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 Engineers 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
This page intentionally left blank
CUDA for Engineers
This page intentionally left blank
CUDA for Engineers

An Introduction to High-Performance Parallel Computing

Duane Storti
Mete Yurtoglu
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.
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.


Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.

First printing, November 2015
To the family, friends, and teachers who inspire us to keep learning cool things and to share what we have learned.
# Contents

Acknowledgments ........................................... xvii
About the Authors ......................................... xix

## Introduction ............................................. 1

What Is CUDA? ............................................. 1
What Does “Need-to-Know” Mean for Learning CUDA? .......... 2
What Is Meant by “for Engineers”? .......................... 3
What Do You Need to Get Started with CUDA? .............. 4
How Is This Book Structured? .............................. 4

## Conventions Used in This Book ......................... 8

## Code Used in This Book ................................. 8

## User’s Guide ............................................. 9

## Historical Context ...................................... 10

## References .............................................. 12

## Chapter 1: First Steps ................................. 13

Running CUDA Samples .................................. 13

CUDA Samples under Windows ............................ 14
CUDA Samples under Linux ............................... 17

Estimating “Acceleration” ............................... 17
## CONTENTS

Running Our Own Serial Apps .............................................. 19
  
  dist_v1 ..................................................................... 19
  
  dist_v2 ..................................................................... 20

Summary ............................................................................. 22
Suggested Projects ............................................................. 23

### Chapter 2: CUDA Essentials ............................................. 25

CUDA’s Model for Parallelism ................................................. 25

Need-to-Know CUDA API and C Language Extensions ............... 28
Summary ............................................................................. 31
Suggested Projects ............................................................. 31
References ........................................................................... 31

### Chapter 3: From Loops to Grids ......................................... 33

Parallelizing dist_v1 ............................................................ 33
  
  Executing dist_v1_cuda ...................................................... 37

Parallelizing dist_v2 ............................................................ 38

Standard Workflow ............................................................ 42

Simplified Workflow ........................................................... 43
  
  Unified Memory and Managed Arrays ................................. 43
  
  Distance App with cudaMallocManaged() ......................... 44

Summary ............................................................................. 47
Suggested Projects ............................................................. 48
References ........................................................................... 48

### Chapter 4: 2D Grids and Interactive Graphics ...................... 49

Launching 2D Computational Grids ........................................ 50
  
  Syntax for 2D Kernel Launch ............................................. 51
## Chapter 7: Interacting with 3D Data

- Launching 3D Computational Grids: dist_3d .................................................. 144
- Viewing and Interacting with 3D Data: vis_3d ............................................... 146
  - Slicing ........................................................................................................... 149
  - Volume Rendering ........................................................................................ 153
  - Raycasting .................................................................................................... 154
  - Creating the vis_3d App .............................................................................. 156
- Summary .......................................................................................................... 171
- Suggested Projects .......................................................................................... 171
- References ....................................................................................................... 171

## Chapter 8: Using CUDA Libraries

- Custom versus Off-the-Shelf .......................................................................... 173
- Thrust .................................................................................................................. 175
  - Computing Norms with inner_product() .................................................... 176
  - Computing Distances with transform() ...................................................... 180
  - Estimating Pi with generate(), transform(), and reduce() ..................... 185
- cuRAND .............................................................................................................. 190
- NPP ..................................................................................................................... 193
  - sharpen_npp ................................................................................................. 194
  - More Image Processing with NPP ................................................................. 198
- Linear Algebra Using cuSOLVER and cuBLAS ............................................. 201
- cuDNN ............................................................................................................... 207
- ArrayFire .......................................................................................................... 207
- Summary .......................................................................................................... 207
- Suggested Projects .......................................................................................... 208
- References ....................................................................................................... 209
Chapter 9: Exploring the CUDA Ecosystem .......... 211
The Go-To List of Primary Sources .................. 211
CUDA Zone ............................................. 211
Other Primary Web Sources .............................. 212
Online Courses ........................................... 213
CUDA Books ............................................. 214
Further Sources .......................................... 217
CUDA Samples .......................................... 217
CUDA Languages and Libraries ......................... 217
More CUDA Books ...................................... 217
Summary .................................................. 218
Suggested Projects ....................................... 219

Appendix A: Hardware Setup ......................... 221
Checking for an NVIDIA GPU: Windows .............. 221
Checking for an NVIDIA GPU: OS X .................... 222
Checking for an NVIDIA GPU: Linux .................... 223
Determining Compute Capability ......................... 223
Upgrading Compute Capability ............................ 225
Mac or Notebook Computer with a CUDA-Enabled GPU . 225
Desktop Computer ........................................ 226

Appendix B: Software Setup ......................... 229
Windows Setup ........................................... 229
Creating a Restore Point .................................. 230
Installing the IDE ........................................ 230
Installing the CUDA Toolkit ............................. 230
Initial Test Run ......................................... 235
Appendix D: CUDA Practicalities: Timing, Profiling, Error Handling, and Debugging  279

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Timing and Profiling</td>
<td>279</td>
</tr>
<tr>
<td>Standard C Timing Methods</td>
<td>280</td>
</tr>
<tr>
<td>CUDA Events</td>
<td>282</td>
</tr>
<tr>
<td>Profiling with NVIDIA Visual Profiler</td>
<td>284</td>
</tr>
<tr>
<td>Profiling in Nsight Visual Studio</td>
<td>288</td>
</tr>
<tr>
<td>Error Handling</td>
<td>292</td>
</tr>
<tr>
<td>Debugging in Windows</td>
<td>298</td>
</tr>
<tr>
<td>Debugging in Linux</td>
<td>305</td>
</tr>
<tr>
<td>CUDA-MEMCHECK</td>
<td>308</td>
</tr>
<tr>
<td>Using Visual Studio Property Pages</td>
<td>309</td>
</tr>
<tr>
<td>References</td>
<td>312</td>
</tr>
<tr>
<td>Index</td>
<td>313</td>
</tr>
</tbody>
</table>
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.
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 \texttt{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 \( W \times H \).

