Senior Project: GPU Computing

John Kloosterman
john.kloosterman@gmail.com

September 2012-May 2013

1 Introduction

This is the introduction.

Main problem I was trying to solve: GPUs are an architecture that has potential for great performance and power improvements, but they are too difficult to use.

Make the parts of your project tell a coherent story about how they fit together. One thing you have done a lot of is algorithms to do fairly simple things on the GPU but don't have a standard library implementation. Idioms like linked lists on CPUs -¿ idioms like skipping threads for trees on GPUs.

Contents

2 Test Computer

[introduce test system here]. The system was paid for out of NSF Grant #XXXX, with the nVidia GPUs provided by nVidia.

2.1 Hardware

This system included four different devices that OpenCL supports: an Intel Core i7-3770K CPU, the integrated Intel HD Graphics 4000 GPU, an AMD Radeon 7970 GPU, and two nVidia GTX 480 GPUs.

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. The benchmarks were run on 64-bit Ubuntu 12.10.

2.2 Software Setup

On Linux, it is currently difficult to have multiple OpenCL platforms installed at the same time. GPU platforms will only work if the X.org driver for that

GPU is currently being used, which meant that I could not use the nVidia GPUs when the AMD GPU was being used to drive the display. As well, Ubuntu would not boot with the nVidia graphics drivers installed when the AMD graphics card was installed in the system. On top of this, the nVidia and Intel platforms inmplemented OpenCL 1.1 whereas the AMD platform implemented OpenCL 1.2, and the headers are incompatible between versions even when only OpenCL 1.1 features are being used. For this reason, I developed exclusively with the AMD APP SDK 2.8 on Linux.

The situation is far easier on Windows, where both AMD and nVidia GPUs were always accessible with OpenCL. The Intel integrated GPU was only available if the system was booted with a display connected to it. AMD has the most robust set of tools for profiling and debugging OpenCL, but these tools only work on Windows in Visual Studio. Therefore, my workflow was to develop on Linux, but ensure that the code was portable to Windows, should I need to test on more GPUs and work with the profiling tools available there.

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 `CLFunction` 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 `societies/util/test/max_min_tester.c` 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 partically 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.

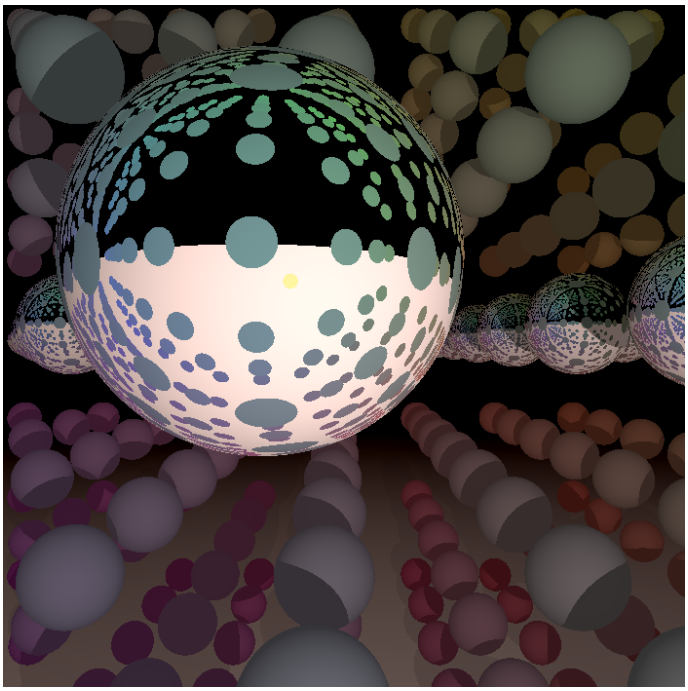

Figure 1: Reflections that only take into account ambient lighting

4.3 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.

