Commands**
  - Queue a marker
- **Query Functions**
  - Get workgroup size for a block
- **Event Functions**
  - Retain & Release events
  - Create user event
  - Set user event status
  - Capture event profiling info
- **Helper Functions**
  - Get default queue
  - Return a 1D, 2D or 3D ND-range descriptor

NESTED PARALLELISM

- https://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf
- http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/
  – Samples DeviceEnqueueBFS, ExtractPrimes, RegionGrowingSegmentation, BinarySearchDeviceSideEnqueue

WORK GROUP FUNCTION

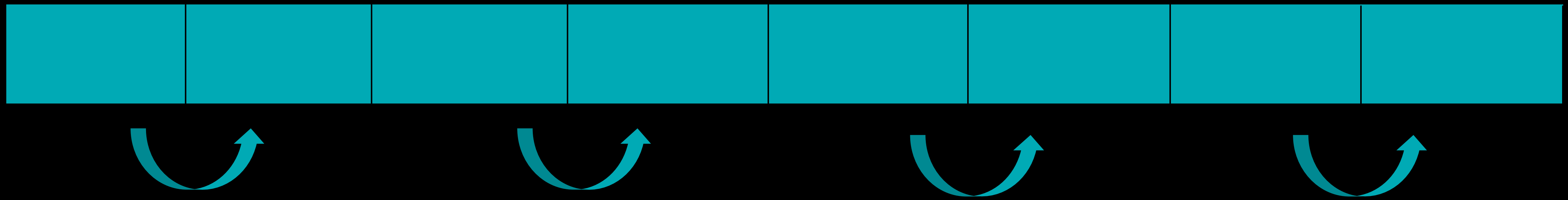
- **Scan**
  - work_group_scan_exclusive<op>
  - work_group_scan_inclusive<op>
- **Reduce**
  - work_group_reduce<op>
- **Voting functions**
  - work_group_all
  - work_group_any
- **Broadcast**
  - work_group_broadcast

WORK GROUP FUNCTION

PREFIX SUM

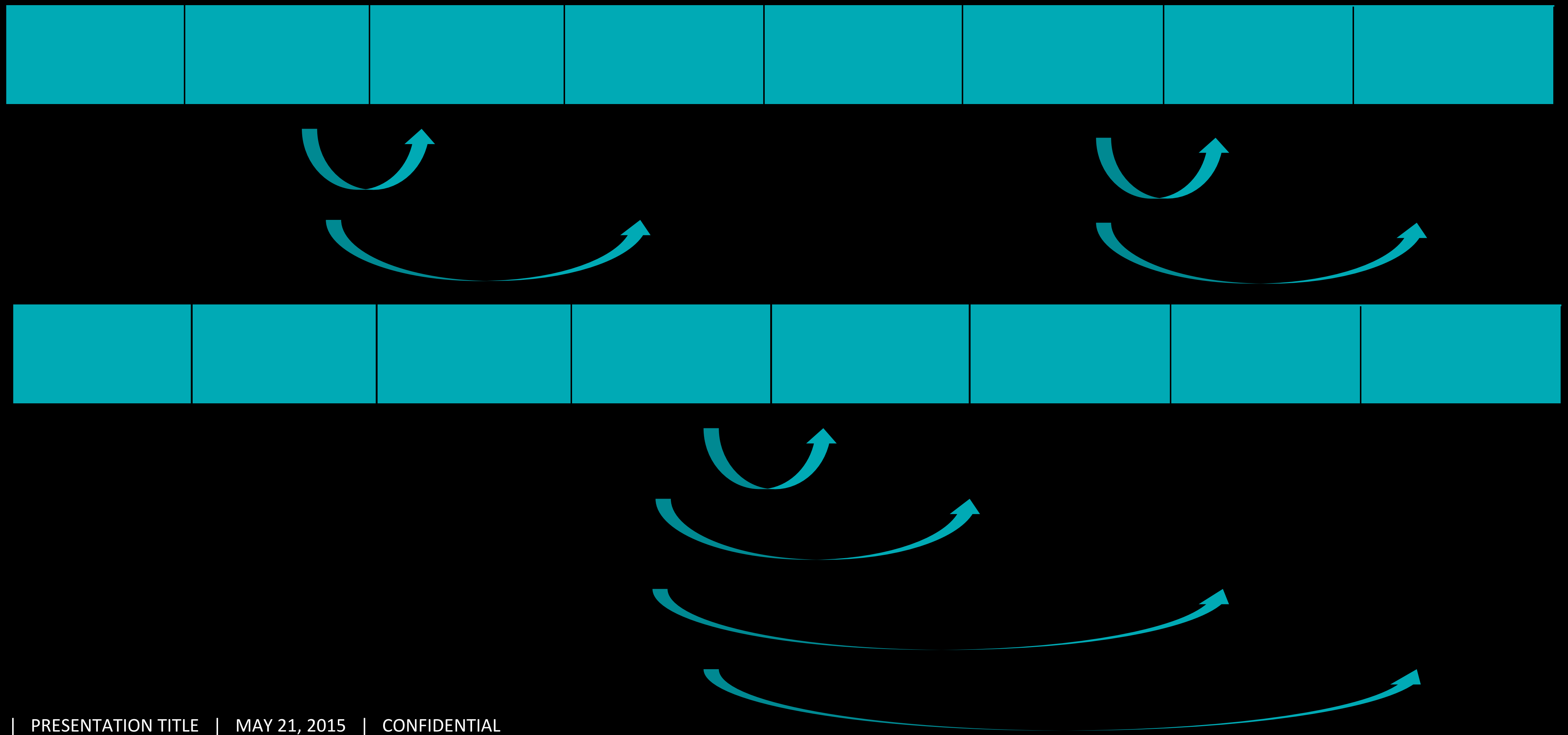
```
__kernel void group_scan_kernel(__global float *in, __global float *out)
{
  float  in_data;
  int i = get_global_id(0);
  in_data = in[i];
  out[i]  = work_group_scan_inclusive_add(in_data);
}
```