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 \( \text{blockDim.x} \), \( \text{blockIdx.x} \), and \( \text{threadIdx.x} \) according to the formula

\[
\text{int } i = \text{blockIdx.x}\times\text{blockDim.x} + \text{threadIdx.x};
\]

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:

\[
\text{int } c = \text{blockIdx.x}\times\text{blockDim.x} + \text{threadIdx.x};
\text{int } r = \text{blockIdx.y}\times\text{blockDim.y} + \text{threadIdx.y};
\]

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 \( \text{idx} \), \( \text{flatIdx} \), and \( \text{offset} \) 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:

\[
\text{int } i = r \times w + c;
\]

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 \texttt{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 \texttt{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 \texttt{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.

\section*{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:

\[
\text{dim3 blockSize(TX, TY);} \quad \text{// Equivalent to dim3 blockSize(TX, TY, 1);}
\]

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

\[
\text{int } bx = (W + \text{blockSize.x} - 1)/\text{blockSize.x} ;
\]
\[
\text{int } by = (H + \text{blockSize.y} - 1)/\text{blockSize.y} ;
\]

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

\[
\text{dim3 gridSize } = \text{dim3(bx, by)} ;
\]

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

\[
kernName<<\text{gridSize, blockSize}>>\text{(args)}
\]
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.

```c
1 __global__
2 void kernel2D(float *d_out, int w, int h, … )
3 {
4   // Compute column and row indices.
5   const int c = blockIdx.x * blockDim.x + threadIdx.x;
6   const int r = blockIdx.y * blockDim.y + threadIdx.y;
7   const int i = r * w + c; // 1D flat index
8
9   // Check if within image bounds.
10  if ((c >= w) || (r >= h))
11    return;
12
13  d_out[i] = INSERT_CODE_HERE; // Compute/store pixel in device array.
14 }
```

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

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

and the function call
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.

```c
#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.

```
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

```c
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   cudaFree(d_out);
36   free(out);
37   return 0;
38 }
```

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 \( \text{pos} \). (Note that we have defined \( \text{pos} \) to have type \( \text{float2} \) so it can store both coordinates of the reference location \( \{\text{pos.x, pos.y}\} \).) 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 \texttt{d\_out} and allocate the memory with \texttt{cudaMalloc()}. Line 25 sets the reference position, and lines 26–29 set the kernel launch parameters: a 2D grid of \texttt{bx} \times \texttt{by} blocks each having \texttt{TX} \times \texttt{TY} threads. Line 31 launches the kernel to compute the distance values, which are copied back to \texttt{out} on the host side on line 34. Lines 36–37 free the allocated device and host memory, then \texttt{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.

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

\begin{verbatim}
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 __device__
7 unsigned char clip(int n) { return n > 255 ? 255 : (n < 0 ? 0 : n); }
8
9 __global__
10 void distanceKernel(uchar4 *d_out, int w, int h, int2 pos)
11 {
12   const int c = blockIdx.x*blockDim.x + threadIdx.x;
13   const int r = blockIdx.y*blockDim.y + threadIdx.y;
14   const int i = r*w + c;
15   if ((c >= w) || (r >= h)) return;
16
17   // Compute the distance (in pixel spacings)
18   const int d = sqrtf((c - pos.x) * (c - pos.x) +
19                      (r - pos.y) * (r - pos.y));
20   // Convert distance to intensity value on interval [0, 255]
21   const unsigned char intensity = clip(255 - d);
22
23   d_out[i].x = intensity; // red channel
24   d_out[i].y = intensity; // green channel
25   d_out[i].z = 0; // blue channel
26   d_out[i].w = 255; // fully opaque
27 }
28
29 int main()
30 {
31   uchar4 *out = (uchar4*)malloc(W*H, sizeof(uchar4));
32   uchar4 *d_out; // pointer for device array
33   cudaMalloc(&d_out, W*H*sizeof(uchar4));
34
35   const int2 pos = {0, 0}; // set reference position
36   const dim3 blockSize(TX, TY);
37   const int bx = (W + TX - 1)/TX;
38   const int by = (W + TY - 1)/TY;
\end{verbatim}
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).
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.

