David Fiala
Nov. 14, 2010
CSC 766 Class Project
Report #2

My Project

Build PTX backend for pseudo language from HW2 and implement a new construct for loop parallelism (PARALLELFOR construct)

Overview of Work to Date

Early on I identified a few key pieces that would be the foundation of this project and were a critical stepping stone to creating PTX code and debugging it. Namely, I needed to design a host application that would load, run, monitor, and support the GPU PTX code that is generated from my PSEUDO-to-PTX compiler. Additionally, it would be necessary early on for me to be able to debug my PTX code so I also decided that implementing READ/WRITE functionality was an imperative task.

At this point I have designed and implemented a generic host application that will run my compiled PTX code. Furthermore, a specialized circular buffer has been designed to allow communication between the host and the GPU application. I will describe the techniques used in the following section.

Solved Issues

Host Application

Unlike traditional CUDA programs, the host application for this project required the ability to dynamically load a GPU kernel and perform asynchronous streaming operations. The regular CUDA Programming API does not allow dynamic loading of PTX code or compiled CUBIN code so it was necessary to drop to a lower level and use the CUDA Driver API (cu). Using the driver API I am able to dynamically load PTX from an external file or directly embed the PTX code in to the produced binary thereby allowing my PSUEDO compiler produce (generate code, compile) a single executable with all functionality built in. Additionally this allows me to lose my dependency on *nvcc* and compile with only the NVIDIA libraries and cuda include files. Gcc can be used for the entire compilation process.

IO Communication between Host and GPU

The pseudo language itself provides two commands to read a number from STDIN and then write a number back to STDOUT. Since the GPU itself has no way to access the host's IO functionality, it was necessary to implement robust communication functionality in both the host and the generated PTX code itself. Neither the GPU nor CPU have direct access to each other's memory, but the CPU can copy to and from the GPU's global memory via the driver API. This provided me the option to store two buffers (one for input, one for output) on the GPU's global memory that could then be updated by either the GPU or the CPU. Special care had to be taken to ensure that both the CPU and GPU code cooperated with each other to ensure that no synchronization problems are introduced.

For each communication buffer I used a circular array to mitigate many synchronization problems. Since there is no way to ensure any synchronization between the host and GPU, I modify the circular buffer only in ways that would prevent a synchronization problem by only allowing the CPU and GPU to write to exclusive data elements. However, overlapping reads will pose no problems due to ordering.

David Fiala
Nov. 14, 2010
CSC 766 Class Project
Report #2

Allowing Parallel IO with a Running Kernel

Since each circular buffer has a fixed maximum size, it is necessary to allow both the host and GPU to write and extract data from the buffers while the GPU kernel is still running. To allow for this, I had to parallelize the host CPU application (OpenMP was used) to run a separate thread that reads data from STDIN and communicated it back to the GPU thread so that it can be pushed on to the GPU's input circular array. Likewise, the CPU host code has to constantly check the GPU's output array for any new data and extract it (print it to stdout on host). A final consideration was to deal with instances where either the input or output buffer get full. Eg: The host provides data to the GPU faster than it can process it, or the GPU provides data faster than the CPU can extract it.

This part of the project proved especially time consuming to implement as there needed to be heavy cooperation between the host (code written in C, but depends off of the ability to read data from the GPU in a asynchronous fashion) and the PTX code that also reads and writes the global buffers. To allow the host to copy data while the kernel itself is still running, an additional GPU stream had to be created and all IO operations where asynchronously run from this stream in order to not block waiting for the kernel to finish. (The kernel may not finish if it is waiting on new IO! Quite a chicken-egg problem.)

Despite the complications of the IO, I have implemented robust CPU and corresponding PTX code that can perform IO flawlessly. Additionally, I have written special test cases to ensure that filling a host buffer or a GPU buffer does not break the applications. The host side will continue to try to add a outstanding data to the buffer if it is full, and on the GPU side the write function will also continue to try to put data in the buffer until the host has freed up a spot.

The PTX read/write code is written as PTX functions (They themselves are simply included in the generated code and are callable in PTX assembly.)

Pseudo Language Modifications

The pseudo language has been modified to add the PARALLELFOR construct as outlined.
An additional modification has been made to allow variables to be declared prior to use. This allows the compiler to allocate memory needed for variables without the need of two passes or dynamic generation.

Open Problems

READ/WRITE code can not be executed from within a PARALLELFOR block. There are a couple workarounds, but I do not see the advantage:

1. One workaround is to have all threads return the same value when a READ is called. Unfortunately there is no equivalent for the WRITE command.
2. Require that each READ/WRITE be run as a critical section. Unfortunately this type of problem has received much discussion on NVIDIA forums and is generally frowned down upon; the answer is to seemingly never try to perform these types of critical sections on a GPU. (Bad outcomes/deadlock can occur)

My recommendation is to simply perform all READs and WRITEs in the sequential areas.

Another problem detected is that a GPU kernel has a timeout of several seconds. (Perhaps 10?) After this time the kernel will forcibly exit. This should only be a problem with programs that are human interactive. Otherwise the code should finish before the timeout occurs!

The memory problem: To allow more than 512 threads, global memory is being utilized to mitigate any synchronization/shared-memory-copy problems. For each thread to have some "private" memory that gets copied in to faster shared memory during a parallel for loop I'll need to somehow calculate how much memory is available, and if we can actually permit all threads to allocate enough shared memory to honor the user's request.

Remaining Milestones

November 19th: Finish implementing all basic PSUEDO-language commands in PTX.
November 26th: Fully implemented PARALLELFOR (eg: handles shared/private data flawlessly)
December 3rd: (date?) Present final project.