CUDA FOR ENGINEERS

An Introduction to High-Performance Parallel Computing

DUANE STORTI
METE YURTOGLU

PRAISE FOR *CUDA FOR ENGINEERS*

“First there was FORTRAN, circa 1960, which enabled us to program mainframes. Then there was BASIC, circa 1980, which enabled us to program the first microcomputers. And now there is CUDA, which enables us to program super-microcomputers.

“*CUDA for Engineers* allows researchers in engineering and mathematics to perform calculations hundreds of times faster than was previously possible on microcomputers. This permits new kinds of calculations to be performed and reveals this book to be a game changer.”

—Richard H. Rand, professor of mechanical and aerospace engineering,
and of mathematics, Cornell University

“*CUDA for Engineer*s has been put together in a very thoughtful and practical way. The reader is quickly immersed in the world of parallel programming with CUDA and results are seen right away. This book is a great introduction and helps readers from many different scientific and engineering disciplines become exposed to the benefits of GPU programming. This book is an enjoyable read and has great support through top-notch example programs and exercises.”

—Dr. Mark Staveley, senior program manager, Azure High Performance Computing

“*CUDA for Engineers* lives up to its name by stepping the reader through concepts, strategies, terminology, and examples, which work together to form an educational framework so that experts and non-experts alike can approach high-performance computing with foresight and understanding.”

—Joseph M. Iaquinto, Ph.D., research specialist, VA Puget Sound

“This book reflects a practical approach that is in perfect agreement with the way I teach numerical methods for engineers. It would make a fine supplement for engineering students or practitioners to add CUDA to their numerical toolbox, and thus embark on the study of high-performance scientific computing. It’s perfect for newcomers to CUDA who already have a foundation in programming. I recommend following the authors’ advice and working immediately with the hands-on exercises, step by step. After this immersion, you will approach proficiency by simply adding some personal projects in GPU computing and delving into the NVIDIA CUDA Guide and developer community.”

—Lorena A. Barba, associate professor of mechanical and aerospace engineering, The George Washington University

CUDA for Engineers

This page intentionally left blank

CUDA for Engineers

An Introduction to High-Performance Parallel Computing

Duane Storti
Mete Yurtoglu

Addison-Wesley

New York • Boston • Indianapolis • San Francisco
Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Names: Storti, Duane, author. | Yurtoglu, Mete.
Title: CUDA for engineers : an introduction to high-performance parallel
computing / Duane Storti, Mete Yurtoglu.
Description: New York : Addison-Wesley, 2015. | Includes bibliographical
references and index.
Identifiers: LCCN 2015034266 | ISBN 9780134177410 (pbk. : alk. paper)
Subjects: LCSH: Parallel computers. | CUDA (Computer architecture)
Classification: LCC QA76.58 .S76 2015 | DDC 004/.35—dc23
LC record available at http://lccn.loc.gov/2015034266

Copyright © 2016 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, 200 Old Tappan Road, Old Tappan, New Jersey 07675, or you may fax your request to (201) 236-3290.

NVIDIA, the NVIDIA logo, CUDA, GeForce GeForce GTX, Jetson, Kepler, NVIDIA Maxwell, Nsight, Optimus, Pascal, Quadro, and Tesla are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and/or other countries.

Microsoft, Visual Studio, and Windows are trademarks of the Microsoft group of companies.

Apple, the Apple logo, Mac, OpenCL, and OS X are trademarks of Apple Inc., registered in the U.S. and other countries.

Intel and Intel Core are trademarks of Intel Corporation in the U.S. and other countries.

ArrayFire and the ArrayFire logo are trademarks of ArrayFire LLC.

UNIX is a registered trademark of The Open Group.

Wikipedia is a registered trademark of the Wikimedia Foundation, Inc.

IBM, Blue Gene, and PowerPC are trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

ISBN-13: 978-0-13-417741-0
ISBN-10: 0-13-417741-X
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, November 2015

Editor-in-Chief
Mark L. Taub

Executive Editor
Laura Lewin

Development Editor
Songlin Qiu

Managing Editor
John Fuller

Project Editor
Elizabeth Ryan

Copy Editor
Teresa Barensfeld

Indexer
Jack Lewis

Proofreader
Anna Popick

Technical Reviewers
Tom Bradley
Richard Rand
Mark Staveley

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
The CIP Group

To the family, friends, and teachers
who inspire us to keep learning cool things
and to share what we have learned.

Contents

This page intentionally left blank

Acknowledgments

The authors wish to express their thanks to a variety of people without whom this book would never have come into existence.

Thank you to all the family members who received a bit less attention while we were consumed by the writing of this book. Thank you to Laura Lewin and everyone at Pearson who contributed to the editing, production, and marketing efforts. Thank you to Nicholas Wilt (formerly at NVIDIA and currently at Amazon), who first put us in contact with Laura and really got the ball rolling. Thanks also to our technical reviewers Thomas Bradley of NVIDIA, Mark Staveley of Microsoft, and Richard Rand of Cornell University, all of whom provided helpful comments, corrections, and insights.

Thank you to the many colleagues here at the University of Washington–Seattle Department of Mechanical Engineering who contributed via discussions ranging from big-picture perspective down to the finest technical details. That list includes but is not limited to Mark Ganter, Di Zhang, and Ben Weiss (who helped create several of the figures and also provided us with some lifesaving software to support logical tags and automated code formatting). We would also like to thank Mechanical Engineering Department Chair Per Reinhall for his approval for us to offer the class that helped to inspire the creation of much of the book's content. Additional thanks go to our colleagues David Haynor of University of Washington Department of Radiology and William Ledoux of the Seattle VA Hospital, whose research initiatives continue to motivate meaningful journeys into CUDA territory.

We wish to say a special thank you to the good folks at NVIDIA, including CEO Jen-Hsun Huang who not only had, but also acted on, a vision of what could be accomplished by enhancing access to GPU-based parallel computing; Chandra Cheij, Academic Programs Manager; Kimberly Powell, Director of Higher Education and Healthcare Industries; Jon Saposhnik and Bob Crovella, helpful and inspirational CUDA gurus; and last, but definitely not least, Jay White, Director

of Strategic Marketing, who sustains the Seattle-area GPU-computing meet-up group and serves as our local go-to guy.

We would also like to thank all the students who had the sense of adventure to participate in the initial CUDA-based class offerings at the University of Washington, especially former ME graduate student Grant Marchelli (now Grant Marchelli, Ph.D., CTO of Envitrum Inc.), who played a key role in everything from setting up the lab to providing code samples and delivering guest lectures. A special thank you goes to Gerald Barnett, who was so generous with his time and expertise when it came time to edit the first draft.

Finally, a big thanks to you, the reader. The value of having something important to share depends on having people to share it with; we appreciate your interest, and we sincerely hope this book provides you with useful and productive experiences.

About the Authors

Duane Storti received a Ph.D. in theoretical and applied mechanics from Cornell University in 1984. Since then, he has served as a professor of mechanical engineering at the University of Washington–Seattle. Duane has 35 years of experience in teaching and research in the areas of engineering mathematics, dynamics and vibrations, computer-aided design, 3D printing, and applied GPU computing. When not on campus, he can often be found in the gym, coaching youth volleyball.

Mete Yurtoglu received a B.S. in physics and a B.S. in mechanical engineering in 2008, and an M.S. in 2011 from Bogazici University in Istanbul, Turkey. He is currently a graduate student at the University of Washington–Seattle pursuing an M.S. in applied mathematics and a Ph.D. in mechanical engineering. His research interests focus on GPU-based methods for computer vision and machine learning. Mete enjoys family time, playing soccer, and working out.

This page intentionally left blank

Chapter 4

2D Grids and Interactive Graphics

In this chapter, we see that the CUDA model of parallelism extends readily to two dimensions (2D). We go through the basics of launching a 2D computational grid and create a skeleton kernel you can use to compute a 2D grid of values for functions of interest to you. We then specialize the kernel to create `dist_2d`, an app that computes the distance from a reference point in the plane to each member of a uniform 2D grid of points. By identifying the grid of points with pixels in an image, we compute data for an image whose shading is based on distance values.