*Figure 4.1 Interactive spot of light in the finished application*
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

```c
#include "kernel.h"
#include <stdio.h>
#include <stdlib.h>
#endif
#include <windows.h>
#endif
#include <GL/glew.h>
#include <GL/freeglut.h>
#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);
```
29    kernelLauncher(d_out, W, H, loc);
30    cudaGraphicsUnmapResources(1, &cuda_pbo_resource, 0);
31 }
32
33 void drawTexture() {
34    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, W, H, 0, GL_RGBA,
35        GL_UNSIGNED_BYTE, NULL);
36    glEnable(GL_TEXTURE_2D);
37    glBegin(GL_QUADS);
38    glTexCoord2f(0.0f, 0.0f); glVertex2f(0, 0);
39    glTexCoord2f(0.0f, 1.0f); glVertex2f(0, H);
40    glTexCoord2f(1.0f, 1.0f); glVertex2f(W, H);
41    glTexCoord2f(1.0f, 0.0f); glVertex2f(W, 0);
42    glEnd();
43    glDisable(GL_TEXTURE_2D);
44 }
45
46 void display() {
47    render();
48    drawTexture();
49    glutSwapBuffers();
50 }
51
52 void initGLUT(int *argc, char **argv) {
53    glutInit(argc, argv);
54    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
55    glutInitWindowSize(W, H);
56    glutCreateWindow(TITLE_STRING);
57    #ifndef __APPLE__
58    glewInit();
59    #endif
60 }
61
62 void initPixelBuffer() {
63    glGenBuffers(1, &pbo);
64    glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo);
65    glBufferData(GL_PIXEL_UNPACK_BUFFER, 4*W*H*sizeof(GLubyte), 0,
66        GL_STREAM_DRAW);
67    glGenTextures(1, &tex);
68    glBindTexture(GL_TEXTURE_2D, tex);
69    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
70    cudaGraphicsGLRegisterBuffer(&cuda_pbo_resource, pbo,
71        cudaMemcpyOrderingHostToDevice, cudaGraphicsMapFlagsWriteDiscard);
72 }
73
74 void exitfunc() {
75    if (pbo) {
76        cudaGraphicsUnregisterResource(cuda_pbo_resource);
77        glDeleteBuffers(1, &pbo);
78        glDeleteTextures(1, &tex);
79    }
80 }
81
82 int main(int argc, char** argv) {
83    printInstructions();
84}
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 \((W \times H)\), 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

```c
#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**

```c
1  #ifndef KERNEL_H
2  #define KERNEL_H
3
4  struct uchar4;
5  struct int2;
6
7  void kernelLauncher(uchar4 *d_out, int w, int h, int2 pos);
8
9  #endif
```

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

```c
1  #ifndef INTERACTIONS_H
2  #define INTERACTIONS_H
3  #define W 600
4  #define H 600
5  #define DELTA 5 // pixel increment for arrow keys
6  #define TITLE_STRING "flashlight: distance image display app"
7  int2 loc = {W/2, H/2};
8  bool dragMode = false; // mouse tracking mode
9
10  void keyboard(unsigned char key, int x, int y) {
11     if (key == 'a') dragMode = !dragMode; // toggle tracking mode
12     if (key == 27) exit(0);
13     glutPostRedisplay();
14 }
15
16  void mouseMove(int x, int y) {
17     if (dragMode) return;
18     loc.x = x;
19     loc.y = y;
20     glutPostRedisplay();
21 }
22
23  void mouseDrag(int x, int y) {
24     if (!dragMode) return;
25     loc.x = x;
26     loc.y = y;
27     glutPostRedisplay();
28 }
29
30  void handleSpecialKeypress(int key, int x, int y) {
31     if (key == GLUT_KEY_LEFT) loc.x -= DELTA;
32     if (key == GLUT_KEY_RIGHT) loc.x += DELTA;
33     if (key == GLUT_KEY_UP) loc.y -= DELTA;
34     if (key == GLUT_KEY_DOWN) loc.y += DELTA;
```
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 \(\text{dragMode}\) that is initialized to false. We use \(\text{dragMode}\) 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 \(\text{dragMode}\) to switch the mouse tracking mode.
  - The ASCII code 27 corresponds to the Esc key. Pressing Esc closes the graphics window.
  - \(\text{glutPostRedisplay}\) is called at the end of each callback function telling to compute a new image for display (by calling \(\text{display}\) in main.cpp) based on the interactive input.
- Lines 16–21 specify the response to a mouse movement. When \(\text{dragMode}\) is toggled, \(\text{return}\) 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 \(\text{glutPostRedisplay}\)).
• 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

```bash
1 UNAME_S := $(shell uname)
2 
3 ifeq ($(UNAME_S), Darwin)
4   LDFLAGS = -Xlinker -framework,OpenGL -Xlinker -framework,GLUT
5 else 
6   LDFLAGS += -L/usr/local/cuda/samples/common/lib/linux/x86_64
7   LDFLAGS += -lglut -lGL -lGLU -lGLEW
8 endif 
9 
10 NVCC = /usr/local/cuda/bin/nvcc
11 NVCC_FLAGS = -g -G -Xcompiler "-Wall -Wno-deprecated-declarations"
12 
13 all: main.exe
14```
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 $y$:

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

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 $t_k = k \times dt$, $x_k = x(t_k)$, 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:

$$
\begin{align*}
    x_{k+1} &= x_k + dt \cdot y_k \\
    y_{k+1} &= y_k + dt \cdot (-x_k - 2by_k)
\end{align*}
$$

We can then choose an initial state $(x_0, y_0)$ 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, $\text{dist}_r$ (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

```c
#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 \( \{x_0, y_0\} \) as the scaled float coordinate values (which range from \(-LEN\) to \(LEN\)) corresponding to the pixel location and compute the initial distance, \( \text{dist}_0 \), from the equilibrium point at the origin. Line 47 calls `euler()` to perform the simulation with fixed time increment \( \text{TIME\_STEP} \) over an interval of duration \( \text{FINAL\_TIME} \) and return \( \text{pos} \), 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**