- Once we have the scan for each work group, we need to sum up the “next group” with the last value of the previous one



WORK GROUP FUNCTION
PREFIX SUM

- This operation needs to be repeated



WORK GROUP FUNCTION

PREFIX SUM

```
__kernel void global_scan_kernel(__global float *out, unsigned int stage)
{
  …
  /* find the element to be added */
  l       = (grid >> stage);
  prev_gr = l*(vlen << 1) + vlen - 1;
  prev_el = prev_gr*szgr + szgr - 1;
  if (lid == 0)
    add_elem    = out[prev_el];

  work_group_barrier(CLK_GLOBAL_MEM_FENCE|CLK_LOCAL_MEM_FENCE);
  add_elem = work_group_broadcast(add_elem,0);

  /* find the array to which the element to be added */
  curr_gr = prev_gr + 1 + (grid % vlen);
  curr_el = curr_gr*szgr + lid;
  out[curr_el] += add_elem;
}
```

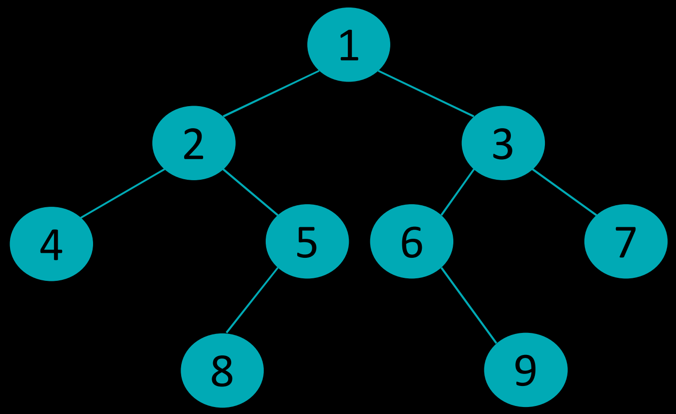
WORK GROUP FUNCTION

- https://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf
- http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/
  - Samples DeviceEnqueueBFS, BuiltInScan, RegionGrowingSegmentation, ExtractPrimes

