Towards a D3Q27 Lattice Boltzmann Method Implementation Using CUDA

Erik Brobyn
City College of New York
Department of Physics
First Draft May 5, 2020

Abstract

In this report, the Lattice Boltzmann method is investigated for use with CUDA devices in a Windows 10 environment. A sample application is created, and techniques are explored using NVIDIA's CUDA Samples v10.2 as a guide to best-practices. An existing implementation of a D2Q9 Lattice Boltzmann is analyzed as a potential starting point for further investigation and future modification into a D3Q27 version of the model.

Introduction

The Lattice Boltzmann methods that are used for computational fluid models are popular and well-suited for numerous situations where atmospheric modeling and analysis is warranted. <<history and more about Lattice Boltzmann…>>

Software Investigated

- Microsoft Visual Studio Community 2017 and 2019
- NVIDIA GPU Computing Toolkit (CUDA) v10.2.89 441.22 for Windows 10
- NVIDIA Corporation's CUDA Samples v10.2
- LATTICE BOLTZMANN SIMULATOR GPU accelerated with CUDA
  by Tom Scherlis and Henry Friedlander (2017)
- NVIDIA Nsight for Visual Studio Community 2019

About NVIDIA CUDA Samples v10.2

The CUDA Samples is a collection of 176 projects spread across several subject areas, including Graphics, Finance, Simulation, and Imaging. The sample code is mostly written in C, with some C++ used, and many of the samples are fairly short and digestible. There is often brief commented discussion of best-practices for structure and style.

Given its instructional nature, the CUDA Samples are a pretty good standard for GPU-related development best-practices for this project. There is also a crash course in OpenGL API functionality implicit in the graphical components of many CUDA samples. Generally speaking, the CUDA graphics sample projects have two pieces, a CPU layer, and a GPU or kernel layer.

The CPU layer handles the windowing functions, like OpenGL Utility Toolkit (GLUT) calls, UI handlers, and other display-related functions such as buffer allocation and initialization and other OpenGL calls. This layer is typically implemented in a standard C++ `.cpp` implementation file.

On the other hand, the GPU or kernel layer, looks somewhat different. Functions in this layer generally have declaration specifiers, `__device__` or `__global__` prefixes, indicating whether or not the function is a kernel function (`__global__`) or meant to be run on a single GPU core (`__device__`). Calls to kernels require the `'<<<'` and `'>>>'` notation; within these brackets blocks and thread sizes must be provided. These kernel layers are typically defined in CUDA-specific `.cu` implementation files.

As a preliminary for this project, all the CUDA v10.2 samples were compiled and built using both 2017 and 2019 versions of Visual Studio Community Edition. 173 projects were built without errors, and there were 6 errors spread across 3 projects. Two of these projects related to missing Vulkan libraries, the other error was a heap exception.

CUDA Errors Reported by Intellisense

CUDA projects and files are compiled using NVIDIA's compiler, `nvcc.exe`. Since it is not Visual Studio's standard compiler (`cl.exe`), the Visual Studio IDE's built-in code-highlighting tool Intellisense is not configured to gracefully handle the nuances of CUDA syntax.

Both the 2017 and 2019 versions of Intellisense found undeclared identifier errors in almost every CUDA Sample project. These errors appear for many CUDA-specific objects (`threadIdx`, `blockIdx`, some CUDA functions, `'<<<'` and `'>>>'` operators, and some variables and other references. Short of turning off Intellisense altogether, the best you can do to fix these errors is to include the following header files in any file that uses CUDA syntax or objects:

```
#include <cuda.h>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
```

However, this does not fix all the errors Intellisense finds. The `'<<<'`, `'>>>'` operators and some other symbols continue to be a nuisance, but again, these errors ultimately have no effect on the build.

About lbm_cuda

lbm_cuda is a Visual Studio 2019 solution created with a default CUDA 10.2 project. All projects in the solution are set to `Debug` configuration mode. Most of the conventions used by CUDA Samples for v10.2 will be used for these projects. They include:

- additional include directories `./common/inc`
- lib files location `./common/lib`
- output directories for executables `./bin/win64/Debug`
- to avoid run-time errors, two dlls are required in `./bin/win64/Debug`
  - `glew64.dll`
  - `freeglut.dll`

The fluidsGL CUDA Samples v10.2 project was added to the project space to provide a comparable example of an OpenGL and CUDA-based fluid simulation. It uses the CUFFT library and was only slightly edited for clarification of some concepts. A GLUT-based window management implementation (in a `.cpp` file) makes calls to the GPU (also called kernel) layer via `extern "C"` function prototypes declared in the `.cpp` file. The `extern "C"` functions with CUDA-specific syntax is then defined in a CUDA-specific `.cu` implementation file. This separation marks a clean logical separation between the CPU and kernel layers.

The lbm project was created from another default CUDA v10.2 project. A CUDA source file lbm.cu was created and was initially taken whole from the Lattice Boltzmann Simulator by Tom Scherlis and Henry Friedlander (2017).

Settings for the lbm project were then configured to match conventions in the CUDA Samples v10.2 projects. The library `glew64.lib` was added to the Additional Dependencies (in Solution Explorer, right-click Project, select Properties and add it to Configuration Properties>Linker>Input>Additional Dependencies).

Scherlis and Friedlander's D2Q9 Simulator

The original, unmodified version of Sherlis and Friedlander's code was written as a single `.cu` implementation file 1167 lines long. There are several global variables and definitions shared between the CPU and GPU layers.

The core of the lbm project is a 2D texture mapping via an OpenGL Pixel Buffer Object.

For this project, this code was first refactored into multiple files with a kernel-based file structure. In addition to providing some improved organization, these steps were also helpful in getting a better understanding of the underlying structure of the program. Some of these changes include:

- `Defines` and `structs` shared between the CPU and GPU were split into a separate header file.
- Global variables were moved into a new `.cpp` file and references to those variables marked extern in the `.cu` file.
- All CPU-related functions were then moved to the `.cpp` file.

- GPU entry points were declared as extern function prototypes in the `.cpp` file, and then defined in the `.cu` file.
- Two new GPU-related functions were created for handling initialization and cleanup in the new, more modular file structure.
- Original comments were edited and the code reformatted.

There are two longer term goals for this project. The first is to create an additional dimension visible in through the viewport. To this end, various samples were tested to find a useful mouse-controlled 3D viewport component for OpenGL/GLUT, and this functionality gradually added to the lbm project.

The second goal of this project is to enhance the Scherlis and Friedlander code to include an additional third dimension to the lattice, along with new velocities to the simulation. The objective is a ultimately a D3Q27 model visualization in three dimensions.

Results

Conclusions

This section contains the most important of the results, which ought to be interpreted and explained. Any statement you make here has to be well thought in advance and absolutely justified.

List of References

All the references cited in the report should be listed here. Reference, which is not cited in the report, should not be listed.

Error Analysis

This section includes a detailed analysis of all error sources for experimental measurements. Include both size of measurement errors and method of estimate. Show how errors propagate to final calculated results. Discuss both accuracy and precision of all measurements. Describe any sources of systematic errors.

Appendix A

This section includes tabulation of all raw data

Appendix B

This section includes sample calculations.

Appendix C

The code-base is publicly available for review, downloading, or forking at the following address:

`https://github.com/charlielobster/lbm_cuda`