CS194-15: Engineering Parallel Software Assignment 5

Assigned: October 13, 2014
Due: October 20, 2014 11:59pm

It's game time guys and girls! This week you're going to implement a reduction and scan on GPUs using OpenCL. These two primatives are the building blocks of many algorithms on GPUs and other data parallel processors. Next week, you'll finish up the lab assignments by using your scan kernel to implement radix sort on a GPU. After implementing radix sort on a GPU, you will enter the ranks of GPU programming Jedi Masters.

With the hyperbole out of the way, I'll describe your actual tasks. First, you'll need to implement a parallel reduction (Section 1) and then you'll tackle a scan (Section 2).

1 Reduction

```
int cpu_reduce(float *arr, int n)
{
  int s = 0;
  for(int i = 0; i < n; i++)
    s+=arr[i];
  return s;
}
```

Listing 1: A serial reduction

As in Listing 1, reductions are common. While serial C code provided is straightforward, the data parallel equivalent is more complicated. For details of what a reduction is and how it works, refer to the class slides or the resources included in this write-up. As always, the internet is your friend but make sure you understand everything in your submission as Paden is lazy and will ask definitely ask you a reduction question on the next midterm.

You'll only need to implement a single kernel but some minor trickery is required to perform reductions larger than the work group size. If you recall from lecture, work items within a work group can communicate through a **__local** memory (remember to synchronize work items after updating a local memory!). This enables a work efficient implementation of a parallel tree reduction. However, on the hardware in the Hive lab the maximium work group size is only 512 elements. If we want to reduce vectors longer than 512 elements, we'll have to do a multistage reduction!

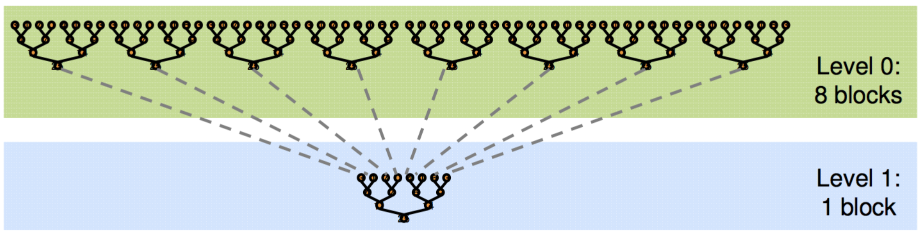

Figure 1: Multistage reduction: Level 0 uses many work groups to generate partial reductions. The Level 1 kernel sums all the partial reductions to generate the final (complete) reduction

Figure 1 (stolen from Nvidia's documentation[1]) hows how a multilevel reduction works. Assuming the array to reduce has length n and each work group does a 512 element reduction, the Level 0 kernel

[1]http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

will emit a vector of length $\frac{n}{512}$ partial products. While there are more than one partial products, we continue to reduce the partial products. For more details of exactly how this works, you might want to look at ideas about reductions GPUs here [2].

1.1 What you EXACTLY need to do

1. Implement a data parallel reduction kernel. We'll assign bonus points for implementations that avoid bank conflicts.
2. Write the appropriate host code to perform a multilevel reduction. This will reuse your reduction kernel. You might want to prototype this first on the CPU before you fully dive into the OpenCL implementation.

1.2 Your deliverables for reduction

Please submit a fully working version (there's testing code included) of a data parallel reduction. We want comments throughout your source. Again, a sentence or two will suffice. No novels in comment form, please :).

2 Scan

```
void cpu_scan(int *in, int *out, int n)
{
  out[0] = in[0];
  for(int i = 1; i < n; i++)
    out[i] = out[i-1]+in[i];
}
```

Listing 2: A serial implementation of scan

In data parallel programming, we use scan (serial CPU version shown in Listing 2) as a building block for bigger and badder algorithms. We want you to implement a scan kernel for this assignment. We've provided all the necessary host API code (no need to modify scan.cpp for correctness). All you need to do is implement the scan kernel in scan.cl.

At the high-level, both scan and reduction are very similar and will use **__local** memory to communicate operands between work-items. However, scan is slightly more complicated due to the more complex data shuffling pattern. The algorithm for scan presented in class (and in the paper **Data Parallel Algorithms**) is one possible way of implementing the core scan kernel. There are other, more efficient scan algorithms for GPUs too. While we have not described them in class, we'll provide bonus points for correct *work-efficient* implementations of scan.

2.1 Your deliverables for scan

Please submit a fully working version (there's testing code included) of a data parallel scan. Please add comments to scan.cpp to describe how our provided multilevel scan operation works.

[2] `http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-simple-reductions/`