NESTED PARALLELISM + PIPES + WORK-GROUP FUNCTIONS

BREADTH FIRST SEARCH

- **BFS is a strategy for searching in a graph. It begins at the root node and inspects all the neighbouring nodes. Then for each of those nodes it inspects their neighbour nodes and so on.**

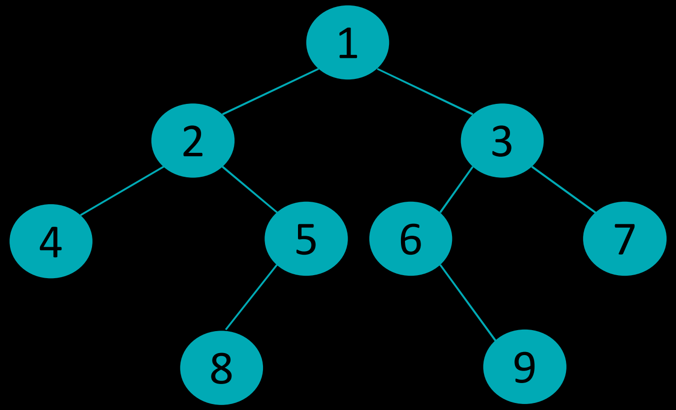


- **The classic serial algorithm uses a queue(fifo) to store the non treated nodes of the graph. Once a node is visited, it is popped out from the queue. We then look for its neighbour nodes to add in the queue.**

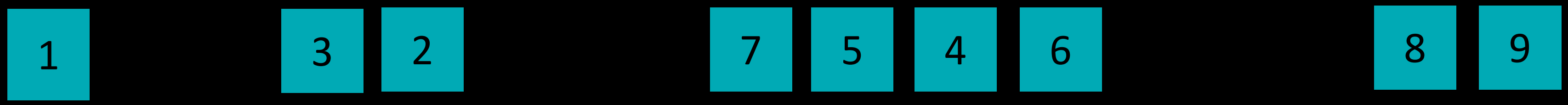
NESTED PARALLELISM + PIPES + WORK-GROUP FUNCTIONS

BREADTH FIRST SEARCH

- **We will use 2 OpenCL pipe objects to simulate our queue**
  - Nodes of the current of level (read pipe)
  - Nodes of the next level (write pipe)
- **We will parallelize the visit of a given level**
  - Each kernel launch will only work on a given level
  - Each thread will treat one node
- **We use the nested parallelism to enqueue a new kernel to work on the next level**



- **Pipe states :**



NESTED PARALLELISM + PIPES + WORK-GROUP FUNCTIONS

READING CURRENT LEVEL, ONE NODE PER WORK-ITEM

```
__kernel
void deviceEnqueueBFSKernel(__global uint *d_rowPtr, __global uint *d_colIndex, __global uint *d_dist,
                            __read_only pipe uint d_vertexFrontier_inPipe,
                            __write_only pipe uint d_edgeFrontier_outPipe, uint parentNodeLevel )
{
        …
        atomic_store_explicit(&g_totalNeighborsCount,0,memory_order_seq_cst, memory_scope_device);
        // read current level's vertices to be visited (/* reading from pipe */)
        res_read_id = reserve_read_pipe(d_vertexFrontier_inPipe, 1);
        if(is_valid_reserve_id(res_read_id))
        {
                if(read_pipe(d_vertexFrontier_inPipe, res_read_id, 0, &node) != 0)
                {
                        return;
                }
                commit_read_pipe(d_vertexFrontier_inPipe, res_read_id);
        }
```

NESTED PARALLELISM + PIPES + WORK-GROUP FUNCTIONS

WRITING CHILD NODE INTO THE SECOND PIPE

```
// we first checked whether node is visited and got the number of child
// expand these neighbours for the next level, only when it has not been visited  (/* Writing into Pipe */)
for(int i = 0; i < numChildPerNode; i++)
{
        childNode = getChildNode(d_colIndex, offset+i);
        if(d_dist[childNode] == INIFINITY)
        {
                res_write_id = reserve_write_pipe(d_edgeFrontier_outPipe, 1);
                if(is_valid_reserve_id(res_write_id))
                {
                        if(write_pipe(d_edgeFrontier_outPipe, res_write_id, 0, &childNode) != 0)
                        {
                                return;
                        }
                        commit_write_pipe(d_edgeFrontier_outPipe, res_write_id);
                }
                tmpNeighborsCount++;
        }
```