Once we are generating image data, it is only natural to take advantage of CUDA's **graphics interoperability** (or **graphics interop** for short) capability, which supports cooperation with standard graphics **application programming interfaces** (**APIs**) including **Direct3D** [1] and **OpenGL** [2]. We'll use OpenGL, and maintaining our need-to-know approach, we'll very quickly provide just the necessities of OpenGL to get your results on the screen at interactive speeds.

By the end of this chapter you will have run a `flashlight` app that interactively displays an image with shading based on distance from a reference point that you can move using mouse or keyboard input and a `stability` app that interactively displays the results of several hundred thousand numerical simulations of the dynamics of an oscillator. This experience should get you to the point where you are ready to start creating your own CUDA-powered interactive apps.

Launching 2D Computational Grids

Here we expand on our earlier examples that involved a 1D array (points distributed regularly along a line segment) and move on to consider applications involving points regularly distributed on a portion of a 2D plane. While we will encounter other applications (e.g., simulating heat conduction) that fit this scenario, the most common (and likely most intuitive) example involves digital image processing. To take advantage of the intuitive connection, we will use image-processing terminology in presenting the concepts—all of which will transfer directly to other applications.

A digital raster image consists of a collection of picture elements or **pixels** arranged in a uniform 2D rectangular grid with each pixel having a quantized intensity value. To be concrete, let's associate the width and height directions with the `x` and `y` coordinates, respectively, and say that our image is `W` pixels wide by `H` pixels high. If the quantized value stored in each pixel is simply a number, the data for an image matches exactly with the data for a matrix of size `W x H`.

As we move on from 1D to 2D problems in CUDA, we hope you will be pleasantly surprised by how few adjustments need to be made. In 1D, we specified integer values for block and grid sizes and computed an index `i` based on `blockDim.x`, `blockIdx.x`, and `threadIdx.x` according to the formula

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
```

Here we reinterpret the expression on the right-hand side of the assignment as the specification of a new index `c` that keeps track of what column each pixel belongs to. (As we traverse a row of pixels from left to right, `c` increases from its minimum value `0` to its maximum value `W-1`.) We also introduce a second index `r` to keep track of row numbers (ranging from `0` to `H-1`). The row index is computed just as the column index is, but using the `.y` components (instead of the `.x` components), so the column and row indices are computed as follows:

```
int c = blockIdx.x*blockDim.x + threadIdx.x;
int r = blockIdx.y*blockDim.y + threadIdx.y;
```

To keep data storage and transfer simple, we will continue to store and transfer data in a "flat" 1D array, so we will have one more integer variable to index into the 1D array. We will continue to call that variable `i`, noting that `i` played this role in the 1D case, but in other places (including the CUDA Samples) you will see variables named `idx`, `flatIdx`, and `offset` indexing the 1D array. We place values in the 1D array in row major order—that is, by storing the data from row 0, followed by the data from row 1, and so on—so the index `i` in the 1D array is now computed as follows:

```
int i = r*w + c;
```

To describe the 2D computational grid that intuitively matches up with an image (or matrix or other regular 2D discretization), we specify block and grid sizes using `dim3` variables with two nontrivial components. Recall that an integer within the triple chevrons of a kernel call is treated as the `.x` component of a `dim3` variable with a default value of 1 for the unspecified `.y` and `.z` components. In the current 2D context, we specify nontrivial `.x` and `.y` components. The `.z` component of the `dim3`, which here has the default value 1, will come into play when we get to 3D grids in Chapter 7, "Interacting with 3D Data."

Without further ado, let's lay out the necessary syntax and get directly to parallel computation of pixel values with a 2D grid.

SYNTAX FOR 2D KERNEL LAUNCH

The 2D kernel launch differs from the 1D launch only in terms of the execution configuration. Computing data for an image involves `W` columns and `H` rows, and we can organize the computation into 2D blocks with `TX` threads in the `x`-direction and `TY` threads in the `y`-direction. (You can choose to organize your 2D grid into 1D blocks, but you will run into limits on both maximum block dimension and total number of threads in a block. See the CUDA C Programming Guide [3] for details.)

We specify the 2D block size with a single statement:

```
dim3 blockSize(TX, TY); // Equivalent to dim3 blockSize(TX, TY, 1);
```

and then we compute the number of blocks (`bx` and `by`) needed in each direction exactly as in the 1D case.

```
int bx = (W + blockSize.x - 1)/blockSize.x ;
int by = (H + blockSize.y – 1)/blockSize.y ;
```

The syntax for specifying the grid size (in blocks) is

```
dim3 gridSize = dim3(bx, by);
```

With those few details in hand, we are ready to launch:

```
kernelName<<<gridSize, blockSize>>>(args)
```

DEFINING 2D KERNELS

The prototype or declaration of a kernel to be launched on a 2D grid will look exactly as before: it starts with the qualifier `__global__` followed by return type `void` and a legal name, such as `kernel2D`, and ends with a comma-separated list of typed arguments (which better include a pointer to a device array `d_out` where the computed image data will be stored, along with the width and height of the image and any other required inputs). The `kernel2D` function begins by computing the row, column, and flat indices and testing that the row and column indices have values corresponding to a pixel within the image. All that is left is computing the value for the pixel.

Putting the pieces together, the structure of a typical 2D kernel is given in Listing 4.1.

Listing 4.1 “Skeleton” listing for a kernel to be launched on a 2D grid. Replace `INSERT_CODE_HERE` with your code for computing the output value.

```
__global__
void kernel2D(float *d_out, int w, int h, … )
{
  // Compute column and row indices.
  const int c = blockIdx.x * blockDim.x + threadIdx.x;
  const int r = blockIdx.y * blockDim.y + threadIdx.y;
  const int i = r * w + c; // 1D flat index

  // Check if within image bounds.
  if ((c >= w) || (r >= h))
    return;

  d_out[i] = INSERT_CODE_HERE; // Compute/store pixel in device array.
}
```

A Note on Capitalization of Variable Names

We need to refer to parameter values such as the width and height of an image inside of function definitions where they are considered as input variables, but the input value in the function call will typically be a constant value specified using `#define`. We will follow the prevailing convention by using uppercase for the constant value and the same name in lowercase for the input variable. For example, the function `kernel2D()` in Listing 4.1 has the prototype

```
void kernel2D(uchar4 *d_out, int w, int h, … )
```

and the function call

```
#define W 500
#define H 500
kernel2D<<<gridSize, blockSize>>>(d_out, W, H, … )
```

indicates that the input values for width and height are constants, here with value 500.

One detail worth dealing with at this point is a common data type for images. The quantized value stored for each pixel is of type `uchar4`, which is a vector type storing four unsigned character values (each of which occupies 1 byte of storage). For practical purposes, you can think of the four components of the `uchar4` (designated as usual by suffixes `.x`, `.y`, `.z`, and `.w`) as specifying integer values ranging from 0 to 255 for the red, green, blue, and alpha (opacity) display channels. This format for describing pixel values in an image is often abbreviated as **RGBA**.

Putting the pieces together, the structure of a typical 2D kernel for computing an image is given in Listing 4.2.

Listing 4.2 “Skeleton” listing for computing data for an image. `RED_FORMULA`, `GREEN_FORMULA`, and `BLUE_FORMULA` should be replaced with your code for computing desired values between 0 and 255 for each color channel.