```c
1 #ifndef INTERACTIONS_H
2 #define INTERACTIONS_H
3 #define W 600
4 #define H 600
5 #define DELTA_P 0.1f
6 #define TITLE_STRING "Stability"
7 int sys = 0;
8 float param = 0.1f;
9 void keyboard(unsigned char key, int x, int y) { 
10   if (key == 27)  exit(0);
11   if (key == '0') sys = 0;
12   if (key == '1') sys = 1;
13   if (key == '2') sys = 2;
14   glutPostRedisplay();
15 }
16
17 void handleSpecialKeypress(int key, int x, int y) { 
18   if (key == GLUT_KEY_DOWN) param -= DELTA_P;
19   if (key == GLUT_KEY_UP)   param += DELTA_P;
20   glutPostRedisplay();
21 }
22
23 // no mouse interactions implemented for this app
24 void mouseMove(int x, int y) { return; }
25 void mouseDrag(int x, int y) { return; }
26```
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 $\text{DELTA}_P$ for the size of parameter value increments. Lines 7–8 initialize variables for the system identifier $\text{sys}$ and the parameter value $\text{param}$, 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 $\text{DELTA}_P$. 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 $\text{param}$ and $\text{sys}$. 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

```c
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);
    // 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
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.
Setting the damping \( \text{param} = 0.0 \) and \( \text{sys} = 2 \) brings us to the final case in the example, the van der Pol oscillator. With \( \text{param} = 0.0 \), 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.
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 $\text{param} = 0.1$. Modify the stability app so the shading depends on the difference between the final distance and a new parameter $\text{rad}$. Implement interactive control of $\text{rad}$, and run the modified app to identify the size of the limit cycle.

References


Symbols
/ (slash), syntax of C language, 248
_ (underscore), variable names and, 249
# (number or hash sign), syntax of C language, 247
() (parentheses), syntax of C language, 247
* (asterisk)
    as dereference operator, 273
    syntax of C language, 248
[ ] (brackets), syntax of C language, 247
{ } (braces), syntax of C language, 247
<<, >> (double angle brackets or chevrons), bit shift operators, 186
<<<<, >>>> (triple angle brackets or chevrons), in syntax for parallel computation, 28

Numbers
1D grids, computing derivatives on, 81–82
1DoF, 66
2D data
    locating centroid of state map. See centroid_2d app
    solving Laplace’s equation in 2D (heat_2d), 88–92
2D grids
    defining 2D kernel, 52–53
dist_2d app, 53–56
launching, 50–51
moving to 3D grids, 144
overview of, 49
    syntax for 2D kernel launch, 51
3D data
    creating vis_3d app, 156
    generating relocatable device code, 167–170
launching 3D computational grids, 144–145
    listing for dist_3d/kernel.cu, 145
    listing for vis_3d/device_funCs.cu, 164–167
    listing for vis_3d/device_funCs.cuh, 164
    listing for vis_3d/interactions.h, 159–161
    listing for vis_3d/kernel.cu, 162–163
    listing for vis_3d/kernel.h, 161
    listing for vis_3d/main.cpp, 157–259
    listing for vis_3d/Makefile, 168–169
    overview of, 141–143
    raycasting, 154–156
    slice visualization of, 149–153
    viewing and interacting with, 146–149
    volume rendering, 153
3D grids
    launching computational grids, 144–145
    steps in, 142
    uses of, 141
5-point stencil, 91

A
AABBs (axis-aligned bounding boxes), in ray-box intersection testing, 149
About CUDA, CUDA resources, 211
About This Mac, checking for NVIDIA GPU, 222–223
Acceleration, estimating, 17–19
Actions/activities, Nsight Visual Studio profiling, 289–290
Algebra, using cuSolver and cuBLAS, 201–207
ALU (Arithmetic logic unit), in CUDA model for parallelization, 26
Angle brackets or chevrons
    bit shift operators, 186
    in syntax for parallel computation, 28
APIs. See Application programming interfaces (APIs)
Apple computers. See Mac OS X systems
INDEX

Application programming interfaces (APIs)
creating CUDA apps, 25
functions of CUDA Runtime API, 30, 292
passing Thrust device vector as argument to
CUDA API function or kernel, 189–190
standard graphics, 49
Apps. See also by individual types
building in C programming, 251
building in Linux, 258–262
building in Windows, 252–258
estimating acceleration, 17–19
running, 13
running serial, 19–22
running under Linux systems, 17
running under Windows systems, 14–17
use in book structure, 4
Arguments
passing by value in C, 246
passing CUDA device array as argument to
Thrust function, 189–190
passing Thrust device vector as argument to
CUDA API function or kernel, 190
typed arguments, 250
Arithmetic logic unit (ALU), in CUDA model for
parallelization, 26
Arithmetic operations
compound or augmented assignment, 250
in syntax of C language, 248
variable names and, 249
ArrayFire library, 207
Arrays
coefficient array. See Stencils
comparing serial and parallel computing, 25
costs and benefits of standard workflow, 42
declarations, 262–263
parallelizing dist_v2, 38–42
passing CUDA device array as argument to
Thrust function, 189–190
producing 2D array of distances from
reference point, 53–56
reduction, 121
running serial apps, 19–22
shared. See Shared arrays
