Raytracing with OpenCL

Andy Haslam

Introduction to OpenCL

“The open standard for parallel programming of heterogeneous systems”

- Open standard for programming on ‘devices’ with many cores, such as GPUs

- Language used is based on C99, but has several significant differences

- Interfaces with ‘host’, using C

Introduction to OpenCL Kernels

Similar to functions in C, but executed in parallel

- May be programmed to access different sections of data based on worker IDs

- Kernels may not be called from within kernels, although standard functions may be used

Shared array (slow)

Passed by value (fast)

```
__kernel void exampleKernel(__global int* array, int value)
{
    array[get_local_id(0)] = value;
}
```

Used to differentiate between threads

Accessing Kernels From C

Harnessing the power of OpenCL in five easy steps

1) Create device context and command queue

2) Build kernels from source code

3) Create device memory and set kernel arguments

4) Execute kernels and wait for completion

5) Copy results from device memory

Accessing Kernels From C

1) Create device context and command queue

```
size_t                  dataBytes;
cl_uint                 numberOfPlatforms;
cl_context_properties   properties[3];


clGetPlatformIDs(1, clrPlatformIds, &numberOfPlatforms);
properties[0] = (cl_context_properties)CL_CONTEXT_PLATFORM;
properties[1] = (cl_context_properties)clrPlatformIds[0];
properties[2] =(cl_context_properties)0;
deviceContext = clCreateContextFromType(properties, CL_DEVICE_TYPE_CPU, NULL, NULL, NULL);
clGetContextInfo(deviceContext, CL_CONTEXT_DEVICES, 0, NULL, &dataBytes);
deviceIds = (cl_device_id *)malloc(dataBytes);
clGetContextInfo(deviceContext, CL_CONTEXT_DEVICES, dataBytes, deviceIds, NULL);
commandQueue = clCreateCommandQueue(deviceContext, deviceIds[0], 0, NULL);
```

Accessing Kernels From C

2) Build kernels from source code

```
cl_program program = clCreateProgramWithSource( deviceContext,
                                                1,
                                                (const char **)&sourceString,
                                                (const size_t *)&sourceSize,
                                                NULL);
clBuildProgram(program, 1, &deviceIds[0], passedOptions, NULL, NULL);
exampleKernel = clCreateKernel(program, “example", NULL);
clReleaseProgram(program);
```

Accessing Kernels From C

3) Create device memory and set kernel arguments

```
int    deviceMemoryValue = 10,
       numberOfWorkers   = 100;
cl_mem deviceMemoryArray = clCreateBuffer(deviceContext,
                                          CL_MEM_WRITE_ONLY,
                                          numberOfWorkers*sizeof(int),
                                          NULL,
                                          NULL);
clSetKernelArg(exampleKernel, 0, sizeof(cl_mem), (void *)&deviceMemoryArray);
clSetKernelArg(exampleKernel, 1, sizeof(int), (void *)&deviceMemoryValue);
```

Accessing Kernels From C

4) Execute kernels and wait for completion

```
clEnqueueNDRangeKernel(commandQueue,
                       exampleKernel,
                       1,
                       NULL,
                       &numberOfWorkers,
                       &numberOfWorkers,
                       0,
                       NULL,
                       NULL);

clFinish(commandQueue);
```

Workgroup Dimensions

Total workers

Total workers in workgroup

Accessing Kernels From C

5) Copy results from device memory

```
int* hostMemoryArray = (int*)malloc(numberOfWorkers*sizeof(int));

clEnqueueReadBuffer(  commandQueue,
                      deviceMemoryArray,
                      CL_TRUE,
                      0,
                      numberOfWorkers*sizeof(int),
                      hostMemoryArray,
                      0,
                      NULL,
                      NULL);


clFinish(commandQueue);
```

Key Differentiators of OpenCL

What makes OpenCL different/(harder!?)

- No recursion within kernels *(but you can call helper functions)*
- No dynamic memory (no pointers, no malloc etc.)
- No standard C headers *(but you can create your own)*
- OpenCL-specific data types, including vectors (e.g. float3)
- OpenCL-specific functions, including vector operations (e.g. cross)

Considerations for Raytracing with OpenCL

Thinking about those differences...