```
__global__
void kernel2D(uchar4 *d_output, int w, int h, … )
{
  // Compute column and row indices.
  int c = blockIdx.x*blockDim.x + threadIdx.x;
  int r = blockIdx.y*blockDim.y + threadIdx.y;
  int i = r * w + c; // 1D flat index

  // Check if within image bounds.
  if ((r >= h) || (c >= w)) {
      return;
  }

  d_output[i].x = RED_FORMULA;    //Compute red
  d_output[i].y = GREEN_ FORMULA; //Compute green
  d_output[i].z = BLUE_ FORMULA;  //Compute blue
  d_output[i].w = 255; // Fully opaque
}
```

`dist_2d`

Let’s tie the general discussion of 2D grids together with our earlier examples involving distance apps by coding up an app that produces a 2D array of distances from a reference point, and then we’ll adapt the app to produce an array of data for an RGBA image. Listing 4.3 provides all the code for computing distances on a 2D grid.

Listing 4.3 Computing distances on a 2D grid

```
 1 #define W 500
 2 #define H 500
 3 #define TX 32 // number of threads per block along x-axis
 4 #define TY 32 // number of threads per block along y-axis
 5
 6 __global__
 7 void distanceKernel(float *d_out, int w, int h, float2 pos)
 8 {
 9   const int c = blockIdx.x*blockDim.x + threadIdx.x;
10   const int r = blockIdx.y*blockDim.y + threadIdx.y;
11   const int i = r*w + c;
12   if ((c >= w) || (r >= h)) return;
13
14   // Compute the distance and set d_out[i]
15   d_out[i] = sqrtf((c - pos.x)*(c - pos.x) +
16                    (r - pos.y)*(r - pos.y));
17 }
18
19 int main()
20 {
21   float *out = (float*)calloc(W*H, sizeof(float));
22   float *d_out; // pointer for device array
23   cudaMalloc(&d_out, W*H*sizeof(float));
24
25   const float2 pos = {0.0f, 0.0f}; // set reference position
26   const dim3 blockSize(TX, TY);
27   const int bx = (W + TX - 1)/TX;
28   const int by = (W + TY - 1)/TY;
29   const dim3 gridSize = dim3(bx, by);
30
31   distanceKernel<<<gridSize, blockSize>>>(d_out, W, H, pos);
32
33   // Copy results to host.
34   cudaMemcpy(out, d_out, W*H*sizeof(float), cudaMemcpyDeviceToHost);
35
36   cudaFree(d_out);
37   free(out);
38   return 0;
39 }
```

The kernel, lines 6–17, is exactly as in Listing 4.1 but with a result computed using the Pythagorean formula to compute the distance between the location {`c`, `r`} and a reference location `pos`. (Note that we have defined `pos` to have type `float2` so it can store both coordinates of the reference location {`pos.x`, `pos.y`}.) The rest of the listing, lines 19–39, gives the details of `main()` starting with declaration of an output array of appropriate size initialized to zero. Lines

22–23 declare a pointer to the device array `d_out` and allocate the memory with `cudaMalloc()`. Line 25 sets the reference position, and lines 26–29 set the kernel launch parameters: a 2D grid of `bx` × `by` blocks each having `TX` × `TY` threads. Line 31 launches the kernel to compute the distance values, which are copied back to `out` on the host side on line 34. Lines 36–37 free the allocated device and host memory, then `main()` returns zero to indicate completion.

Next we make a few minor changes to produce an app that computes an array of RGBA values corresponding to a distance image. The full code is provided in Listing 4.4.

Listing 4.4 Parallel computation of image data based on distance from a reference point in 2D

```
#define W  500
#define H  500
#define TX 32 // number of threads per block along x-axis
#define TY 32 // number of threads per block along y-axis

__device__
unsigned char clip(int n) { return n > 255 ? 255 : (n < 0 ? 0 : n); }

__global__
void distanceKernel(uchar4 *d_out, int w, int h, int2 pos)
{
  const int c = blockIdx.x*blockDim.x + threadIdx.x;
  const int r = blockIdx.y*blockDim.y + threadIdx.y;
  const int i = r*w + c;
  if ((c >= w) || (r >= h)) return;

  // Compute the distance (in pixel spacings)
  const int d = sqrtf((c - pos.x) * (c - pos.x) +
                      (r - pos.y) * (r - pos.y));
  // Convert distance to intensity value on interval [0, 255]
  const unsigned char intensity = clip(255 - d);

  d_out[i].x = intensity; // red channel
  d_out[i].y = intensity; // green channel
  d_out[i].z = 0; // blue channel
  d_out[i].z = 255; // fully opaque
}

int main()
{
  uchar4 *out = (uchar4*)calloc(W*H, sizeof(uchar4));
  uchar4 *d_out; // pointer for device array
  cudaMalloc(&d_out, W*H*sizeof(uchar4));

  const int2 pos = {0, 0}; // set reference position
  const dim3 blockSize(TX, TY);
  const int bx = (W + TX - 1)/TX;
  const int by = (W + TY - 1)/TY;
```

```
  const dim3 gridSize = dim3(bx, by);

  distanceKernel<<<gridSize, blockSize>>>(d_out, W, H, pos);

  // Copy results to host.
  cudaMemcpy(out, d_out, W*H*sizeof(uchar4), cudaMemcpyDeviceToHost);

  cudaFree(d_out);
  free(out);
  return 0;
}
```

Here the distance is computed in pixel spacings, so the reference position, `pos`, now has type `int2`, and the distance `d` has type `int`. The distance value is then converted to `intensity` of type `unsigned char`, whose value is restricted to the allowed range of 0 to 255 using the function `clip()`. The output arrays, `out` and `d_out`, have the corresponding vector type `uchar4`. The assignments `d_out[i].x = intensity` and `d_out[i].y = intensity` store the intensity value in the red and green channels to produce a yellow distance image. (We set the blue component to zero and the alpha to 255, corresponding to full opacity, but you should experiment with other color specifications.)

Live Display via Graphics Interop

Now that we can construct apps that produce image data, it makes sense to start displaying those images and exploring what CUDA's massive parallelism enables us to do in real time.

Real-time graphic interactivity will involve CUDA's provision for interoperability with a standard graphics package. We will be using OpenGL, which could be (and is) the subject of numerous books all by itself [2,4,5], so we will take our usual need-to-know approach. We introduce just enough OpenGL to display a single textured rectangle and provide a few examples of code to support interactions via keyboard and mouse with the help of the **OpenGL Utility Toolkit (GLUT)**. The idea is that the rectangle provides a window into the world of your app, and you can use CUDA to compute the pixel shading values corresponding to whatever scene you want the user to see. CUDA/OpenGL interop provides interactive controls and displays the changing scene as a texture on the displayed rectangle in real time (or, more accurately, at a rate comparable to the ~60Hz refresh rate typical of modern visual display systems).

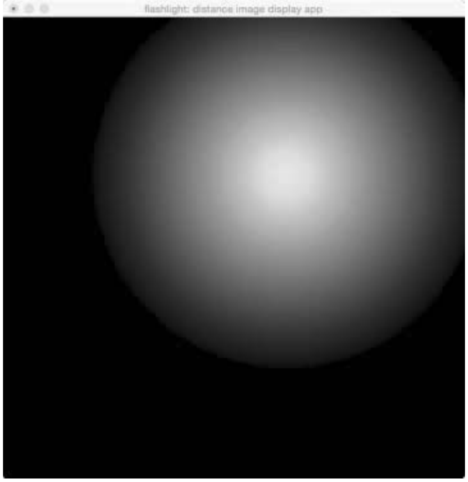


Figure 4.1 Interactive spot of light in the finished application

Here we present the code for a sample app that opens a graphics window and interactively displays an image based on distance to a reference point that can be changed interactively using keyboard or mouse input. We call the app `flashlight` because it produces a directable circle of light whose intensity diminishes away from the center of the "spot." Figure 4.1 shows the screenshot of the app in its finished state.

This entire app requires a total of less than 200 lines of code, which we have organized into three files:

- `main.cpp` contains the essentials of the CUDA/OpenGL set up and interop. It is about 100 lines of code (half of the total), and while we will provide a brief explanation of its contents, you should be able to create your own apps by using `flashlight` as a template by making only minor changes to `main.cpp`.
- `kernel.cu` contains the essential CUDA code, including the `clip()` function described above, the definition of the `kernelLauncher()` function, and the definition of the actual kernel function (here `distanceKernel()`), which must write its output to a `uchar4` array.
- `interactions.h` defines the **callback** functions `keyboard()`, `mouseMove()`, and `mouseDrag()` to specify how the system should respond to inputs.

While we will go through the entire code, the important point is that you can use the `flashlight` app as a template to readily create your own apps in just a few steps:

1. Create a new app based on `flashlight` by making a copy of the code directory under Linux or by creating a new project using `flashlight` as a template in Visual Studio under Windows.
2. Edit the kernel function to produce whatever data you want to display.
3. In `interactions.h`, edit the callback functions to specify how your app should respond to keyboard and mouse inputs, and edit `printInstructions()` to customize the instructions for user interactions.
4. Optionally, edit the `#define TITLE_STRING` statement in `interactions.h` to customize the app name in the title bar of the graphics window.

Listings 4.5, 4.6, 4.7, and 4.8 show all the code necessary to display a distance image on your screen using CUDA/OpenGL interop, and we will walk you through the necessities while trying not to get hung up on too many details.

Listing 4.5 `flashlight/main.cpp`