unified memory and managed arrays, 43–47
using Thrust functions with, 178
Assignments
array declarations and, 263
in C programming, 248–250
compound or augmented, 250
Asterisk (*)
as dereference operator, 273
syntax of C language, 248
Asynchronous operations, execution timing and,
282
Atomic functions
available in CUDA, 129
dealing with race conditions, 128–129
specifying value of ATOMIC set, 124
use in centroid_2d app, 130
Axis-aligned bounding boxes (AABBs), in ray-box
intersection testing, 149

B
Basic Linear Algebra Subprograms (BLAS), 201
Bit shift operators (<<, >>), 186
BLAS (Basic Linear Algebra Subprograms), 201
blockDim, 29
blockIdx, 29
Blocks
about, 26
in 3D computational grid, 146
choosing execution configuration, 37
execution configuration for parallelizing
dist_v2, 41
execution configuration of kernel launch, 28
launching kernel and, 27
organizing threads by, 26
sharing information between, 79
syntax for 2D kernel launch, 51
Books, CUDA resources, 214–218
Bottlenecks, thread, 79
Bounding boxes, terminating loops, 155
Bounds checking, in 3D computational grid, 146
Braces { }, syntax of C language, 247
Brackets [ ], syntax of C language, 247
break statements, types of control statements
in C, 267
Breakpoints
setting for debugging, 306–308
setting when building apps, 256
Buffering
double buffering for efficiency of graphics
applications, 60
pixel buffers, 61
Builds
building apps in Linux, 258–262
building apps in Windows, 252–258
files in build process, 259–260

C
C programming
array declarations, 262–263
basics, 246–248
building apps (create, compile, run, debug), 251
building apps in Linux, 258–262
building apps in Windows, 252–258
characterizations of, 245–246
control statements, 263, 267
data types, declarations, and assignments, 248–250
defining functions, 250–251
dist_v1 sample, 267–270
dist_v2 sample, 271–275
dist_v2 sample with dynamic memory, 275–277
foundations for working with CUDA, 3
if statements, 265–266
invoking NVIDIA C Compiler, 37
for loops, 264–265
pointer arithmetic, 273
samples, 267
syntax, 247

C++
benefits of unified memory, 44
Clmg toolkit for, 102–104
CUDA development environment based on, 1
lambda functions, 183
syntax, 176
templates, 175
Cache, requirements for minimizing latency, 27
Callback functions
defined by interactions.h file, 57
keyboard and mouse, 95
vis_3d app, 159–161
calloc, creating host array out, 146
Case sensitivity
in C language, 249
capitalization of variable names, 52
Central processing units (CPUs), GPUs compared with, 26–27
centroid_2d app
centroid_2d/kernel.cu with Thrust, 188–189
computing integral properties, 130–131
listing for centroid_2d/kernel.cu, 131–133
listing for centroid_2d/kernel.h, 135
listing for centroid_2d/main.cpp, 135–136
listing for centroid_2d/Makefile, 136
output image with axes locating centroid, 136–137
Chevrons or angle brackets
bit shift operators, 186
in syntax for parallel computation, 28
Clmg toolkit
for C++, 102–104
importing/exporting data from image files, 174
clip(), utility functions of heat2d app, 96
clock(), timing methods in C, 280
Code
code libraries, 122
generating relocatable device code, 167–170
libraries. See Libraries
placing Thrust code in .cu file, 177
used in book, 8–9
Color, image processing using NPP, 198–199
Command-line interface
checking for NVIDIA GPU in Linux, 223
cconsole window, 15–17
Compilation, splitting kernel code into multiple files requiring separate compilation, 167–170
Compiled languages, C as, 245
Compiler
invoking from IDE, 245
invoking NVIDIA C Compiler, 37
Computational grids
2D. See 2D grids
3D. See 3D grids
Compute capability, 223–224
Computed tomography (CT), uses of 3D data grids, 141
Computer graphics, impact on surface appearance, 156
Computing integral properties, 130–131. See also centroid_2d app
Console window, accessing command-line interface, 15–17
Containers, in C++ standard library, 175–176
continue statements, types of control statements in C, 267
Control panel, checking for NVIDIA GPU in Windows, 221–222
Control statements, in C
  if statements, 265–266
  for loops, 264–265
  other options, 267
  overview of, 263
Conventions, used in book, 8
CPUs (central processing units), GPUs compared with, 26–27
CT (computed tomography), uses of 3D data grids, 141
.cu file, placing Thrust code in, 177
cuBLAS library, for linear algebra, 201–207
CUDA Deep Neural Network (CuDNN) library, 207
CUDA ecosystem
  books, 214–218
  CUDA languages and libraries, 217
  CUDA samples, 217
  CUDA Zone website, 211–212
  online courses, 213–214
  primary web sources, 212–213
CUDA events, 282–284
CUDA Random Number Generation (CuRAND) library, 190–192
CUDA Runtime API, 292
CUDA Toolkit
  about, 1
  install under Linux, 240
  install under OS X, 239
  install under Windows, 230
cudaDeviceSynchronize(), 30
cudaFree(), 30
cuda-gdb commands
  for debugging, 258, 261
dereference operator (*), 273
declarations
  array, 262–263
  C programming, 248–250
  “Deep copy” issues, benefits of unified memory for, 44
Deep Neural Networks, 207
debugging Linux systems, 305–308
CUDA-MEMCHECK tool, error handling with, 308–309
cuDNN (CUDA Deep Neural Network) library, 207
Data types
  C programming, 248–250
  CUDA supported, 30
  for images, 53
typed arguments and, 250
dd_1d_global app
  computing derivatives on 1D grid, 81–82