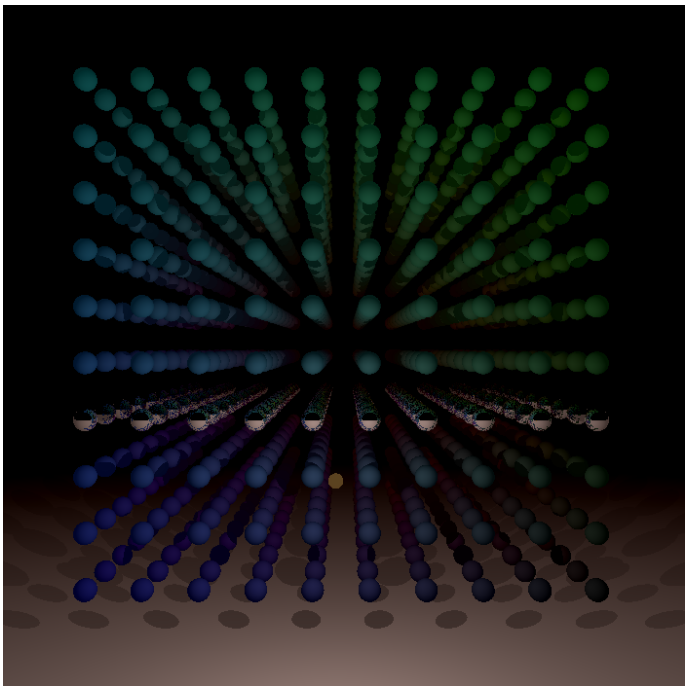

Figure 2: Test scene, featuring 1000 spheres and ground plane

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).

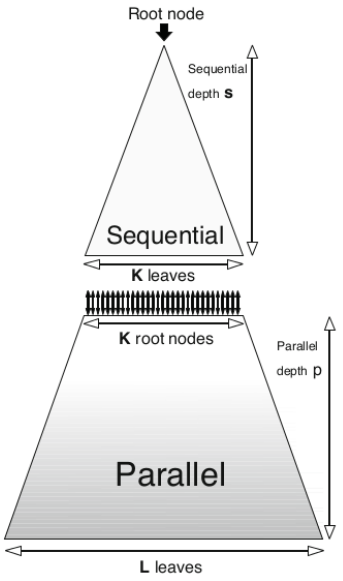

Figure 3: Sequential and parallel segements of minimax tree, taken from Rocki and Suda[3].

The algorithm run on the GPU for evaluating the bottom layers of the minimax tree is as follows:

5.1.1 Generating boards

In this step, the game boards that represent the game tree for the next number of moves are generated from the start board. The tree is stored in global memory in an array, where the child of the node stored at location n in the array is at location 6n + 1. This can be computed in O(log(n)) time with n^d threads, by having the first thread compute the first node, 6 threads compute the first level of child nodes, 36 threads compute the second level, and so on.

5.1.2 Evaluating boards

In the standard minimax algorithm, the evaluate function needs to be computed only for boards at the leaf nodes of the game tree. Because all threads execute the same instruction counter, however, it is not slower to run the function on all the generated boards. This saves time in a later step, as should the game end at a stage before the bottom level of the minimax tree, the board is already evaluated.

5.1.3 Minimax

The minimax scores at a node can be computed in O(log(n),n) time by using n^{d-1} threads to compute the minimax values for the nodes one level above the leaf nodes, n^{d-2} threads to compute the minimax values for the nodes 2 levels about the leaf nodes, and so on. After this process, the parent node for the game tree stores the minimax value for the tree.

5.1.4 Limitations

Because the boards are stored in local memory, and there are sequential dependencies between these steps, all threads for one of these minimax trees must be in the same workgroup. On the Radeon 7970, the maximum work group size is 256. Therefore, for Kalah, with a branching factor of 6, a minimax tree of 4 levels ($6^0+6^1+6^2+6^3 = 259$ nodes) was used, with a workgroup size of $6^3 = 216$. Since there were more nodes in the tree than threads in the workgroup, some threads had to evaluate 2 boards.

5.2 Increasing Search Depth

Because of memory bottlenecks, there is a limit to how many GPU minimax instances can be dispatched efficiently. On the test system, 8 levels of sequential minimax (genering an average of [–] GPU instances) was optimal for performance. As well, as Rocki and Suda[3] noted, the search tree has to be traversed twice at this level, once to determine what the leaf nodes are, and second to run the minimax reduction on the nodes.

Therefore, to gain more search depth, another level of sequential minimax was run above this first round of sequential minimax. Adding more levels at

the topmost level of minimax increased the depth of the search but not the number of GPU instances dispatched at one time. Only one traversal of the search tree is necessary at this step.

5.3 Performance

5.3.1 Optimal Sequential Depth

One optimization question is for a minimax tree of a given depth how many levels shuld be computed in each of the 3 minimax stages. The GPU portion is fixed at 4 levels, so the distribution needs to be between the two sequential minimax stages. On the test system, the optimal allocation was XXXXXXXXXXX.

REDO THIS, because for depth 13, (0,10) is fastest, but (0,11) is too large for the GPU. It would make more sense to do depth 14, even though ti would take longer to benchmark.