```
#include "kernel.h"
#include <stdio.h>
#include <stdlib.h>
#ifdef _WIN32
#define WINDOWS_LEAN_AND_MEAN
#define NOMINMAX
#include <windows.h>
#endif
#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glew.h>
#include <GL/freeglut.h>
#endif
#include <cuda_runtime.h>
#include <cuda_gl_interop.h>
#include "interactions.h"

// texture and pixel objects
GLuint pbo = 0;     // OpenGL pixel buffer object
GLuint tex = 0;     // OpenGL texture object
struct cudaGraphicsResource *cuda_pbo_resource;

void render() {
  uchar4 *d_out = 0;
  cudaGraphicsMapResources(1, &cuda_pbo_resource, 0);
  cudaGraphicsResourceGetMappedPointer((void **)&d_out, NULL,
                                       cuda_pbo_resource);
```

```
  kernelLauncher(d_out, W, H, loc);
  cudaGraphicsUnmapResources(1, &cuda_pbo_resource, 0);
}

void drawTexture() {
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, W, H, 0, GL_RGBA,
               GL_UNSIGNED_BYTE, NULL);
  glEnable(GL_TEXTURE_2D);
  glBegin(GL_QUADS);
  glTexCoord2f(0.0f, 0.0f); glVertex2f(0, 0);
  glTexCoord2f(0.0f, 1.0f); glVertex2f(0, H);
  glTexCoord2f(1.0f, 1.0f); glVertex2f(W, H);
  glTexCoord2f(1.0f, 0.0f); glVertex2f(W, 0);
  glEnd();
  glDisable(GL_TEXTURE_2D);
}

void display() {
  render();
  drawTexture();
  glutSwapBuffers();
}

void initGLUT(int *argc, char **argv) {
  glutInit(argc, argv);
  glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
  glutInitWindowSize(W, H);
  glutCreateWindow(TITLE_STRING);
#ifndef __APPLE__
  glewInit();
#endif
}

void initPixelBuffer() {
  glGenBuffers(1, &pbo);
  glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo);
  glBufferData(GL_PIXEL_UNPACK_BUFFER, 4*W*H*sizeof(GLubyte), 0,
               GL_STREAM_DRAW);
  glGenTextures(1, &tex);
  glBindTexture(GL_TEXTURE_2D, tex);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  cudaGraphicsGLRegisterBuffer(&cuda_pbo_resource, pbo,
                               cudaGraphicsMapFlagsWriteDiscard);
}

void exitfunc() {
  if (pbo) {
    cudaGraphicsUnregisterResource(cuda_pbo_resource);
    glDeleteBuffers(1, &pbo);
    glDeleteTextures(1, &tex);
  }
}

int main(int argc, char** argv) {
  printInstructions();
```

```
84   initGLUT(&argc, argv);
85   gluOrtho2D(0, W, H, 0);
86   glutKeyboardFunc(keyboard);
87   glutSpecialFunc(handleSpecialKeypress);
88   glutPassiveMotionFunc(mouseMove);
89   glutMotionFunc(mouseDrag);
90   glutDisplayFunc(display);
91   initPixelBuffer();
92   glutMainLoop();
93   atexit(exitfunc);
94   return 0;
95 }
```

This is the brief, high-level overview of what is happening in `main.cpp`. Lines 1–17 load the header files appropriate for your operating system to access the necessary supporting code. The rest of the explanation should start from the bottom. Lines 82–95 define `main()`, which does the following things:

- Line 83 prints a few user interface instructions to the command window.
- `initGLUT` initializes the GLUT library and sets up the specifications for the graphics window, including the display mode (RGBA), the buffering (double), size (`W` x `H`), and title.
- `gluOrtho2D(0, W, H, 0)` establishes the viewing transform (simple orthographic projection).
- Lines 86–89 indicate that keyboard and mouse interactions will be specified by the functions `keyboard`, `handleSpecialKeypress`, `mouseMove`, and `mouseDrag` (the details of which will be specified in `interactions.h`).
- `glutDisplayFunc(display)` says that what is to be shown in the window is determined by the function `display()`, which is all of three lines long. On lines 47–49, it calls `render()` to compute new pixel values, `drawTexture()` to draw the OpenGL texture, and then swaps the display buffers.
    - `drawTexture()` sets up a 2D OpenGL texture image, creates a single quadrangle graphics primitive with texture coordinates (0.0f, 0.0f), (0.0f, 1.0f), (1.0f, 1.0f), and (1.0f, 0.0f); that is, the corners of the unit square, corresponding with the pixel coordinates (0, 0), (0, H), (W, H), and (W, 0).
    - Double buffering is a common technique for enhancing the efficiency of graphics programs. One buffer provides memory that can be read to “feed” the display, while at the same time, the other buffer provides memory into which the contents of the next frame can be written. Between frames in a graphics sequence, the buffers swap their read/write roles.

- `initPixelBuffer()`, not surprisingly, initializes the pixel buffer on lines 62–72. The key for our purposes is the last line which "registers" the OpenGL buffer with CUDA. This operation has some overhead, but it enables low-overhead "mapping" that turns over control of the buffer memory to CUDA to write output and "unmapping" that returns control of the buffer memory to OpenGL for display. Figure 4.2 shows a summary of the interop between CUDA and OpenGL.

- `glutMainLoop()`, on line 92, is where the real action happens. It repeatedly checks for input and calls for computation of updated images via `display` that calls `render`, which does the following:

  - Maps the pixel buffer to CUDA and gets a CUDA pointer to the buffer memory so it can serve as the output device array

  - Calls the wrapper function `kernelLauncher` that launches the kernel to compute the pixel values for the updated image

  - Unmaps the buffer so OpenGL can display the contents

- When you exit the app, `atexit(exitfunc)` performs the final clean up by undoing the resource registration and deleting the OpenGL pixel buffer and texture before zero is returned to indicate completion of `main()`.

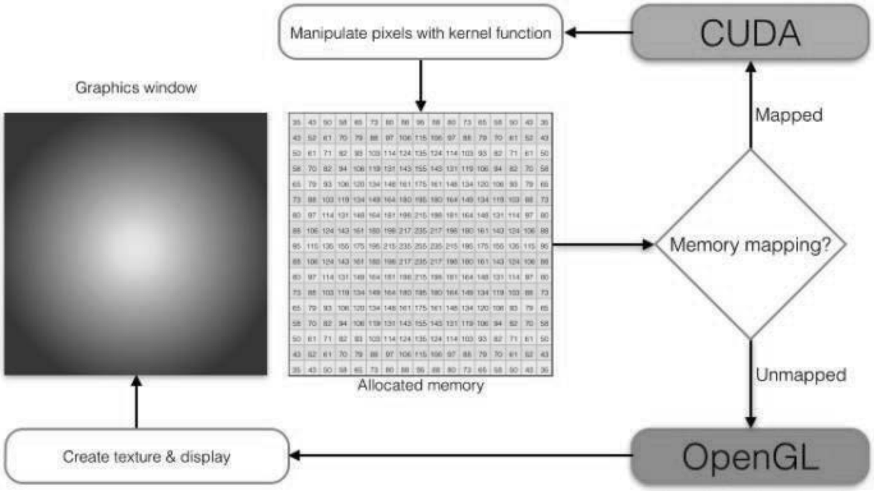


Figure 4.2 Illustration of alternating access to device memory that is mapped to CUDA to store computational results and unmapped (i.e., returned to OpenGL control) for display of those results

Of all the code in `main.cpp`, the only thing you need to change when you create your own CUDA/OpenGL interop apps is the `render()` function, where you will need to update the argument list for `kernelLauncher()`.

Listing 4.6 `flashlight/kernel.cu`

