GPU Programming with CUDA

Dan Kearney, Noam Rubin, Theo Thompson

Introduction

For this final project, we decided to explore GPU programming. Programming a GPU is an interesting application of skills we learned in software systems. It requires a different way of thinking, and relies heavily on synchronization techniques. We followed along with the Udacity course on CUDA because this was a good beginner's course to GPU programming. It also included enough information to allow us to write some applications which take advantage of a CUDA GPU. In addition to the class lectures, we did some of the homeworks included in order to get real experience utilizing CUDA.

Topics we covered in the course

GPU Hardware Architecture

GPUs and CPUs are optimized for fundamentally different problems. Unlike CPUs, GPUs focus mostly on throughput. Their hardware architecture reflects this priority for parallelization. GPUs are organized into many streaming multiprocessors (SMs) that each contain many simple processors (the number of SMs and processes depend on the GPU). This allows the GPU to spawn many different threads at once, each of which runs on a dedicated processor.

This hardware design is communicated through the API as threads and thread blocks. In GPU programming, threads are truly concurrent and parallel operators. Threads are organized into thread blocks. A given GPU program will usually use a two or three dimensional matrix (known as a grid) of thread blocks, and within each block threads are organized in two three dimensions. This nested dimensional structure gives programmers a lot of flexibility to structure their problems in different ways.

Code Execution on a GPU

Code executed within each GPU thread is known as a kernel. When a GPU runs code, each SM in the GPU runs several kernels simultaneously. To support the use of thread and thread block dimensions to break down problems more easily, each kernel knows the index of the thread executing the kernel and which block the thread is in. In this way, each kernel can operate on a specific portion of input data based on the dimensions of the thread executing the kernel.

GPU Memory Model

There are three types of memory on the GPU. First, each processor has some amount of memory in the form of registers (similar to an L1 cache on a CPU). Next, each block of SMs has shared memory. This shared memory is meant to allow threads to communicate with each other, or modify the same values in memory. This shared memory is somewhat slower than thread memory, but it turns out to be very useful.

The last type of memory is global memory. This type of memory allows data to be shared

between blocks. This is the slowest type of memory, so optimizing CUDA programs typically involved reducing the amount of calls to global memory.

Basic GPU Patterns

Map

Map is a very basic GPU pattern where the same function acts on every element of a set. For example, every element of an array can be mapped to a squaring function to achieve a squared array. By definition, map is a 1-to-1 operation where every input has a corresponding output.

Gather

Gather is similar to map in that a function is transforming some data to achieve some output. The difference is that gather is not 1-to-1. Gather turns some amount of inputs into one output. For example, a running average of an array could be thought of as a gather operation.

Scatter

Scatter is the inverse of gather: rather than combine data from several sources into a single element, the scatter pattern involves copying data to several destinations.

Stencil

Stencil is a little different from map, gather, and scatter. Stencil modifies neighboring elements in an array or matrix based on a specific pattern. A good example of this is image blurring, where each pixel is an average of all its neighboring pixels.

More Advanced Patterns

Apart from map, gather, scatter, and stencil, there are a few advanced patterns that prove to be especially useful when programming with CUDA. Since they are so prevalent, these patterns are called “parallel primitives”.

Reduce

Reduce can take several inputs, a binary operator, and an accumulator, and reduce several elements of input data into a single element of output data. A common use case to reduce is summing all the elements in an array. While iterative code accumulates values one at a time, GPU-optimized reduce would form a reduction tree, progressively adding adjacent elements and reducing the number of input elements until a final result is reached.

Scan

Scan is similar to reduce in that it involves reducing a set of elements into a single element. The big difference with scan is that the binary operation takes as input the result of the previous operation and a new element. All operations begin with an identity element as the initial result, the identity element being decided by the type of operation (e.g., 0 for addition, 1 for multiplication).

A good example of scan is calculating the Fibonacci sequence given an array of consecutive elements as input. A scan performed with addition as on the operator on an array of consecutive elements would produce the Fibonacci sequence.

There are two algorithms most commonly used to perform scan: Hillis-Steele scan and Bleloch scan. To understand the difference between these two algorithms, we define two properties of GPU programs: work and steps. If we imagine a tree of work that a GPU program does, each level of the tree is a step, whereas each individual operation is considered work. Hillis-Steele is a more step-efficient scan, whereas Bleloch is a more work-efficient scan.