- Acceleration structures without pointers or recursion
- Tracing of non-primary rays without recursion
- Taking advantage of OpenCL-specific data types and functions

Considerations for Raytracing with OpenCL

Acceleration structures without pointers or recursion

1) Represent all triangles in single array

2) Recursively build bounding volume hierarchy on host as array of bounding boxes (using array indices in place of pointers)

3) Copy BVH to device

4) Using a local traversal stack (array whose size is declared in header), traverse BVH within OpenCL kernel on device

Considerations for Raytracing with OpenCL

Tracing of non-primary rays without recursion

```
cameraRaysKernel(input camera, output rays);
for(number of recursions)
{
   intersectionsKernel(input rays, output hits);
   shadeKernel(input hits, output rays, output pixels);
}
```

Considerations for Raytracing with OpenCL

Tracing of non-primary rays without recursion (path tracing)

```
for(number of rays per pixel)
{

   cameraRaysKernel(input camera, output rays);

   for(number of recursions)
   {

      intersectionsKernel(input rays, output hits);

      for(each light)
      {

         lightRaysKernel(input hits, input light, output lightRays);
         intersectionsKernel(input lightRays, output lightHits);
         shadeKernel(input hits, input lightHits, input lightRays, input light, output rays, output pixels);

      }

   }

}
```

Considerations for Raytracing with OpenCL

Tracing of non-primary rays without recursion (shading kernel)

Within `shadeKernel`:

```
for(number of pixels)
{
   if(hits[pixelNumber].triangle == lightHits[pixelNumber].triangle)
   {
      pixels[pixelNumber] += shadeColor(hits[pixelNumber].normal,
                                        lightRays[pixelNumber].direction,
                                        light);
      rays[pixelNumber]   =  newRandomRay(hits[pixelNumber]);
   }
}
```

Considerations for Raytracing with OpenCL

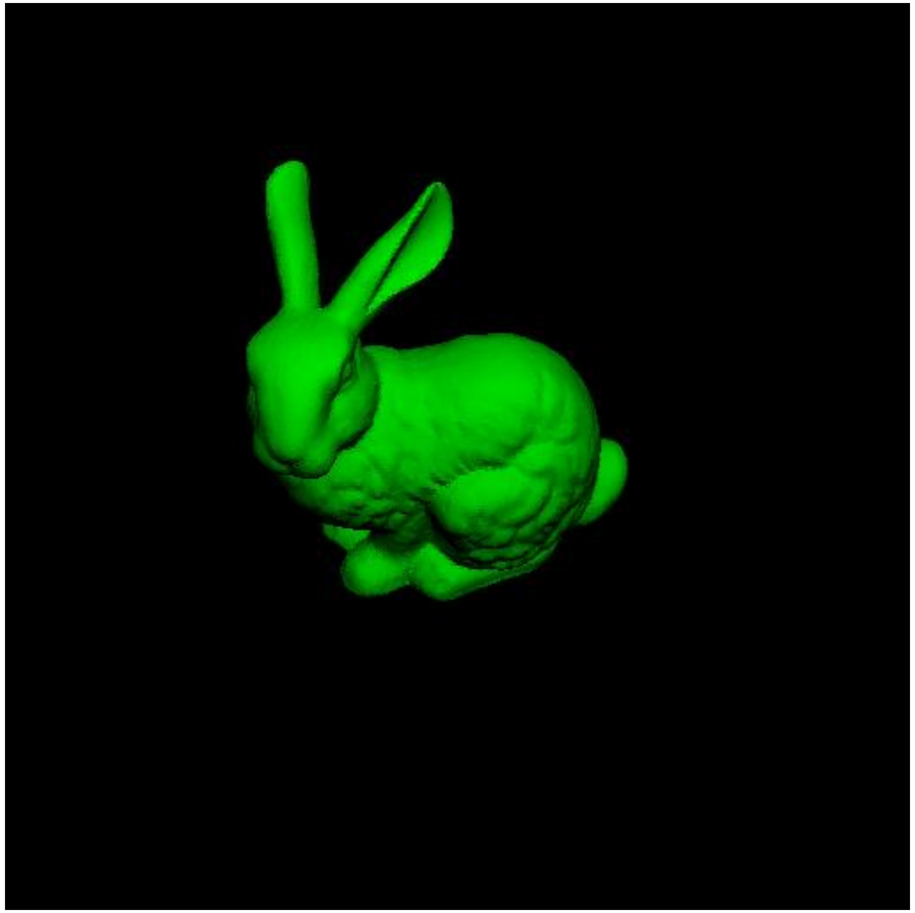
Taking advantage of OpenCL-specific data types and functions