```
#include "kernel.h"
#define TX 32
#define TY 32

__device__
unsigned char clip(int n) { return n > 255 ? 255 : (n < 0 ? 0 : n); }

__global__
void distanceKernel(uchar4 *d_out, int w, int h, int2 pos) {
  const int c = blockIdx.x*blockDim.x + threadIdx.x;
  const int r = blockIdx.y*blockDim.y + threadIdx.y;
  if ((c >= w) || (r >= h)) return; // Check if within image bounds
  const int i = c + r*w; // 1D indexing
  const int dist = sqrtf((c - pos.x)*(c - pos.x) +
                         (r - pos.y)*(r - pos.y));
  const unsigned char intensity = clip(255 - dist);
  d_out[i].x = intensity;
  d_out[i].y = intensity;
  d_out[i].z = 0;
  d_out[i].w = 255;
}

void kernelLauncher(uchar4 *d_out, int w, int h, int2 pos) {
  const dim3 blockSize(TX, TY);
  const dim3 gridSize = dim3((w + TX - 1)/TX, (h + TY - 1)/TY);
  distanceKernel<<<gridSize, blockSize>>>(d_out, w, h, pos);
}
```

The code from `kernel.cu` in Listing 4.6 should look familiar and require little explanation at this point. The primary change is a wrapper function `kernelLauncher()` that computes the grid dimensions and launches the kernel. Note that you will not find any mention of a host output array. Computation and display are both handled from the device, and there is no need to transfer data to the host. (Such a transfer of large quantities of image data across the PCIe bus could be time-consuming and greatly inhibit real-time interaction capabilities.) You will also not find a `cudaMalloc()` to create space for a device array. The `render()` function in `main.cpp` declares a pointer `d_out` that gets its value from `cudaGraphicsResourceGetMappedPointer()` and provides the CUDA pointer to the memory allocated for the pixel buffer.

The header file associated with the kernel is shown in Listing 4.7. In addition to the include guard and kernel function prototype, `kernel.h` also contains forward declarations for `uchar4` and `int2` so that the compiler knows of their existence before the CUDA code (which is aware of their definitions) is built or executed.

Listing 4.7 `flashlight/kernel.h`

```
#ifndef KERNEL_H
#define KERNEL_H

struct uchar4;
struct int2;

void kernelLauncher(uchar4 *d_out, int w, int h, int2 pos);

#endif
```

Listing 4.8 `flashlight/interactions.h` that specifies callback functions controlling interactive behavior of the `flashlight` app

```
#ifndef INTERACTIONS_H
#define INTERACTIONS_H
#define W 600
#define H 600
#define DELTA 5 // pixel increment for arrow keys
#define TITLE_STRING "flashlight: distance image display app"
int2 loc = {W/2, H/2};
bool dragMode = false; // mouse tracking mode

void keyboard(unsigned char key, int x, int y) {
  if (key == 'a') dragMode = !dragMode; // toggle tracking mode
  if (key == 27)  exit(0);
  glutPostRedisplay();
}

void mouseMove(int x, int y) {
  if (dragMode) return;
  loc.x = x;
  loc.y = y;
  glutPostRedisplay();
}

void mouseDrag(int x, int y) {
  if (!dragMode) return;
  loc.x = x;
  loc.y = y;
  glutPostRedisplay();
}

void handleSpecialKeypress(int key, int x, int y) {
  if (key == GLUT_KEY_LEFT)  loc.x -= DELTA;
  if (key == GLUT_KEY_RIGHT) loc.x += DELTA;
  if (key == GLUT_KEY_UP)    loc.y -= DELTA;
  if (key == GLUT_KEY_DOWN)  loc.y += DELTA;
```

```
35   glutPostRedisplay();
36 }
37
38 void printInstructions() {
39   printf("flashlight interactions\n");
40   printf("a: toggle mouse tracking mode\n");
41   printf("arrow keys: move ref location\n");
42   printf("esc: close graphics window\n");
43 }
44
45 #endif
```

The stated goal of the `flashlight` app is to display an image corresponding to the distance to a reference point that can be moved interactively, and we are now ready to define and implement the interactions. The code for `interactions.h` shown in Listing 4.8 allows the user to move the reference point (i.e., the center of the flashlight beam) by moving the mouse or pressing the arrow keys. Pressing `a` toggles between tracking mouse motions and tracking mouse drags (with the mouse button pressed), and the `esc` key closes the graphics window. Here's a quick description of what the code does and how those interactions work:

- Lines 3–6 set the image dimensions, the text displayed in the title bar, and how far (in pixels) the reference point moves when an arrow key is pressed.
- Line 7 sets the initial reference location at `{W/2, H/2}`, the center of the image.
- Line 8 declares a Boolean variable `dragMode` that is initialized to `false`. We use `dragMode` to toggle back and forth between tracking mouse motions and "click-drag" motions.
- Lines 10–14 specify the defined interactions with the keyboard:
  - Pressing the `a` key toggles `dragMode` to switch the mouse tracking mode.
  - The ASCII code `27` corresponds to the `Esc` key. Pressing `Esc` closes the graphics window.
  - `glutPostRedisplay()` is called at the end of each callback function telling to compute a new image for display (by calling `display()` in `main.cpp`) based on the interactive input.
- Lines 16–21 specify the response to a mouse movement. When `dragMode` is toggled, `return` ensures that no action is taken. Otherwise, the components of the reference location are set to be equal to the `x` and `y` coordinates of the mouse before computing and displaying an updated image (via `glutPostRedisplay()`).

- Lines 23–28 similarly specify the response to a “click-drag.” When `dragMode` is `false`, `return` ensures that no action is taken. Otherwise, the reference location is reset to the last location of the mouse while the mouse was clicked.
- Lines 30–36 specify the response to special keys with defined actions. (Note that standard keyboard interactions are handled based on ASCII key codes [6], so special keys like arrow keys and function keys that do not generate standard ASCII codes need to be handled separately.) The `flashlight` app is set up so that depressing the arrow keys moves the reference location `DELTA` pixels in the desired direction.
- The `printInstructions()` function on lines 38–43 consists of print statements that provide user interaction instructions via the console.

While all the code and explanation for the `flashlight` app took about nine pages, let’s pause to put things in perspective. While we presented numbered listings totaling about 200 lines, if we were less concerned about readability, the entire code could be written in many fewer lines, so there is not a lot of code to digest. Perhaps more importantly, over half of those lines reside in `main.cpp`, which you should not really need to change at all to create your own apps other than to alter the list of arguments for the `kernelLauncher()` function or to customize the information displayed in the title bar. If you start with the `flashlight` app as a template, you should be able to (and are heartily encouraged to) harness the power of CUDA to create your own apps with interactive graphics by replacing the kernel function with one of your own design and by revising the collection of user interactions implemented in `interactions.h`.

Finally, the Makefile for building the app in Linux is provided in Listing 4.9.

Listing 4.9 `flashlight/Makefile`

```
UNAME_S := $(shell uname)

ifeq ($(UNAME_S), Darwin)
  LDFLAGS = -Xlinker -framework,OpenGL -Xlinker -framework,GLUT
else
  LDFLAGS += -L/usr/local/cuda/samples/common/lib/linux/x86_64
  LDFLAGS += -lglut -lGL -lGLU -lGLEW
endif