  implementing, 82
  listing for dd_1d_global/kernel.cu, 83
  listing for dd_1d_global/kernel.h, 84
  listing for dd_1d_global/main.cpp, 82–83
  listing for dd_1d_global/Makefile, 84
dd_1d_shared app, implementing, 85–88
Debugging. See also Error handling; Profiling
  building apps in Windows and, 254–258
  cublas and cuBLAS in, 207
  Debug execution in Visual Studio, 270
  limitations of Visual Studio or gdb, 38
  in Linux systems, 305–308
  viewing thrust vectors in debugger, 190
  in Windows systems, 298–304
Degrees of freedom (DOF)
  1DOF in stability app, 66–67
  velocity of 1DOF oscillators, 70
Densities, of 3D data values, 147
Depth, of stack, 144
Derivatives. See also dd_1d_global app
  computing derivatives of a function, 80
  computing on 1D grid, 81–82
  implementing dd_1d_global app, 82–85
  implementing dd_1d_shared app, 85–86
  solving Laplace’s equation in 2D, 88–92
Desktop computers, upgrading computing capability of GPU, 226–228
__device__ qualifier, function qualifiers, 29, 45
Device
about, 26
generating relocatable device code, 167–170
lambda, 183
passing CUDA device array as argument to Thrust function, 189–190
requirements for changing from serial to parallel computing, 25–26
transferring data between host and device, 44
volumetric medical scanners, 141
Devices, 3D data app
listing for vis_3d/device_funcs.cu, 164–167
listing for vis_3d/device_funcs.cuh, 164
Difference operator, finite
computing derivatives of a function, 80
computing derivatives on 1D grid, 81–82
Differential equations, 66–67
Differential equations, equilibrium temperature distribution, 88–92
Digital image processing, uses of 2D grids, 50
Digital light projection (DLP), uses of 3D data grids, 142
dim3 variable, CUDA supported vector types, 30
Dimension variables, kernel qualifiers, 29
Direct numerical simulation (DNS), in fluid dynamics, 141
Direct3D graphics APIs, 49
dist_1d app
computing distances, 180
for experimental lambda feature compatibility, 183–184
listing for dist_1d_thrust/kernel.cu, 180, 184–185
listing for dist_1d_thrust/kernel.cu with device lambda expressions, 183
listing for dist_1d_thrust/Makefile, 183
dist_2d app
computing distances on 2D grid, 54
parallel computation of image data based on distance from reference point in 2D grid, 55–56
producing 2D array of distances from reference point, 53–54
dist_3d app, parallel computation of 3D grid of distances from reference point, 145
dist_v1 app
applying for loop, 19–20
C programming sample, 267–270
executing dist_v1_cuda, 37–38
listing for dist_v1/main.cpp, 267–268
parallelizing, 33–37
dist_v1_cuda app
executing, 37–38
parallelizing dist_v1, 33–37
dist_v2 app
C programming sample, 271–275
creating array of input points, 19
dist_v2_cuda/Makefile for building app with error handling, 295
files in, 20–22
listing for dist_v1/main.cpp with dynamic memory management, 276
listing for dist_v2/aux_functions.cpp, 273
listing for dist_v2/aux_functions.h, 272
listing for dist_v2/main.cpp, 271
listing for dist_v2/Makefile, 274
listing for dist_v2_cuda/kernel.cu for CPU timing, 280–281
listing for dist_v2_cuda/kernel.cu modified for memory transfer, 283–284
modifying for error handling, 294–295
parallelizing, 38–42
sample with dynamic memory, 275–277
distanceArray() function
comparing serial and parallel computing, 25
parallelizing dist_v2, 41, 44
running serial apps, 19–22
Distances
2D array of distances from a reference point.
See dist_2d app
3D grid of distances from a reference point, 145
computing image distance using NPP, 199–200
computing using Thrust library, 180–185
interactive display of an image based on distance to a reference point. See flashlight app
parallel distance app. See dist_v2 app
serial distance app. See dist_v1 app
signed distance functions, 155
divUp()
utility function used with 2D grid, 96
utility function used with 3D grid, 144
INDEX

DLP (digital light projection), uses of 3D data grids, 142
DNS (direct numerical simulation), in fluid dynamics, 141
do statements, types of control statements in C, 267
Documentation, 212.  See also CUDA ecosystem
DOF (degrees of freedom)
1DOF in stability app, 66–67
velocity of 1DOF oscillators, 70
Dot product
computing norms with Thrust built-in functions, 176–180
parallelizing computation of.  See parallel_dot app
dotKernel(), dot product kernel, 124
dotLauncher(), dot product wrapper function, 124
Double buffering, for efficiency of graphics applications, 60
Downloads, CUDA resources, 212
Dynamic memory, in serial distance app, 276

E
Ecosystem.  See CUDA ecosystem
Education & Training, CUDA resources, 212–214
Element-wise operations, ease of parallelization, 122
else clause, using with if statements, 266
Embarrassingly parallel, threads with no interactions between, 122
Engineers, value of CUDA for, 3
Equilibrium.  See also heat_2d app
adding stability app to flashlight app, 67–68
solving Laplace’s equation in 2D, 88–92
Error handling.  See also Debugging; Profiling
building distance app with error handling, 295
with CUDA-MEMCHECK tool, 308–309
cuSolver and cuBLAS for, 207
modifying dist_v2_cuda/kernel.cu for, 294–295
overview of, 292–294
Errors, profiling to test for, 112–117
Events, CUDA events, 282–284
Executables, locating in bin folder, 14–15
Execution configuration