- Within C, vectors must be built from user-defined structs
  - Better to use float3 and native operations in OpenCL
  - Create converter functions!

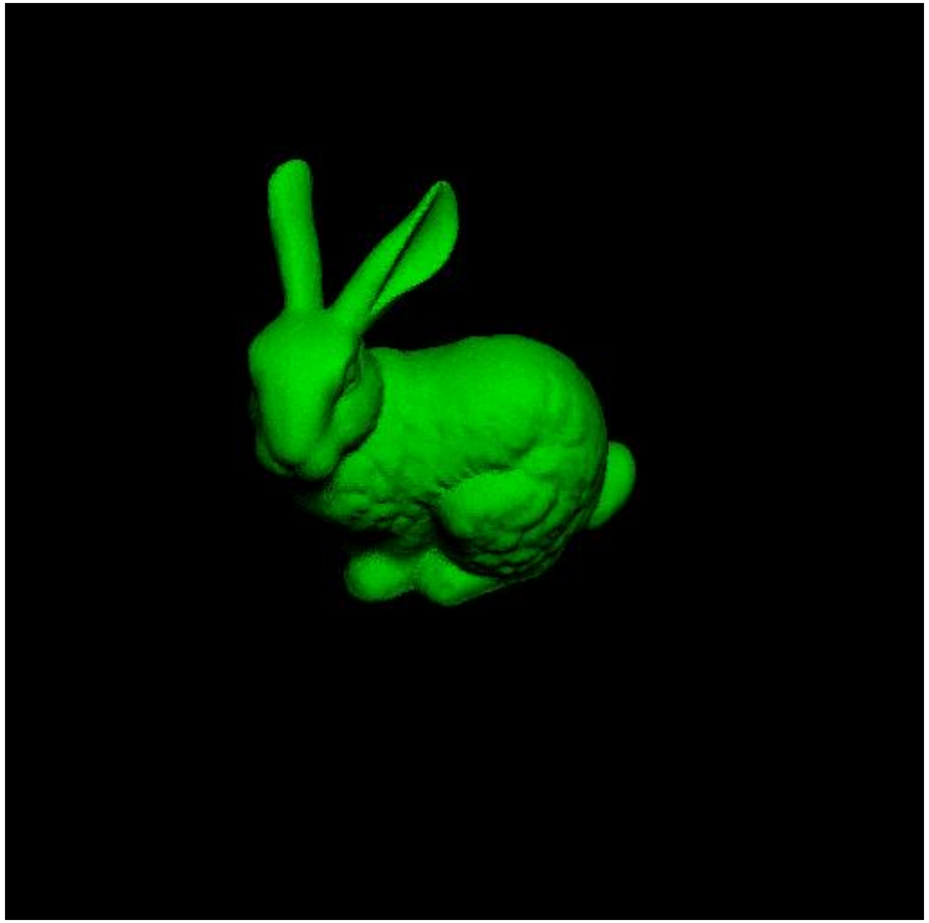
- Some data may only ever exist on the device:-
  - Rays, hits and allLightHits – all of which use vectors
  - Structs can be built with float3 to prevent conversion!
  - Headers must only be included in OpenCL C files to prevent compile errors
  - Size must be known and hardcoded when defining device memory in C

Demonstration

See it in action…



Without shadows



With soft shadows