local_malloc(): malloc() for OpenCL __local memory

John Kloosterman
Calvin College
3201 Burton SE
Grand Rapids, MI 49546
john.kloosterman@gmail.com

Joel Adams (advisor)
Calvin College
3201 Burton SE
Grand Rapids, MI 49546
adams@calvin.edu

ABSTRACT

One of the complexities of writing kernels in OpenCL is managing the scarce per-workgroup `__local` memory on a device. For instance, temporary blocks of `__local` memory are necessary to implement algorithms like non-destructive parallel reduction. However, all `__local` memory must be allocated at the beginning of a kernel, and programmers are responsible for tracking which buffers can be reused in a kernel. We propose and implement an extension to OpenCL that provides a malloc()-like interface for allocating workgroup memory. This extension was implemented by using an extension to the Clang compiler[1] to perform a source-to-source transformation on OpenCL C programs.

1. MOTIVATION

Making effective use of `__local` memory space in OpenCL programs is essential for performance, since `__local` memory is, unlike the other OpenCL memory spaces, both low-latency and shared between threads in a workgroup[3, pp. 27]. Temporary buffers of `__local` memory are needed to efficiently implement basic parallel algorithms such as reduction. Because the amount of `__local` memory needed is an implementation detail of an algorithm, a memory allocation API makes reuse of modules for a common task like parallel reduction easier. With this interface, the code itself specifies how much memory is needed and at which times. The module can be inserted into a new kernel and the API will adjust the needed amount of of memory automatically.

As well, `__local` memory is a scare resource, as even top-of-the-line GPUs have only 64 kilobytes of `__local` memory per workgroup[2, pp. 105], and the amount of `__local` memory used is a bottleneck in the number of threads that can be concurrently executing on the GPU. In complex OpenCL kernels, buffers of `__local` memory can be reused by different algorithms, but managing the resulting complexity is the responsibility of the programmer, with no help from the OpenCL C programming language.

2. API

Our memory allocation API is intentionally similar to the familiar POSIX `malloc()` dynamic memory system. The `local_malloc()` function allocates an amount of `__local` memory, which must be later freed by a call to `local_free()`. However, because memory allocation is not possible in OpenCL once a kernel begins running on a device, we simulate dynamic memory for the programmer by computing at compile time the maximum amount of memory a kernel will allocate and transparently inserting an allocation of a buffer of the required size into the source code of an input program.

To perform the source code transformations, we implemented a library in C++ that is called by OpenCL host code. OpenCL C source code is passed into the library, which performs transformations and returns rewritten source code. The host program can pass the rewritten source code to vendors' existing OpenCL implementations to run on a device.

3. PROGRAM ANALYSIS

Because all OpenCL `__local` memory buffers must be allocated at the beginning of the kernel, analysis of the maximum memory allocation is done at compile time. This is accomplished by building a call graph of the OpenCL kernel and defining the maximum allocation of a function as the maximum amount of memory a function and its children have allocated at any one time. Because OpenCL C prohibits recursion, the call graph cannot have cycles, making it a call tree.

The Clang C frontend exposes an API for visiting abstract syntax tree (AST) nodes as source code is parsed. Hooking into this API, our tool listens for function call AST nodes. If the call is to `local_malloc()` or `local_free()`, a record of the amount of memory allocated and deallocated is inserted into the call tree. When calls are made to other functions, edges are inserted into the call tree. After this process is completed, the call tree is traversed to find the maximum allocation for the entire kernel.

4. SOURCE REWRITING

OpenCL C does not have global variables, so an object storing the state of the memory allocation system must be passed as a parameter to every function that calls `local_malloc()`. Instead of putting the burden on the programmer to include this parameter in every function call, we use Clang to automatically rewrite function calls and declarations. Calls to OpenCL built-in functions are detected so that calls to them

are not modified.

Clang exposes its Rewriter interface to allow source code to be efficiently edited. Because Clang does not store the AST in memory, the input source code must be processed a second time for rewriting. In this pass, function declarations and calls are rewritten to include a pointer to the state required by the `local_malloc()` system. The buffer of `__local` memory is allocated at the beginning of the kernel, and the source code for the `local_malloc()` and `local_free()` functions is prepended to the input source code. The resulting source code is standard OpenCL C and can be compiled by device vendors' pre-existing implementations of OpenCL.

5. CONCLUSIONS

By implementing `local_malloc()` with a source-to-source transformation using Clang, developers can allocate `__local` memory using the malloc() interface they are already familiar with while at the same time maintaining compatibility with vendors' pre-existing OpenCL C toolchains. This memory allocation system frees programmers from manually finding opportunities for memory re-use, and makes modules more reusable as the code itself describes its memory needs.

6. REFERENCES

[1] clang: a C language family frontend for LLVM. `http://clang.llvm.org`.

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

[3] Kronos Group. OpenCL 1.2 Specification. `http://www.khronos.org/registry/cl`, November 2011.