Learning CUDA: Lab Exercises and Experiences

Nate Anderson, Jens Mache, William Watson

Lewis & Clark College, Portland, OR

Introduction

Since 2005/2006, processor manufacturers have shifted their method of scaling performance from increasing clock speeds to increasing the number of cores. Processors in the future are predicted to have hundreds of cores.

The rise of multi- and many-core processing has introduced new urgency to learning parallel programming. Whereas typical machines today have two cores, today's graphics cards can already run hundreds of threads in parallel. General-purpose computing on graphics processing units (GPGPU) broke into the mainstream with the introduction of Nvidia's CUDA API in 2007.

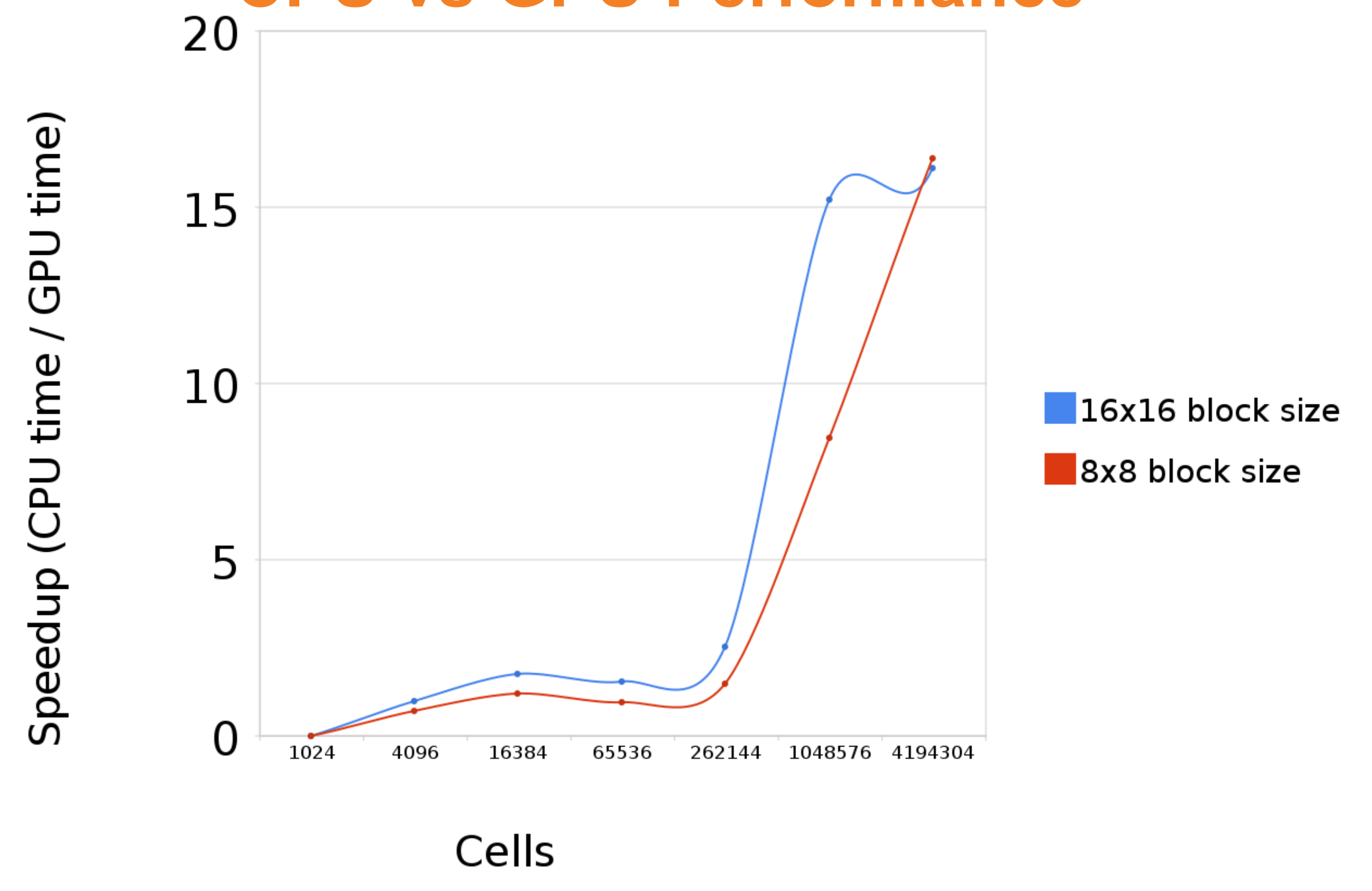
We focus on lab exercises at the undergraduate level. Three undergraduate students and one faculty member spent several weeks on CUDA lab exercises, starting with the book by Kirk and Hwu [1] which was published in February 2010. We describe our experiences and lessons learned working with the book and its accompanying labs. After this, we share our experiences and suggestions and discuss possible extended labs.

Background

The CUDA programming model is an extension of the C language. Programmers write an application with two portions of code — functions to be executed on the CPU host and functions to be executed on the GPU device. The entry functions of the device code are tagged with a __global__ keyword, and are referred to as kernels. A kernel executes in parallel across a set of parallel threads in a Single Instruction Multiple Thread (SIMT) model. Since the host and device codes execute in two different memory spaces, the host code must include special calls for host-to-device and device-to-host data transfers.