Pre	Seq	Parallel	Time for 20	Time for 1
0	10	4	188.03s	9.40s
1	9	4	221.17s	11.05s
2	8	4	212.45s	10.62s
3	7	4	233.92s	11.70s
4	6	4	281.78s	14.09s

5.3.2 Speedup

A recursive sequential algorithm was implemented for performance comparisons. As the table below demonstrates, the using the GPU yielded about a 10x speedup once the problem became sufficiently large. A 10x performance increase for a game with a branching factor of 6 means that the player can look another move ahead in the game.

Depth	Sequential (s)	CPU OpenCL (s)	GPU OpenCL (s)	Speedup
4	0.000068	0.000656	0.000919	0.07x
5	0.000343	0.008957	0.000907	0.37x
6	0.001658	0.004544	0.001054	1.57x
7	0.008301	0.023702	0.001382	6.00x
8	0.040561	0.102351	0.005487	7.39x
9	0.203477	0.516266	0.021644	9.40x
10	0.980497	2.61265	0.1048	9.35x
11	4.80143	11.0873	0.438745	10.9x
12	23.4349	60.0211	2.23596	10.5x
13	113.512	231.792	9.41622	12.1x

Since minimax is an embarrasingly parallel problem, a parallel CPU implementation should be able to achieve a speed of $\frac{sequential}{\#cores}$. Since the OpenCL parallel implementation was worse on the CPU than the sequential implementation, which itself can be sped up 4 times on the test system, this illustrates how algorithms tuned for GPUs are very different than ones tuned for CPUs.

5.4 Testing

Along with testing of each of the GPU minimax algorithm's components, I used fuzz testing to ensure that the output of the GPU minimax algorithm was the same as the output of the sequential CPU minimax implementation. This was done by generating random valid boards then running both algorithms on them and comparing the output.

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 efficent, 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 independent, 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()

One problem I ran across when implementing the Societies code was that most functions required many parameters to pass around scratch `__local` memory that was needed by some algorithm several function calls deep. These scratch buffers needed to be declared at the beginning of a kernel rather then when they were actually used, and I had to keep track of exactly how large they needed to be. Also, because `__local` memory usage restricts the number of hardware threads that can be concurrently running on the GPU, it is important to minimize the amount used, which means reusing the same buffers when possible. The result is that the complexity of managing buffers of `__local` memory became a bottleneck.

Because the sizes of all these buffers is known at compile-time, and OpenCL does not support recursion or function pointers, simple static analysis is enough to determine the maximum amount of `__local` memory a kernel needs. This should make it easy to shift the accounting burden from the programmer onto a tool.

As well, OpenCL does not support global variables, so any state used by a `malloc()`-like tool needs to be passed as a parameter to every function. This is inconvenient for the programmer, and should be easy for a tool to write into a program.

7.1 Integration with Clang

The Clang C-family compiler is written to be extensible. Clang's abstract syntax tree (AST) includes references to the text that every AST node is built from, which is meant to make it easy for tools to analyze and rewrite source code. Clang's `Rewriter` class allows for sections of source code to be added, deleted, and moved around. Clang also is used in many OpenCL vendors' backends, which means it natively supports the OpenCL keywords like `__kernel` and adds these annotations to its AST.

Therefore, this tool links against Clang, and creates an instance of an OpenCL compiler that does semantic analysis but stops before code generation. By using Clang's AST nodes and its `Rewriter` class, this tool offloads the heavy lifting of parsing and rewriting C source code to Clang.

Because of the tight integration with Clang, the tool is vulnerable to modifications to Clang's internals. The tool was developed against Clang 3.2 and LLVM 3.2.

7.2 Phase 1: Computing Maximum Allocation

The tool registers hooks into Clang's AST processing methods. Whenever a function declaration is encountered, the tool generates a new node in a call graph. Then, whenever a function call or a call to `local_malloc()` or `local_free()` is encountered, the tool adds these actions to the node in the call graph. Clang's AST nodes for the `local_malloc()` and `local_free()` have methods that make it easy to get a numerical value for the amount of memory allocated or freed, even when that value is an expression or defined by a preprocessor macro.

After all nodes in the AST have been visited, this call graph is guaranteed to have no cycles, because OpenCL does not support recursion. Therefore, in this call tree, the maximum amount of memory allocated by a function is the maximum amount allocated by any of a function's children plus any memory the function has allocated at the same time.

7.3 Phase 2: Program Rewriting

The tool then revisits all the nodes in the AST again, and rewrites function declarations and calls. To the kernel function, it inserts initialization code and allocates the `__local` memory buffer at the size computed in the previous phase. The tool adds a state object to every function call and declaration, so that that state object is accessible anywhere in the program.

8 Conclusions

References

[1] Advanced Micro Devices. AMD Accelerated Parallel Processing Programming 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 Heidelberg, 2009.

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