choosing for best performance, 37
of kernel launch, 28
for parallelizing dist_v2, 41
specifying computational stream number, 87
Execution timing
CUDA events and, 282–284
overview of, 279–280
standard methods in C, 280–282

F
Factorization, solving with cuSolver, 202–203
Files
in build process, 259–260
in dist_v2 app, 20–22
in flashlight app, 57
importing/exporting data from image files, 174
library files, 66
makefiles.  See Makefiles
Filtering, 79.  See also Stencils
Finite difference operator
computing derivatives of a function, 80
computing derivatives on 1D grid, 81–82
flashlight app
adding stability visualizer to.  See stability app
explanation of functionality of flashlight/interactions.h, 64–65
explanation of functionality of flashlight/main.cpp, 60–61
files in, 57
interactive display of an image based on distance to a reference point, 57
listing for flashlight/interactions.h, 63–64
listing for flashlight/kernel.cu, 62
listing for flashlight/kernel.h, 63
listing for flashlight/main.cpp, 58–60
listing for flashlight/Makefile, 65–66
as template, 58, 66
flatten(), utility functions of heat2d app, 97
Floating point operations per second (FLOPS), 1
Floating-point numbers, associative nature of floating-point addition in reduction, 130
Floating-point units (FPUs), in CUDA model for parallelization, 26
FLOPS (floating point operations per second), 1
for loops
control statements in C, 264–265
in dist_v1 app, 19–20
scale() qualifier, 45
syntax of, 264
FPUs (floating-point units), in CUDA model for parallelization, 26
Function calls
  kernel launch similar to, 28
  specifying input value of, 52
Function objects (functor), in Thrust transform(), 182
Function qualifiers, 28
Functions
  of CUDA Runtime API, 30
  defining in C programming, 250–251
  overloading, 175
  parallel functions in NPP, 193
  passing CUDA device array as argument to Thrust function, 189–190
  syntax, 250–251
  type qualifiers, 28
Functor (function objects), in Thrust transform(), 182
Fusion, 184
G
gdb
cuda-gdb commands, 258, 261, 305–308
  limitations of debugging tools, 38
GeForce, CUDA-enabled GPUs, 223
General purpose GPU (GPGPU), advantages of parallelism and, 25
generate(), estimating Pi using built-in Thrust functionality, 185–189
Getting Started, CUDA resources, 212
GLEW (OpenGL Extension Wrangler), 66
Global index, referencing global memory, 85
Global memory
  allocating arrays in device memory, 80–81
  implementing dd_1d_global app, 82–85
  profiling, 113
  slow access speed of, 81
  transferring data to shared memory, 87
__global__ qualifier
  defining 2D kernel, 52
  kernel qualifiers, 28
GLUT. See OpenGL Utility Toolkit (GLUT)
gmake, building apps in Linux, 258
goto statements, types of control statements in C, 267
GPGPU (General Purpose GPU), advantages of parallelism and, 25
Graphical processing units (GPUs)
  checking for NVIDIA GPU in Linux, 223
  checking for NVIDIA GPU in OS X, 222–223
  checking for NVIDIA GPU in Windows, 221–222
  CPUs compared with, 26–27
  CUDA-enabled parallel processing, 1–2
determining computing capability of, 223–225
  Kepler GPUs, 133–134
  memory areas of, 81
  upgrading computing capability of, 225–228
Graphics, impact on surface appearance, 156
Graphics interoperability
  adding stability app to flashlight app, 66–69
  explanation of functionality of flashlight/interactions.h, 64–65
  explanation of functionality of flashlight/main.cpp, 60–61
  explanation of functionality of stability/interactions.h, 72–73
  explanation of functionality of stability/kernel.cu, 70–71
files in flashlight app sample, 57
dist_v2, 41
listing for flashlight/interactions.h, 63–64
listing for flashlight/kernel.cu, 62
listing for flashlight/kernel.h, 63
listing for flashlight/main.cpp, 58–60
listing for flashlight/Makefile, 65–66
listing for stability/interactions.h, 71–72
listing for stability/kernel.cu, 69–70
OpenGL graphic package for, 56–57
running the stability visualizer app, 73–76
using flashlight app as template, 58, 66
gridDim, 29
Grids
  2D. See 2D grids
  3D. See 3D grids
choosing execution configuration, 37
computational grid in kernel launch, 27
computing derivatives on 1D grid, 81–82
converting loops to. See Loops, converting to grids
execution configuration for parallelizing dist_v2, 41
execution configuration of kernel launch, 28
INDEX

H
Halo cells
obtaining and storing values of, 87–88
use with shared array, 85
Hardware requirements, for changing from serial to parallel computing, 25–26
Hardware setup
checking for NVIDIA GPU in Linux, 223
checking for NVIDIA GPU in OS X, 222–223
determining computing capability of GPU, 223–225
upgrading computing capability of GPU, 225–228
Harris, Mark, 133
heat_2d app
listing for heat_2d/interactions.h, 95–96
listing for heat_2d/kernel.h, 94
listing for heat_2d/main.cpp, 92–94
solving for equilibrium temperature distribution in region with specified temperature, 88–89
solving Laplace’s equation in 2D, 89–92
Header guard. See Include guard
Heterogeneous Parallel Programming, online courses, 213–214
__host__ qualifier, function qualifiers, 28
Host
about, 26
transferring data between host and device, 44
I
IDE (Integrated Development Environment), invoking compiler from, 245
idxclip(), utility functions of heat2d app, 97
if statements
control statements in C, 265–266