When the device executes a kernel it runs within a grid with parameters defined in terms of number of blocks (up to two dimensions) and number of threads per block (up to three dimensions), but totalling no more than 512 threads per block. Streaming multiprocessors on the video card execute the thread blocks. The graphics card we used was the Nvidia 9800GT, which has 14 streaming multiprocessors, each of which can execute 8 blocks or 768 threads at one time for a maximum of 10,762 concurrent threads.

CPU vs GPU Performance



This chart details the speed increase of the lab 3 GPU matrix multiplier over its reference CPU implementation for matrices of different sizes. The different lines represent the different block sizes into which the matrices were divided.

We wanted to see how much faster the CUDA version of matrix multiplication from lab 3 was than the CPU version, so we wrapped a timer around each function. We think that students might find a performance comparison like this more interesting than a "Test PASSED" message.

Lab Exercises

Lab 1: Matrix Multiplication

The first lab involves implementing basic matrix multiplication on the GPU. The supplied matrices are only 16*16, so they can fit in a single block of threads. It provides functions that handle the creation of matrices and host-to-device as well as device-to-host memory management. It does not provide the kernel code or the code to invoke the kernel. Completing the kernel code was a relatively easy process, given a proper understanding of the Matrix structure and how to access elements within the matrix in terms of a one-dimensional array.

We feel that, even though it is meant as an introduction, the supplied code does too much work for the student. Memory management is important and, in a simple lab like this, not hard to understand. We believe that students might benefit from familiarizing themselves with this process from the beginning. This lab took approximately two hours of reading, half an hour to write the code and about two hours to debug the program in order to get it working.

Lab 2: Reduction

The second lab from the book is the implementation of a parallel reduction algorithm. The kernel is given an array of 512 values and returns their sum. The concept introduced here is thread diversion. A naive implementation would cause threads in every warp to diverge, decreasing performance substantially. The following testimonial is also from Student 2.

We feel that it is easy to understand how divergent threads could be minimized and the lab itself is easy to code. Because it is limited to 512 elements, only one block is needed. The concepts behind the lab were easy to understand. It took about an hour to code, although we had more experience with CUDA than a student assigned this lab would.

Lab 3: Mapping and shared memory

Lab 3 involves matrix multiplication like the first lab but extends it by removing the size limit for input matrices. This necessitates using more than one thread block because a single block can hold up to 512 threads, or enough for a 32*16 matrix. The input matrices must be broken up into manageable blocks for the GPU to execute. This is the first lab which uses both the thread index and block index It also introduces shared memory, memory that all threads in a block can read, which can increase the efficiency of the program.

This lab is a more effective tool for understanding the structure of CUDA. Forcing the student to use multiple blocks makes the kernel invocation syntax more clear. We felt that shared vs. global memory should be explained to students and could even be taught separately by having students create tiled matrix multiplication programs both with and without shared memory.

Lab 4: Convolution

Convolution is important in image and signal processing applications. The largest hurdle when programming a convolution kernel is dealing with the edge cases. In our 5x5 convolution lab, each element uses the 24 elements around it, but if it is on or near an edge, it must substitute zeroes for the missing elements. The solution is to load the elements operated on by the block into shared memory but with a halo of required elements around it.

This lab is more challenging than those preceding it. The solution code dedicated threads to loading the halo elements, leaving them idle for the computation itself. Our more efficient solution was much more difficult to code and debug. This lab took about an hour to code and 3 hours to debug, along with an hour of reading. The debugging time would have been cut down if the errors in the program had not crashed the computer when the it ran.

Experiences and Suggestions

Due to their similar nature and their effectiveness as introductory examples into CUDA, labs 1 and 3 are best paired together. The continuity of moving between the two may help solidify the student's understanding of the basic operations of memory allocation and kernel execution.

However, students may be more engaged with a more **results-driven** problem. The current reward for completing a lab is the command line stating "Test PASSED" rather than "Test FAILED." We feel that the benefit of CUDA is lost in the lack of feedback. One method we found interesting was the inclusion of timers to compare the CPU and GPU versions of the solution (see chart). In **lab 3**, we were able to measure a speedup of a factor of 16 for large matrices over the included CPU based matrix multiplier. We believe that displaying the speed increase is effective for motivating students.

Visual feedback for correct solutions is another means to motivate students. Getting a "Test PASSED" message upon completion is not as satisfying as seeing something that the student creates. That is why we propose supplementing lab 4 (convolution) with a lab on the game of life (see [5]). Instead of looking at a 5 by 5 block of elements around the one being worked on, the game of life looks at a 3 by 3. It has similar challenges and would still need haloing to work efficiently, but provides visual feedback.

One additional idea for a lab could be the inclusion of a **simple ray tracer** for those students familiar with graphical concepts. The nature of ray traced 3D graphics allows for pixels to be calculated independently of each other. This project may make for a good final or semester lab as it provides more opportunities for parallel optimization as well as providing a graphical result as a reward for a successful implementation.

Acknowledgements & More Information

We would like to thank Ben Perkins for his participation in this study.

To contact the researchers, email: noa@lclark.edu, jmache@lclark.edu, wwatson@lclark.edu

This research will be presented SPLASH 2010.

References

[1] David Kirk and Wen-mei Hwu, "Programming Massively Parallel Processors: A Hands-on Approach", Morgan Kaufmann, 2010, http://www.elsevierdirect.com/morgan_kaufmann/kirk/