NVCC = /usr/local/cuda/bin/nvcc
NVCC_FLAGS = -g -G -Xcompiler "-Wall -Wno-deprecated-declarations"

all: main.exe

```

```
main.exe: main.o kernel.o
	$(NVCC) $^ -o $@ $(LDFLAGS)

main.o: main.cpp kernel.h interactions.h
	$(NVCC) $(NVCC_FLAGS) -c $< -o $@

kernel.o: kernel.cu kernel.h
	$(NVCC) $(NVCC_FLAGS) -c $< -o $@
```

Windows users will need to change one build customization and include two pairs of library files: the **OpenGL Utility Toolkit** (**GLUT**) and the **OpenGL Extension Wrangler** (**GLEW**). To keep things simple and ensure consistency of the library version, we find it convenient to simply make copies of the library files (which can be found by searching within the CUDA Samples directory for the filenames `freeglut.dll`, `freeglut.lib`, `glew64.dll`, and `glew64.lib`), save them to the project directory, and then add them to the project with `PROJECT ⇒ Add Existing Item`.

The build customization is specified using the Project Properties pages: Right-click on `flashlight` in the Solution Explorer pane, then select `Properties ⇒ Configuration Properties ⇒ C/C++ ⇒ General ⇒ Additional Include Directories` and edit the list to include the CUDA Samples' `common\inc` directory. Its default install location is `C:\ProgramData\NVIDIA Corporation\CUDA Samples\v7.5\common\inc`.

Application: Stability

To drive home the idea of using the `flashlight` app as a template for creating more interesting and useful apps, let's do exactly that. Here we build on `flashlight` to create an app that analyzes the stability of a linear oscillator, and then we extend the app to handle general single degree of freedom (1DoF) systems, including the van der Pol oscillator, which has more interesting behavior.

The linear oscillator arises from models of a mechanical mass-spring-damper system, an electrical RLC circuit, and the behavior of just about any 1DoF system in the vicinity of an equilibrium point. The mathematical model consists of a single second-order ordinary differential equation (ODE) that can be written in its simplest form (with suitable choice of time unit) as $x'' + 2bx' + x = 0$, where x is the displacement from the equilibrium position, b is the damping constant, and the primes indicate time derivatives. To put things in a handy form for finding solutions, we convert to a system of two first-order ODEs by introducing the velocity y as a new variable and writing the first-order ODEs that give the rate of change of x and y:

$$\begin{aligned} x' &= y \\ y' &= -x - 2by = f\left(x,\ y,\ t,\ \ldots\right) \end{aligned}$$

As a bit of foreshadowing, everything we do from here generalizes to a wide variety of 1DoF oscillators by just plugging other expressions in for $f(x, y, t, \ldots)$ on the right-hand side of the y-equation. While we can write analytical solutions for the linear oscillator, here we focus on numerical solutions using finite difference methods that apply to the more general case. Finite difference methods compute values at discrete multiples of the time step dt (so we introduce $t_k = k * dt$, $x_k = x(t_k)$, and $y_k = y(t_k)$ as the relevant variables) and replace exact derivatives by difference approximations; that is, $x' \rightarrow (x_{k+1} - x_k) / dt$, $y' \rightarrow (y_{k+1} - y_k) / dt$. Here we apply the simplest finite difference approach, the explicit Euler method, by substituting the finite difference expressions for the derivatives and solving for the new values at the end of the time step, x_{k+1} and y_{k+1}, in terms of the previous values at the beginning of a time step, x_k and y_k, to obtain:

$$\begin{aligned} x_{k+1} &= x_k + dt^* y_k \\ y_{k+1} &= y_k + dt^*\left(-x_k - 2by_k\right) \end{aligned}$$

We can then choose an initial state $\{x_o, y_o\}$ and compute the state of the system at successive time steps.

We've just described a method for computing a solution (a sequence of states) arising from a single initial state, and the solution method is completely serial: Entries in the sequence of states are computed one after another.

However, stability depends not on the solution for one initial state but on the solutions for *all* initial states. For a stable equilibrium, all nearby initial states produce solutions that approach (or at least don't get further from) the equilibrium. Finding a solution that grows away from the equilibrium indicates instability. For more information on dynamics and stability, see [7,8].

It is this collective-behavior aspect that makes stability testing such a good candidate for parallelization: By launching a computational grid with initial states densely sampling the neighborhood of the equilibrium, we can test the solutions arising from the surrounding initial states. We'll see that we can compute hundreds of thousands of solutions in parallel and, with CUDA/OpenGL interop, see and interact with the results in real time.

In particular, we'll choose a grid of initial states that regularly sample a rectangle centered on the equilibrium. We'll compute the corresponding solutions and assign shading values based on the fractional change in distance, `dist_r` (for distance ratio) from the equilibrium during the simulation. To display the results, we'll assign each pixel a red channel value proportional to the distance ratio (and clipped to [0, 255]) and a blue channel value proportional to the inverse distance ratio (and clipped). Initial states producing solutions that are attracted to the equilibrium (and suggest stability) are dominated by blue, while initial states that produce solutions being repelled from the equilibrium are dominated by red, and the attracting/repelling transition is indicated by equal parts of blue and red; that is, purple.

Color Adjustment to Enhance Grayscale Contrast

Since it is difficult to see the difference between red (R) and blue (B) when viewing figures converted to grayscale, the figures included here use the green (G) channel to enhance contrast and brightness according to the formula $G = 0.3 + (R - B) / 2$. Full color images produced by the `stability` app are available at www.cudaforengineers.com.

The result shown in the graphics window will then consist of the equilibrium (at the intersection of the horizontal *x*-axis and the vertical *y*-axis shown using the green channel) on a field of red, blue, or purple pixels. Figure 4.3 previews a result from the stability application with both attracting and repelling regions.

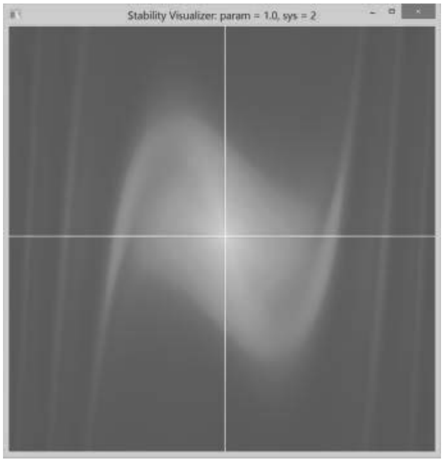


Figure 4.3 Stability map with shading adjusted to show a bright central repelling region and surrounding darker attracting region

We now have a plan for producing a stability image for a single system, but we will also introduce interactions so we can observe how the stability image changes for different parameter values or for different systems.

With the plan for the kernel and the interactions in mind, we are ready to look at the code. As promised, the major changes from the `flashlight` app involve a new kernel function (and a few supporting functions), as shown in Listing 4.10, and new interactivity specifications, as shown in Listing 4.11.

Listing 4.10 `stability/kernel.cu`

```
#include "kernel.h"
#define TX 32
#define TY 32
#define LEN 5.f
#define TIME_STEP 0.005f
#define FINAL_TIME 10.f

// scale coordinates onto [-LEN, LEN]
__device__
float scale(int i, int w) { return 2*LEN*(((1.f*i)/w) - 0.5f); }

// function for right-hand side of y-equation
__device__
float f(float x, float y, float param, float sys) {
  if (sys == 1) return x - 2*param*y; // negative stiffness
  if (sys == 2) return -x + param*(1 - x*x)*y; //van der Pol
  else return -x - 2*param*y;
}

// explicit Euler solver
__device__
float2 euler(float x, float y, float dt, float tFinal,
             float param, float sys) {
  float dx = 0.f, dy = 0.f;
  for (float t = 0; t < tFinal; t += dt) {
    dx = dt*y;
    dy = dt*f(x, y, param, sys);
    x += dx;
    y += dy;
  }
  return make_float2(x, y);
}

__device__
unsigned char clip(float x){ return x > 255 ? 255 : (x < 0 ? 0 : x); }

// kernel function to compute decay and shading
__global__
void stabImageKernel(uchar4 *d_out, int w, int h, float p, int s) {
  const int c = blockIdx.x*blockDim.x + threadIdx.x;
  const int r = blockIdx.y*blockDim.y + threadIdx.y;
  if ((c >= w) || (r >= h)) return; // Check if within image bounds
```

