Senior Project: GPU Computing

John Kloosterman

`john.kloosterman@gmail.com`

September 2012-May 2013

1 Introduction

This is the introduction.

Contents

2   Test Computer

Using funds from one of Prof. Adams' NSF grants, I built a system for testing GPU programs. This system included four different devices that OpenCL supports: the CPU, the small Intel GPU integrated into the processor, an AMD Radeon 7970 GPU, and two nVidia GTX 480 GPUs.

The theoretical peak performance of this system is XXX TFLOPS. This is comparable to Dahl, Calvin's supercomputer, which has a theoretical peak performance of XXX TFLOPS.

This system was used for all benchmarks in this report. The CPU used is an Intel Core i7-3770, and the GPU used for benchmarks is the AMD Radeon 7970.

3   OpenCL Framework

OpenCL is designed to be flexible, but this means that it is unwieldly for developers to use. The simplest OpenCL program that runs code on a GPU is on the order of 50 lines long. The framework is an attempt at making OpenCL kernel calls syntactially as similar as possible to calling a C++ function or method.

3.1   Functions vs. Kernels

The entry point of an OpenCL program is a kernel, marked with the `__kernel` keyword. This framework wraps around OpenCL's native functionality with the CLKernel class, making it simpler to compile kernels, pass parameters to them, and set the local and global workgroup size of the kernel. An example usage of the CLKernel class is [FILE NAME].

OpenCL does not support calling non-kernel functions, but these functions need some way to be tested. The textttCLFunction class in this framework removes OpenCL's limitation. A kernel to call the function is automatically generated at compile-time, and that kernel is passed to OpenCL. `CLFunction` will run the function on one thread.

More complex functions, especially those that involve threads cooperating on a task with data stored in local memory, cannot be called by `CLFunction`. One idiom I developed when testing these kinds of functions, is to write a shim kernel that copies data into the correct memory space, calls the function to be tested, then copies the results back to `__global` memory. (See [FILE NAME] for an example)

3.2 Usage

C++11 constructs allow a class to syntactically behave like a variadic function, by defining an overloaded () operator using a variadic template. At this time, compilers only partially support the features needed to make using variadic templates elegant. With C++11, a `CLKernel` or `CLFunction` can be called like this:

```
#include <CLKernel.h>

string src; // some kernel source code
cl_int i, j, k;
CLKernel theKernel( "kernel_name", src );
theKernel( i, j, k );
```

Microsoft Visual Studio 2008 (the version that AMD's OpenCL tools currently target) does not support C++11. This requires a clunkier syntax:

```
#include <CLKernel.h>
#include <vector>

string src; // some kernel source code
cl_int i, j, k;
CLKernel theKernel( "kernel_name", src );

std::vector<CLUnitArgument> arguments;
arguments.push_back( i );
arguments.push_back( j );
arguments.push_back( k );

theKernel( arguments );
```

The CLUnitArgument class has constructors for many different types, which means that variables of those types can be passed into a CLKernel or a CLFunction without needing to explicitly create a CLUnitArgument.

3.3 Other Features

I found I was often developing on my laptop, which does not have an OpenCL-supported GPU. The framework automatically degrades to using a CPU if there are no GPUs, so that programs will still run, albeit much more slowly in most cases.

If the CL_DEBUG environment variable is set to 1, the framework will compile kernels with debugging symbols and run them on the CPU. This allows for debugging kernels using gdb as described in the AMD OpenCL programming guide[1].

4 Raytracer

As a simple application to run on top of my framework, I implemented an OpenCL raytracer for honours credit in CS 352 (Computer Graphics). The ray-

tracer maps one pixel onto one hardware thread. The objective was for the raytracer to support real-time user interaction.

4.1 Capabilities

The raytracer has two geometric primitives: spheres and planes. Geometry can have a solid colour or be reflective. There can be any number of geometric primitives.

The lighting model takes into account ambient and diffuse lighting. There can be any number of diffuse light sources.

