Some notes on debugging CUDA programs

Here are some debugging tips that came up in office hours today.

Tracking Down a Bug

Let's say you run your kernel (e.g. your convolution kernel) and you get a wrong result. How can you track it down? From Chenxi's program, you know which pixel is wrong. It would be nice if you knew what the correct value is. If anybody knows how to find that, I'd be delighted to hear the hint. In the meantime, you at least know which pixel it is. If you have your non-tiled version of the convolutions working, then you can use those to find out what the pixel value should be.

You know the index of the wrong pixel in the array that you copied to and from the GPU. Now, find out which thread wrote the value. Basically, we're trying to figure out which of two conjectures to pursue:

1. Two or more threads tried to write the value. This creates a race condition, and the "wrong" thread ends up writing the value.

2. The right thread wrote the value, but it the value is wrong.

The second conjecture is the one you're used to from sequential computing – in office hours, it seemed to be the one most students in the class pursued first (or exclusively). The first conjecture is a bug that arises because CUDA is parallel. It also comes up because CUDA is not functional: one thread can modify a value that was established by another thread.

I'll start with the first conjecture – how can we determine if two or more threads are writing to the same location? First, you find a pixel that gets the wrong value. Determine its index in the global array. Now, find all assignments in your CUDA code of the form:

```
dev_mem[index] = someValue;
```

Let's say that you know that the element with index `123456` gets the wrong value. Then, you can add some code like:

```
if(index == 123456) {
   printf("dev_mem[%d] = %f; blockIdx.x = %d, blockIdx.y = %d, threadIdx.x = %d, threadIdx.y = %d\n",
          index, someValue, blockIdx.x, blockIdx.y, threadIdx.x, threadIdx.y);
}
dev_mem[index] = someValue;
```

When you run the code, is the `printf` executed for just one thread, or is it executed for two or more? If more than one thread executes the `printf`, then more than one thread is trying to set that value in `dev_mem`. That's probably not what you want. You now need to figure out why multiple threads are trying to write to the same location.

If exactly one thread writes the specified index, is it the thread you intended? If not, you've probably got some indexing problem to fix. If it is the right thread, you can go to the "second conjecture" below.

There's one more possibility: no thread writes to the location. If that's the case, then you need to figure out why that happened. This is probably some indexing problem. You can determine what thread should have written to it, and continue with my description about the second kind of error.

Now, we're ready for the second kind of error: you know what thread is causing the error. Good news – you now have a sequential debugging problem (well, mostly). Because you know the block and thread indices for the thread, you can write if-statements that only execute for that thread. You can then add `printf` statements to trace the execution of that thread. Of course, a problem can be that it's getting the wrong value from shared memory. In that case, you need to figure out which thread is writing to the shared memory location in question.

Finding the writer to a shared memory location is very much like what we just did to figure out which thread was writing to the global memory. You first want to figure out if the location is written by exactly one thread (and if that's the thread you intended) or if you have a race problem that two or more threads are writing to the location, or you have a problem that no thread is writing to the location. Because shared

memory is local to a CUDA block, you only need to consider threads in the same block as the one that was reading the bad value from shared memory.

In CUDA, you can set a `float *` to the address of the element in shared memoroy that gets the wrong value. Using the methods described above, you can figure out its index (or indices if your array is two or more dimensional). Then, at the beginning of your kernel, you can get the address of that location:

```
float *trigger = &(sh_mem[YouWriteThis][AndThis]);
```

Now, anyplace you have an assignment to `sh_mem`, you can add tracing code:

```
if(&(sh_mem[index1][index2]) == trigger) {
   printf("sh_mem[%d][%d]) = %f; threadIdx.x=%d, threadIdx.y=%d\n",
          index1, index2, SomeValue, threadIdx.x, threadIdx.y);
}
sh_mem[index1][index2]) = SomeValue;
```

The main idea is to figure out which thread is writing the bad value. Don't assume that you know for sure. Add some tracing code, and check. If you see two threads writing to the same location, that probably means you have a bug. Once you know which thread is writing the bad value, you can trace the execution of that thread to find the bug.

Test Cases For Convolution

You've got the sample images. How can you test the code on something simpler? The main thing to notice is that convolution is a linear operator. You can figure out a lot with a few simple test cases. For example, you can allocate a *small* array for data. For example, when testing horizontal convolution, make the array three times the your block width. Why three? Then you see the block on the left, a block in the middle, and a block on the right. Having one row is probably a good place to start. In some cases, you may need two or three rows if an indexing error causes a "wrap-around" bug where the thread for the right-end of a row reads or writes a pixes from the left end of the next row, or vice-versa. If you make the array small, you can print it out to see the results. This is a little messier than I would like. The convolution stencils for this problem are 21 pixels wide. You probably want (or require) your blocks to be at least as wide as the stencil. Three blocks that are 25 pixels wide gives you 75 pixels in a row (minus some for overlap).

Next, create the world's simplest stencil. What's that? A good starting point is a stencil that has `1.0` for the middle element (i.e. `stencil[10]`) and `0.0` everywhere else. Convolving with this stencil is the identity function – the output image should be the same as the input. That's easy to check.

If the identity-stencil works, then try a stencil with a `1.0` in some other location and `0.0` everywhere else. This simply shifts the image to the left or the right. Trying a few cases should help you find if there are bugs in your halo code.

In most cases, these simple stencils should be enough. If they aren't you can take advantage of the fact the convolution is linear. Choose *two* locations in the stencil to have two different non-zero values, and make the rest of the stencil 0. The output image should be the weighted sum (with the weights set by the two values) of the input image shifted by amounts determined by the positions of the two non-zero values.

Have Fun!