```
43   const int i = c + r*w; // 1D indexing
44   const float x0 = scale(c, w);
45   const float y0 = scale(r, h);
46   const float dist_0 = sqrt(x0*x0 + y0*y0);
47   const float2 pos = euler(x0, y0, TIME_STEP, FINAL_TIME, p, s);
48   const float dist_f = sqrt(pos.x*pos.x + pos.y*pos.y);
49   // assign colors based on distance from origin
50   const float dist_r = dist_f/dist_0;
51   d_out[i].x = clip(dist_r*255); // red ~ growth
52   d_out[i].y = ((c == w/2) || (r == h/2)) ? 255 : 0; // axes
53   d_out[i].z = clip((1/dist_r)*255); // blue ~ 1/growth
54   d_out[i].w = 255;
55 }
56
57 void kernelLauncher(uchar4 *d_out, int w, int h, float p, int s) {
58   const dim3 blockSize(TX, TY);
59   const dim3 gridSize = dim3((w + TX - 1)/TX, (h + TY - 1)/TY);
60   stabImageKernel<<<gridSize, blockSize>>>(d_out, w, h, p, s);
61 }
```

Here is a brief description of the code in `kernel.cu`. Lines 1–6 include `kernel.h` and define constant values for thread counts, the spatial scale factor, and the time step and time interval for the simulation. Lines 8–35 define new device functions that will be called by the kernel:

- `scale()` scales the pixel values onto the coordinate range [-`LEN`, `LEN`].
- `f()` gives the rate of change of the velocity. If you are interested in studying other 1DoF oscillators, you can edit this to correspond to your system of interest. In the sample code, three different versions are included corresponding to different values of the variable `sys`.
    - The default version with `sys = 0` is the damped linear oscillator discussed above.
    - Setting `sys = 1` corresponds to a linear oscillator with negative effective stiffness (which may seem odd at first, but that is exactly the case near the inverted position of a pendulum).
    - Setting `sys = 2` corresponds to a personal favorite, the van der Pol oscillator, which has a nonlinear damping term.
- `euler()` performs the simulation for a given initial state and returns a `float2` value corresponding to the location of the trajectory at the end of the simulation interval. (Note that the `float2` type allows us to bundle the position and velocity together into a single entity. The alternative approach, passing a pointer to memory allocated to store multiple values as we do to handle larger sets of output from kernel functions, is not needed in this case.)

Lines 34–35 define the same `clip()` function that we used in the `flashlight` app, and the definition of the new kernel, `stabImageKernel()`, starts on line 38. Note that arguments have been added for the damping parameter value, `p`, and the system specifier, `s`. The index computation and bounds checking in lines 40–43 is exactly as in `distanceKernel()` from the `flashlight` app. On lines 44–45 we introduce {`x0, y0`} as the scaled `float` coordinate values (which range from −`LEN` to `LEN`) corresponding to the pixel location and compute the initial distance, `dist_0`, from the equilibrium point at the origin. Line 47 calls `euler()` to perform the simulation with fixed time increment `TIME_STEP` over an interval of duration `FINAL_TIME` and return `pos`, the state the simulated trajectory has reached at the end of the simulation. Line 50 compares the final distance from the origin and to the initial distance. Lines 51–54 assign shading values based on the distance comparison with blue indicating decay toward equilibrium (a.k.a. a vote in favor of stability) and red indicating growth away from equilibrium (which vetoes other votes for stability). Line 52 uses the green channel to show the horizontal *x*-axis and the vertical *y*-axis which intersect at the equilibrium point.

Lines 57–61 define the revised wrapper function `kernelLauncher()` with the correct list of arguments and name of the kernel to be launched.

Listing 4.11 `stability/interactions.h`

```
#ifndef INTERACTIONS_H
#define INTERACTIONS_H
#define W 600
#define H 600
#define DELTA_P 0.1f
#define TITLE_STRING "Stability"
int sys = 0;
float param = 0.1f;
void keyboard(unsigned char key, int x, int y) {
  if (key == 27)  exit(0);
  if (key == '0') sys = 0;
  if (key == '1') sys = 1;
  if (key == '2') sys = 2;
  glutPostRedisplay();
}

void handleSpecialKeypress(int key, int x, int y) {
  if (key == GLUT_KEY_DOWN) param -= DELTA_P;
  if (key == GLUT_KEY_UP)   param += DELTA_P;
  glutPostRedisplay();
}

// no mouse interactions implemented for this app
void mouseMove(int x, int y) { return; }
void mouseDrag(int x, int y) { return; }

```

```
27 void printInstructions() {
28   printf("Stability visualizer\n");
29   printf("Use number keys to select system:\n");
30   printf("\t0: linear oscillator: positive stiffness\n");
31   printf("\t1: linear oscillator: negative stiffness\n");
32   printf("\t2: van der Pol oscillator: nonlinear damping\n");
33   printf("up/down arrow keys adjust parameter value\n\n");
34   printf("Choose the van der Pol (sys=2)\n");
35   printf("Keep up arrow key depressed and watch the show.\n");
36 }
37 
38 #endif
```

The description of the alterations to `interactions.h`, as shown in Listing 4.11, is also straightforward. To the `#define` statements that set the width `W` and height `H` of the image, we add `DELTA_P` for the size of parameter value increments. Lines 7–8 initialize variables for the system identifier `sys` and the parameter value `param`, which is for adjusting the damping value.

There are a few `keyboard` interactions: Pressing `Esc` exits the app; pressing number key `0`, `1`, or `2` selects the system to simulate; and the up arrow and down arrow keys decrease or increase the damping parameter value by `DELTA_P`. There are no planned mouse interactions, so `mouseMove()` and `mouseDrag()` simply `return` without doing anything.

Finally, there are a couple details to take care of in other files:

- `kernel.h` contains the prototype for `kernelLauncher()`, so the first line of the function definition from `kernel.cu` should be copied and pasted (with a colon terminator) in place of the old prototype in `flashlight/kernel.h`.
- A couple small changes are also needed in `main.cpp`:
  - The argument list for the `kernelLauncher()` call in `render()` has changed, and that call needs to be changed to match the syntax of the revised kernel.
  - `render()` is also an appropriate place for specifying information to be displayed in the title bar of the graphics window. For example, the sample code displays an application name ("Stability") followed by the values of `param` and `sys`. Listing 4.12 shows the updated version of `render()` with the title bar information and updated kernel launch call.

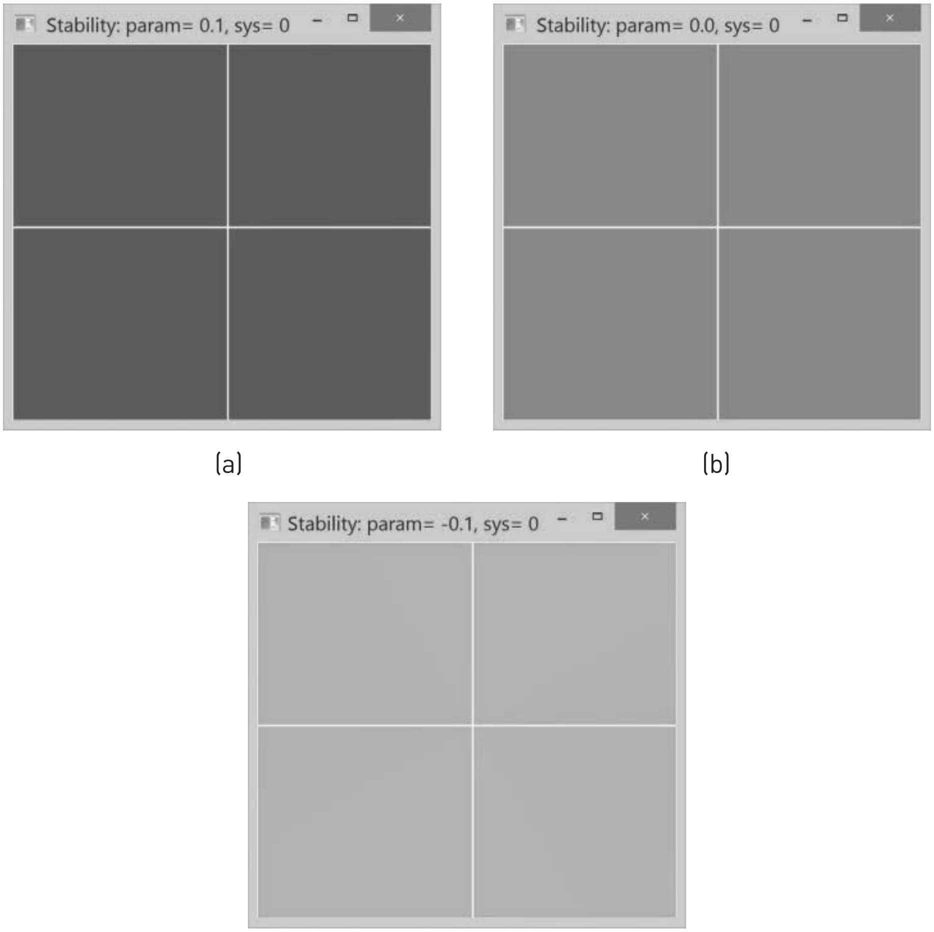
Listing 4.12 Updated `render()` function for `stability/main.cpp`