NESTED PARALLELISM + PIPES + WORK-GROUP FUNCTIONS

COMPUTING THE NUMBER OF CHILD NODES AT THE NEXT LEVEL

```
        //summing number of Neighbours within work group
        wgCnt = work_group_reduce_add(tmpNeighborsCount);
}
//summing total number of Neighbours across all work-groups
if(lid == 0)
{
        atomic_fetch_add_explicit(&g_totalNeighborsCount, wgCnt, memory_order_seq_cst, memory_scope_device
}
```

NESTED PARALLELISM + PIPES + WORK-GROUP FUNCTIONS

RELAUNCH THE NEW KERNEL

```
if(gid == 0) //only one work item will enqueue a new kernel
{
          globalThreads = 1;
          currentLevel = d_dist[node];
          queue_t q = get_default_queue();
          ndrange_t ndrange1 = ndrange_1D(globalThreads);

          void (^bfsDummy_device_enqueue_wrapper_blk)(void) = ^{deviceEnqueueDummyKernel(…
                                                                               d_edgeFrontier_outPipe,
                                                                               d_vertexFrontier_inPipe,
                                                                               currentLevel );};
          int err_ret = enqueue_kernel(q, CLK_ENQUEUE_FLAGS_WAIT_KERNEL, ndrange1, bfsDummy_device_enqueue_wrapper_blk);

if(err_ret != 0)
 {
                    return;
 }
```

NESTED PARALLELISM + PIPES + WORK-GROUP FUNCTIONS

LAUNCH MAIN KERNEL WITH THE NUMBER OF CHILD NODES

```
void deviceEnqueueDummyKernel(…)
{
  uint globalThreads = atomic_load_explicit(&g_totalNeighborsCount, memory_order_seq_cst, memory_scope_device);

  if(globalThreads == 0)   // don't need to launch kernel if there is no child
    return;

  queue_t q = get_default_queue();
  ndrange_t ndrange1 = ndrange_1D(globalThreads);

  void (^bfs_device_enqueue_wrapper_blk)(void) = ^{ deviceEnqueueBFSKernel (d_rowPtr,
                                                                           d_colIndex,
                                                                           d_dist,
                                                                           d_edgeFrontier_outPipe,
                                                                           d_vertexFrontier_inPipe,
                                                                           parentNodeLevel );};
  int err_ret = enqueue_kernel (q, CLK_ENQUEUE_FLAGS_WAIT_KERNEL, ndrange1, bfs_device_enqueue_wrapper_blk);
```

GENERIC ADDRESS SPACE

AMD

- **In OpenCL 1.2, function arguments that are a pointer to a type must declare the address space of the memory region pointed to**
- **Many examples where developers want to use the same code but with pointers on different address spaces**

```
void
my_func (local int *ptr, …)
{
 …
 foo(ptr, …);
 …
}
```

```
void
my_func (global int *ptr, …)
{
 …
 foo(ptr, …);
 …
}
```

- **Above example is not supported in OpenCL 1.2**
- **Results in developers having to duplicate code, which prone to errors**

GENERIC ADDRESS SPACE

AMD

- **OpenCL 2.0 no longer requires an address space qualifier for arguments to a function that are a pointer to a type**
  - Except for kernel functions
- **Generic address space assumed if no address space is specified**
- **Makes it really easy to write functions without having to worry about which address space arguments point to**

```
void
my_func_generic_pointer (int *ptr, …)
{
 …
}

kernel void
foo(global int *g_ptr, local int *l_ptr, …)
{
 …
 my_func_generic_pointer (g_ptr, …);
 my_func_generic_pointer (l_ptr, …);
}
```

GENERIC ADDRESS SPACE

- https://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf
- http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/
  - Sample : SimpleGenericAddressSpace