4.2 Limitations

Because OpenCL does not support recursion, reflective surfaces do not behave as they do in other raytracers. Reflective surfaces shoot a ray off the reflective surface, and the ray takes the colour of the first object it hits, taking into account only ambient lighting (see Figure 1). Other raytracers are able to take into account other types of lighting from the reflected surface, and can simulate rays being reflected more than once. This is not possible with this implementation, because it would involve a recursive call from the lighting function to the lighting function.
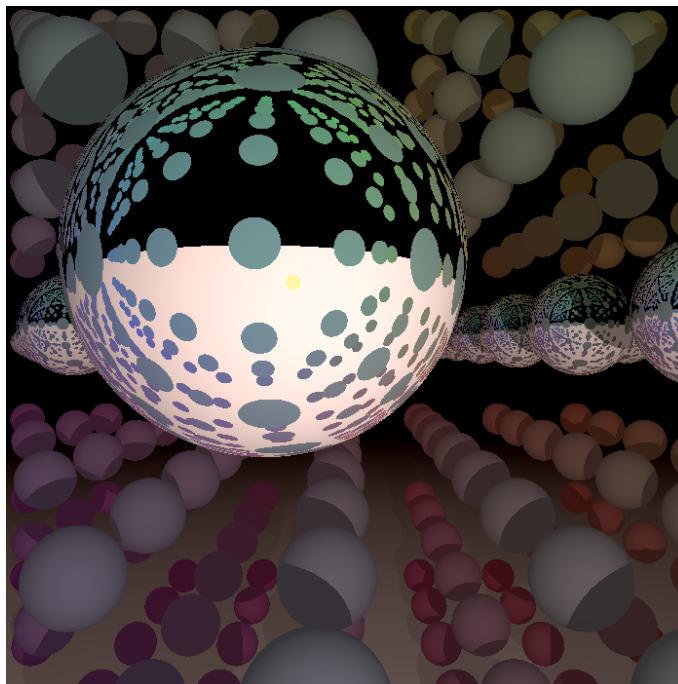

Figure 1: Reflections that only take into account ambient lighting

4

4.3   User Interface

The user interface was implemented in GTK+, with the rendered image being displayed in a `GtkImage`. This is inefficient, as the image is rendered on the GPU, copied to the CPU, copied to the `GtkImage`, then pushed back to the GPU. An alternative that trades increased complexity for more performance would be taking advantage of OpenCL's OpenGL interoperability features to draw the image.

4.4   Performance

The raytracer is able to render a 700x700 pixel test scene with 1000 spheres and a moveable diffuse light source at speeds that make it interactive (see figure 2). Using the CPU, this scene takes 1.28 seconds per frame (0.78 frames per second). Using the Radeon 7970, the scene takes 0.055 seconds per frame (18 frames per second). If the number of spheres is reduced to 216, the Radeon 7970 can render the scene at 60 frames per second.
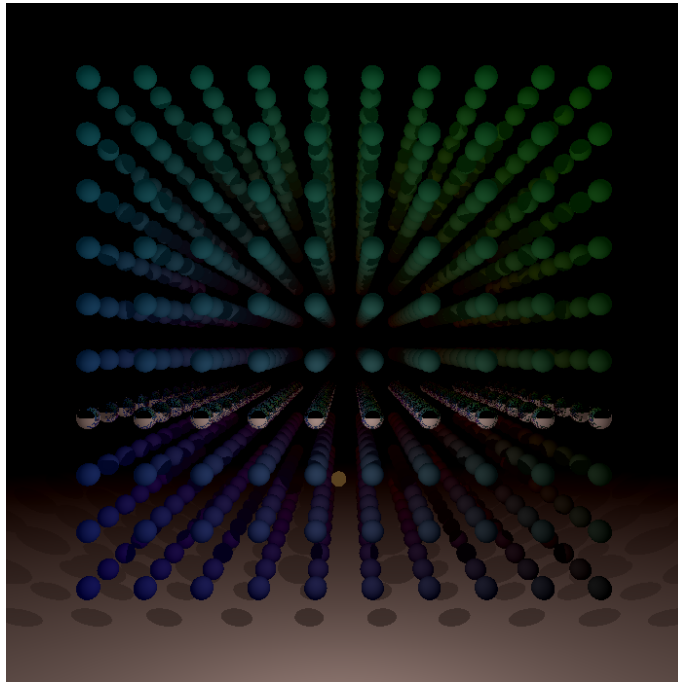

Figure 2: Test scene, featuring 1000 spheres and ground plane

5 Mankalah Minimax AI

This part of the project implemented a minimax player for the Mankalah game introduced in CS 212. Minimax is a much harder algorithm to implement on a GPU than a raytracer, because the minimax tree has dependencies between nodes, and the parallelism is less obvious.

5.1 Strategy

Following previous work on another minimax player implemented in CUDA[3], the minimax tree is broken up into layers. On the CPU, the first layers of game boards in the minimax tree are computed and the bottom-level leaf nodes of that tree are put into a C++ vector. The boards in the vector are copied over to the GPU, where 4 more levels of minimax are computed. Because the Mankalah minimax tree has a branching factor of 6, the overwhelming majority of the work is done in the bottom 4 levels of the tree (see figure 3).
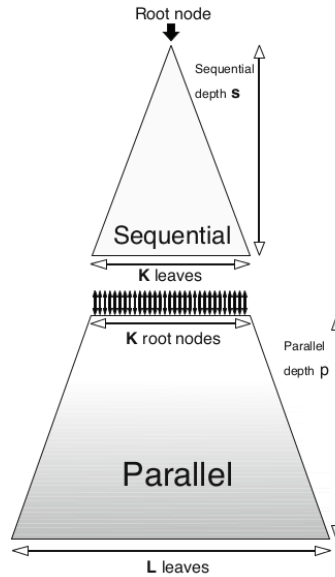

Figure 3: Sequential and parallel segements of minimax tree, taken from Rocki and Suda[3].

