L16: Sorting and OpenGL Interface

Administrative
- STRSM due March 23 (EXTENDED)
- Midterm coming
  - In class March 28, can bring one page of notes
  - Review notes, readings and review lecture
  - Prior exams are posted
- Design Review
  - Intermediate assessment of progress on project, oral and short
    - In class on April 4
- Final projects
  - Poster session, April 23 (dry run April 18)
  - Final report, May 3

Sources for Today's Lecture

OpenGL Rendering
http://www.nvidia.com/content/cudazone/download/
Advanced_CUDA_Training_NVISION08.pdf
Chapter 3.2.7.1 in the CUDA Programming Guide

Sorting
- (Bitonic sort in CUDA SDK)
  http://www.ce.chalmers.se/~uffe/hybridsortElsevier.pdf

OpenGL Rendering
- OpenGL buffer objects can be mapped into the CUDA address space and then used as global memory
  - Vertex buffer objects
  - Pixel buffer objects
- Allows direct visualization of data from computation
  - No device to host transfer
  - Data stays in device memory - very fast compute / viz cycle
- Data can be accessed from the kernel like any other global data (in device memory)
**OpenGL Interoperability**

1. Register a buffer object with CUDA
   - `cudaGLRegisterBufferObject(GLuint buffObj);`
   - OpenGL can use a registered buffer only as a source
   - Unregister the buffer prior to rendering to it by OpenGL
2. Map the buffer object to CUDA memory
   - `cudaGLMapBufferObject(void** devPtr, GLuint buffObj);`
   - Returns an address in global memory Buffer must be registered prior to mapping
3. Launch a CUDA kernel to process the buffer
   - Unmap the buffer object prior to use by OpenGL
   - `cudaGLUnmapBufferObject(GLuint buffObj);`
4. Unregister the buffer object
   - `cudaGLUnregisterBufferObject(GLuint buffObj);`
   - Optional: needed if the buffer is a render target
5. Use the buffer object in OpenGL code

---

**Example from simpleGL in SDK**

1. GL calls to create and initialize buffer, then registered with CUDA:
   ```
   // create buffer object
   glGenBuffers( 1, vbo);
   glBindBuffer( GL_ARRAY_BUFFER, *vbo);
   unsigned int size = mesh_width * mesh_height * 4 * sizeof( float)*2;
   glBufferData( GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW);
   glBindBuffer( GL_ARRAY_BUFFER, 0);
   // register buffer object with CUDA
   cudaGLRegisterBufferObject(*vbo);
   ```

---

**Example from simpleGL in SDK, cont.**

2. Map OpenGL buffer object for writing from CUDA
   ```
   float4 *dptr;
   cudaGLMapBufferObject( (void**)&dptr, vbo);
   ```
3. Execute the kernel to compute values for dp
   ```
   dim3 block(8, 8, 1);
   dim3 grid(mesh_width / block.x, mesh_height / block.y, 1);
   kernel<<<grid, block>>>(dptr, mesh_width, mesh_height, anim);
   ```
4. Unregister the OpenGL buffer object and return to OpenGL
   ```
   cudaGLUnmapBufferObject(vbo);
   ```

---

**Key issues in sorting?**

- Data movement requires significant memory bandwidth
- Managing global list may require global synchronization
- Very little computation, memory bound
Hybrid Sorting Algorithm, Key Ideas

- Imagine a "recursive" algorithm
  - Use different strategies for different numbers of elements
  - Algorithm depends on how much work, and how much storage was required
- Here we use different strategies for different-sized lists
  - Very efficient sort for float4
  - Use shared memory for sublists
  - Use global memory to create pivots

Hybrid Sorting Algorithm (Sintorn and Assarsson)

- Each pass:
  - Merge 2L sorted lists into L sorted lists
- Three parts:
  - Histogramming: to split input list into L independent sublists for Pivot Points
  - Bucketsort: to split into lists than can be sorted using next step
  - Vector-Mergesort:
    - Elements are grouped into 4-float vectors and a kernel sorts each vector internally
    - Repeat until sublist is sorted
- Results:
  - 20% improvement over radix sort, best GPU algorithm
  - 6-14 times faster than quicksort on CPU

Sample Sort (Detailed slides)

- Divide and Conquer
  - Input as an array
  - Identifying number and size of divisors or buckets
- Histogramming in global memory constructs buckets for the elements.
- A priori select pivot values – if this results in load imbalance, update pivots and repeat

Hybrid Sort

- To handle the buckets each thread does the following:
- Bring in the elements from the input array into its shared memory
- Use merge-sort to sort its local array.
- Pushes the elements in output array in appropriate location.
Sort two vectors from A & B (Bitonic Sort)

// get the four lowest floats
a.xyzw = (a.xyzw < b.wzyx) ? a.xyzw : b.wzyx

// get the four highest floats
b.xyzw = (b.xyzw >= a.wzyx) ? b.xyzw : a.wzyx

Call sortElements(a);
Call sortElements(b);

Key Computation: Vector MergeSort

Idea: Use vector implementation to load 4 elements at a time, and "swizzling" to move vector elements around

Output: a sorted vector of four elements for
[2, 6, 3, 1]

// Bitonic sort within a vector
// Meaning: r.xyzw is original order; r.wzyx is reversed order
sortElements(float4 r) {
    r = (r.xyzw > r.yxwz) ? r.yyww : r.xxzz
    r = (r.xyzw > r.zwxy) ? r.zwzw : r.xyxy
    r = (r.xyzw > r.xzyw) ? r.xzzw : r.xyyw
}

Working Through An Example

// get four lowest elements
a.xyzw = (a.xyzw < b.wzyx) ? a.xyzw : b.wzyx
a = (2,6,9,10)
b = (1,3,8,11)

[2,6,9,10] < (11,8,3,1) becomes
2 < 11 ? 2 : 11 -> 2
6 < 8 ? 6 : 8 -> 6
9 < 3 ? 9 : 3 -> 3
10 < 1 ? 10 : 1 -> 1
**Summary**

- **OpenGL rendering**
  - Key idea is that a buffer can be used by either OpenGL or CUDA, but only one at a time.
  - Protocol allows memory mapping of buffer between OpenGL and CUDA to facilitate access.

- **Hybrid sorting algorithm**
  - Histogram constructed in global memory to identify pivots.
  - If load is imbalanced, pivots are revised and step repeated.
  - Bucket sort into separate buckets.
  - Then, sorted buckets can be simply concatenated.
  - MergeSort within buckets.
    - Vector sort of float4 entries.
    - Vector sort of pair of float4s.