```
void render() {
  uchar4 *d_out = 0;
  cudaGraphicsMapResources(1, &cuda_pbo_resource, 0);
  cudaGraphicsResourceGetMappedPointer((void **)&d_out, NULL,
                                       cuda_pbo_resource);
  kernelLauncher(d_out, W, H, param, sys);
  cudaGraphicsUnmapResources(1, &cuda_pbo_resource, 0);
  // update contents of the title bar
  char title[64];
  sprintf(title, "Stability: param = %.1f, sys = %d", param, sys);
  glutSetWindowTitle(title);
}
```

RUNNING THE STABILITY VISUALIZER

Now that we've toured the relevant code, it is time to test out the app. In Linux, the Makefile for building this project is the same as the Makefile for the `flashlight` app that was provided in Listing 4.9. In Visual Studio, the included library files and the project settings are the same as described in `flashlight`. When you build and run the application, two windows should open: the usual command window showing a brief summary of supported user inputs and a graphics window showing the stability results. The default settings specify the linear oscillator with positive damping, which you can verify from the title bar that displays `Stability: param = 0.1, sys = 0`, as shown in Figure 4.4(a). Since all solutions of an unforced, damped linear oscillator are attracted toward the equilibrium, the graphics window should show the coordinate axes on a dark field, indicating stability. Next you might test the down arrow key. A single press reduces the damping value from 0.1 to 0.0 (which you should be able to verify in the title bar), and you should see the field changes from dark to moderately bright, as shown in Figure 4.4(b). The linear oscillator with zero damping is neutrally stable (with sinusoidal oscillations that remain near, but do not approach, the equilibrium). The explicit Euler ODE solver happens to produce small errors that systematically favor repulsion from the origin, but the color scheme correctly indicates that all initial states lead to solutions that roughly maintain their distance from the equilibrium. Another press of the down arrow key changes the damping parameter value to –0.1, and the bright field shown in Figure 4.4(c) legitimately indicates instability.

Now press the `1` key to set `sys = 1` corresponding to a system with negative effective stiffness, and increase the damping value. You should now see the axes



(a) (b)

(c)

Figure 4.4 Stability visualization for the linear oscillator with different damping parameter values. (a) For `param = 0.1`, the dark field indicates solutions attracted to a stable equilibrium. (b) For `param = 0.0`, the moderately bright field indicates neutral stability. (c) For `param = -0.1`, the bright field indicates solutions repelled from an unstable equilibrium.

on a bright field with a dark sector (and moderately bright transition regions), as shown in Figure 4.5. In this case, some solutions are approaching the equilibrium, but almost all initial conditions lead to solutions that grow away from the equilibrium, which is unstable.

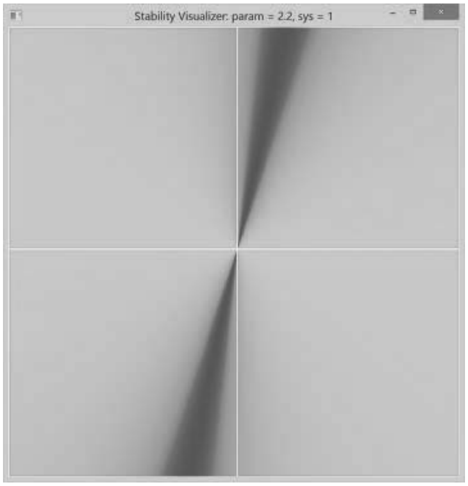


Figure 4.5 Phase plane of a linear oscillator with negative stiffness. A dark sector appears, but the bright field indicates growth away from an unstable equilibrium.

Setting the damping `param = 0.0` and `sys = 2` brings us to the final case in the example, the van der Pol oscillator. With `param = 0.0`, this system is identical to the undamped linear oscillator, so we again see the equilibrium in a moderately bright field. What happens when you press the up arrow key to make the damping positive? The equilibrium is surrounded by a bright region, so nearby initial states produce solutions that are repelled and the equilibrium is unstable. However, the outer region is dark, so initial states further out produce solutions that are attracted inwards. There is no other equilibrium point to go to, so where do all these solutions end up? It turns out that there is a closed, attracting loop near the shading transition corresponding to a stable period motion or "limit cycle" (Figure 4.6).

Note that the results of this type of numerical stability analysis should be considered as advisory. The ODE solver is approximate, and we *only* test a few hundred thousand initial states, so it is highly likely but not guaranteed that we did not miss something.

Before we are done, you might want to press and hold the up arrow key and watch the hundreds of thousands of pixels in the stability visualization change in real time. This is something you are not likely to be able to do without the power of parallel computing.

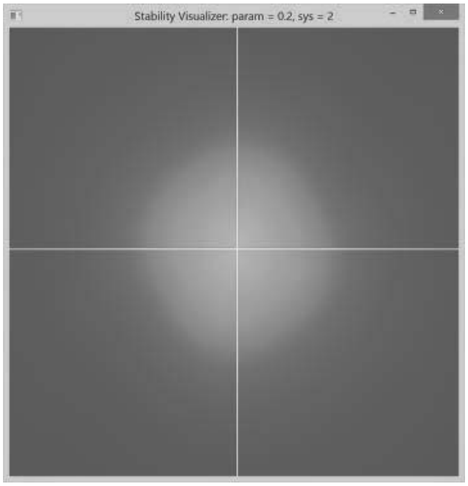


Figure 4.6 Phase plane of the van der Pol oscillator. The bright central region indicates an unstable equilibrium. The dark outer region indicates solutions decaying inwards. These results are consistent with the existence of a stable periodic “limit cycle” trajectory in the moderately bright region.

Summary

In this chapter, we covered the essentials of defining and launching kernels on 2D computational grids. We presented and explained sample code, the `flashlight` app that takes advantage of CUDA/OpenGL interop to implement real-time graphical display and interaction with the results from 2D computational grids. Finally, we showed how to use `flashlight` as a template and perform modifications to make it applicable to a real engineering problem, numerical exploration of dynamic stability.

Suggested Projects

1. Modify the `flashlight` app to be a version of the “hotter/colder” game. Provide an interface for player A to pick a target pixel. Player B then seeks out the target pixel based on the color of the spot, which turns blue (or red) as it is moved farther from (or closer to) the target.

2. Find another 1DoF system of interest and modify the `stability` app to study the nature of its equilibrium.
3. The explicit Euler method is perhaps the simplest and least reliable method for numerical solution of ODEs. Enhance the `stability` app by implementing a more sophisticated ODE solver. A Runge-Kutta method would be a good next step into a major field.
4. The van der Pol limit cycle turns out to be nearly circular for `param = 0.1`. Modify the `stability` app so the shading depends on the difference between the final distance and a new parameter `rad`. Implement interactive control of `rad`, and run the modified app to identify the size of the limit cycle.

References

[1] Microsoft Windows Dev Center. "Direct3D," 2015, https://msdn.microsoft.com/en-us/library/windows/desktop/hh309466(v=vs.85).aspx.

[2] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2, Third Edition.* (Reading, MA: Addison-Wesley, 1999).

[3] NVIDIA Corporation. "CUDA C Programming Guide," NVIDIA Developer Zone, CUDA Toolkit Documentation, 2015, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#abstract.

[4] Graham Sellers, Richard S. Wright, Jr., and Nicholas Haemel, *OpenGL Superbible: Comprehensive Tutorial and Reference, Seventh Edition.* (Boston, MA: Addison-Wesley, 2016).

[5] Randi J. Rost et al., *OpenGL Shading Language, Third Edition.* (Boston, MA: Addison-Wesley, 2010).

[6] cppreference.com, "ASCII Chart," 2015, http://en.cppreference.com/w/cpp/language/ascii.

[7] Richard H. Rand. *Lecture Notes on Nonlinear Vibrations*, Cornell University eCommons, May 2012, http://hdl.handle.net/1813/28989.

[8] Steven H. Strogatz, *Nonlinear Dynamics and Chaos, Second Edition.* (Cambridge, MA: Westview Press, 2014).

Index