5.2 O(log(n),n) Tree Algorithms

5.3 Performance

6 Economics Simulation

As a larger, more complex problem, I worked on implementing a GPU version of an economics simulation used for research in Calvin's economics department.[2] The simulation already exists in Python, but it takes on the order of weeks to run. The simulation consists of a number of agents, which each hold a number of resources, that can harvest resources, invent machines to make harvesting resources more effcient, and trade resources and machines with each other.

This problem has promise for a speedup with a GPU's massive parallelism. Each of the agents in the simulation is largely autonomous, and most of the decisions that agents have to make need to evaluate the relative worth of their resources. Therefore, there is a natural mapping of one hardware thread to one resource, with each agent mapped to its own workgroup.

I did not complete the reimplementation of the simulation, but have met my objective of applying GPU computing to a complex, real-world problem. The Societies paper[2] breaks the simulation into 6 phases, of which the first 3 are non-trivial, the fourth is very complex, and the last 2 are trivial. I implemented the first 2 phases. This was enough to encounter difficult problems that required very different algorithms to efficiently solve on a GPU.

6.1 Phase 1: Resource Extraction

In this phase, agents harvest resources while there is time left in the "day". In each round, agents choose one of the resources that is most valuable for them to have one more unit of. Agents gain experience extracting resources, which reduces the amount of time needed to extract that resource.

The challenges implementing this phase were implementing the maximum and minimum algorithms, creating a mechanism for threads to create a variable-size array of options, and finding a random number generator.

6.1.1 Array Maximum and Minimum

Several times in this simulation, I needed to find the maximum or minimum in an array without modifying the values in the array. The most efficient way to do this with at least 1 thread for every 2 elements is to build a max/min tree, since this allows the maximum or minimum to be found in $O(log(n),n)$ time. Unfortunately, for an array with n elements, this requires a scratch array in `__local` memory of $\frac{n}{2}$ elements.

The algorithm has two phases. In the first phase, $\frac{n}{2}$ threads compare 2 elements of the original array, and put the largest in the scratch array. In the second phase, $\frac{n}{4}$ threads compare 2 elements of the scratch array, and put the maximum/minimum of the two values in the location of the first value. The

second phase iterates, each time with half as many threads, until there is one value left, which is the minimum/maximum.

The implementation of this algorithm is in `societies/util/max_min.cl`. This implementation also includes the ability to pass in a mask array, which allows the algorithm to ignore certain values in the array. This permits using the algorithm multiple times to find the maximum/minimum n elements in an array. (See the `max_n_indices()` function.)

6.1.2 Variable-length Arrays

There are cases where one thread needs to make a choice between different values on behalf of the workgroup. One implementation of this idiom can be found in `societies/util/choose_thread.cl`, where several threads can register their ability to be chosen, then one thread randomly makes a choice between them. Since not all threads want to be chosen, there needs to be a data structure that can hold a variable number of elements and that all threads can add elements too.

This data structure can be implemented in OpenCL with an atomic counter variable in `__local` memory initialized to 0, with an array in `__local` memory that is large enough for the maximum possible number of elements. When a thread wants to add an element, it calls the OpenCL-builtin `atomic_inc()` function to increment the counter, and puts a value in the array at the position that `atomic_inc()` returns, which is the previous value of the counter variable. After all threads have finished adding values to the array, the counter variable holds the number of items in the array.

6.1.3 Random Numbers

Many of the Societies algorithms required a source of random numbers. On a GPU, this is difficult because there is no hardware source of random numbers, and PRNGs require a unique seed per workgroup so that each workgroup does not generate the same sequence of random numbers. I made use of the MWC64X random number generator, which is ideal for GPUs because it requires very little state to be preserved across runs. I found an OpenCL implementation which was verified against statistical tests.[4]

6.2 Phase 2:

6.3 Testing

One weakness of the Python Societies code is that it is not written in a way that makes it easy to test. I made sure to make my code very clean and wrote unit tests for all my functions, so that my code will be useful to the Societies project in the future. The algorithms I developed to find the maximum and minimum elements in an array I was also careful to test; this code should be useful as a reference for others doing similar work in OpenCL.

7 __local Memory malloc()

References

[1] Advanced Micro Devices. AMD Accelerated Parallel Processing Pro-gramming Guide. `http://developer.amd.com/download/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf`, July 2012.

[2] Anthony J. Ditta, Loren Haarsma, and Rebecca Roselius Haney. Societies: A Model of a Complex Technological Evolving Economy. *Journal of Artificial Societies and Social Simulations*. Working paper submitted to journal.

[3] Kamil Rocki and Reji Suda. Parallel Minimax Tree Searching on GPU. In *PPAM 2009, Part I, LNCS 6067*, pages 449–456. Springer-Verlag Berlin Hei-delberg, 2009.

[4] David B. Thomas. The MWC64X Random Number Genera-tor. `http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.html`.