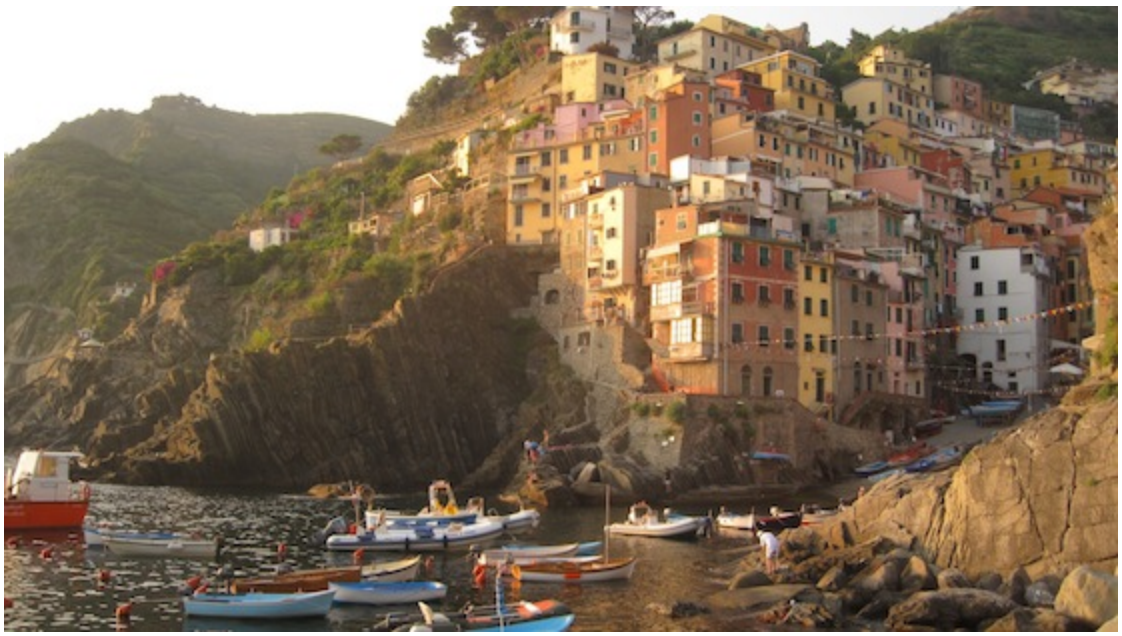
Histogram

Histograms turn out to be extremely prevalent in GPU programming. Oftentime, when processing an image, you want to map from one space to another. For example, mapping an image with millions of colors to a space with a smaller color set involves using a histogram with a discrete number of bins.

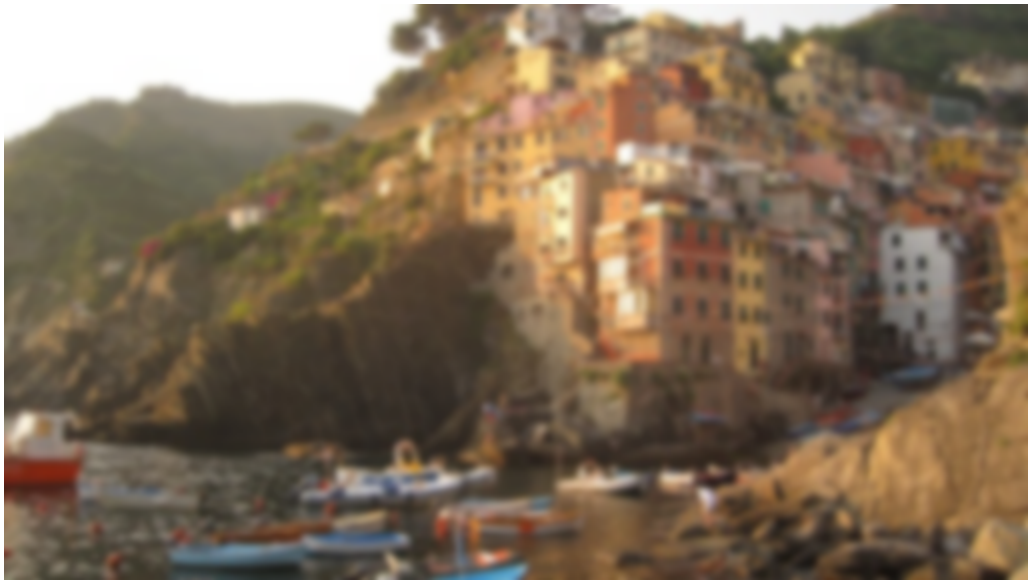
Forming a histogram serially is trivial. The task is only about determining what bin the value should be put into. The parallel implementation of a histogram is not so simple. Since many threads must access the histogram at the same time, this task must be done either atomically or by creating several histograms and reducing them. Since atomic operations are slow in this context, it is best to simply create a histogram for every thread and reduce them into one at the end.

Excercise

In conjunction with this document, you will find code in this repo, we wrote as part of the course exercises, which blurs an image, in this repo (called blur.cu). We also included before and after images to show the effect the code has.



(Original)



(Blurred)