syntax of, 265
Image files, importing/exporting data from, 174
Images
data types, 53
processing using NPP, 193, 198–201
sharpening image edges. See sharpen app
sharpening image edges using NPP. See sharpen_npp app
Include guard, 272
Index/indices
for 2D grids, 50–51
for 3D grids, 144
array declarations, 263
comparing serial and parallel computing, 38
in computational grids, 36, 41
global and local, 85
index variables of kernel, 29
for serial loops, 25–27
Inner products, applications of reduction. See parallel_dot app
inner_product(), Thrust built-in functions, 176–180
int type, 41
Integer arithmetic, in execution configuration for parallelizing dist_v2, 41
Integral properties, computing, 130–131. See also centroid_2d app
Integrated Development Environment (IDE), invoking compiler from, 245
Intensities, of pixel values (2D array), 147
Interactions, flashlight app
editing callback functions, 58
explanation of functionality of flashlight/interactions.h, 64–65
files in flashlight app, 57
listing for flashlight/interactions.h, 63–64
Interactions, heat_2d app, 95–96
Interactions, stability app
explanation of functionality of stability/interactions.h, 72–73
listing for stability/interactions.h, 71–72
Interactions, visualization app
listing for vis_3d/interactions.h, 159–161
viewing and interacting with 3D data, 146–149
Interactive graphics. See Graphics interoperability
Interop. See Graphics interoperability
Interpreted languages, compared with compiled, 246
Introductory section
benefits of CUDA for engineers, 3–4
book structure, 4–6
chapter dependencies, 7
code used, 8–9
copyrights used, 8
historical context, 10–12
need-to-know GPU-based parallel computing, 2–3
references, 10–12
user’s guide, 9–10
what is CUDA, 1–2
Iterators, C++ style, 176

J
Jacobi iteration
occurring between screen updates, 97
reduction and, 122
solving Laplace’s equation and, 91

K
Kepler GPUs, 133–134
Kernel capabilities and limits, 29
costs and benefits of standard workflow, 43
distanceKernel() qualifier, 45
launching, 28
mandatory return type is void, 36
passing Thrust device vector as argument to CUDA API function or kernel, 189–190
requirements for SIMT implement of parallelization, 27
splitting kernel code into multiple files, 167–170
stencil functions, 97–98
Kernel, 1D distance app
listing for dist_1d_thrust/kernel.cu, 180, 184
listing for dist_1d_thrust/kernel.cu with device lambda expressions, 183
Kernel, 2D distance app
defining 2D kernel, 52–53
syntax for 2D kernel launch, 51
Kernel, 3D visualization app
listing for vis_3d/kernel.cu, 162–163
listing for vis_3d/kernel.h, 161
Kernel, computing derivative on 1D grid
distances, 83
kernel.h, 84
Kernel, computing integral properties
listing for centroid_2d/kernel.cu, 131–133
listing for parallel_dot/kernel.h, 135
Kernel, graphics interop app
files in flashlight app, 57
Lambda expressions, passing functions and, 183–184
Lamberian radiance model, matte appearance and, 155
Languages. See also C programming
compiled, 245
CUDA ecosystem, 217
Interpreted, 246
Laplace’s equation, solving in 2D [heat_2d], 88–92
Laser Doppler velocimetry (LDV), uses of 3D data grids, 141
Latency, comparing CPUs with GPUs, 27
Launcher function. See Wrapper functions
LDV (Laser Doppler Velocimetry), uses of 3D data grids, 141
Least-squares problems, solving with cuSolver, 202
Libraries
 ArrayFire, 207
 computing distances using built-in Thrust functionality, 180–185
 computing norms using built-in Thrust functionality, 176–180
 CUDA ecosystem, 217
 cuDNN, 207
 cuRAND (CUDA Random Number Generation), 190–192
 cuSolver and cuBLAS, 201–207
 customization vs. off-the-shelf approach, 173–175
 documentation of Thrust library, 175–176
 estimating Pi using built-in Thrust functionality, 185–189
 image processing using NPP, 198–201
 linking library files, 193–194
 navigating NPP documentation, 197–198
 NVIDIA Performance Primitives (NPP), 193
 passing CUDA device array as argument to Thrust function, 189–190
 passing Thrust device vector as argument to CUDA API function or kernel, 190
 reduction and, 122
 Windows system requirements for flashlight app, 66
 Linear Algebra Package (LAPACK), 201–202
 Linear algebra, using cuSolver and cuBLAS, 201–207
 Linear oscillator. See Oscillators
 Linear regression, solving with cuSolver, 203–206
 Linking. See Libraries, linking library files
 Linux software setup
 initial test run, 242
 installing CUDA samples to user directory, 241–242
 installing CUDA toolkit, 240–241
 overview of, 240
 preparing for CUDA installation, 240
 Linux systems
 building apps using C, 258–262
 checking for NVIDIA GPU in Linux, 223
 converting dist_v1 to executable app, 269
 debugging in, 305–308
 invoking NVIDIA C Compiler, 37
 running CUDA samples under, 17
 viewing values in managed arrays, 46
 Local index, of items in shared array, 85
 Locals pane, 256
 Locals window. See Locals pane
 Loops
 for loops. See for loops
 termination by exiting bounding box, 155
 while loops, 155, 267
 Loops, converting to grids
 executing dist_v1_cuda, 37–38
 overview of, 33
 parallelizing dist_v1, 33–37
 simplified workflow, 43–47
 standard workflow, 42–43
 M
 Mac. See OS X
 main()
 building apps and, 251
 in heat_2d app, 91
 main.cpp
 dd_1d_global app, 82–83
 explanation of functionality of flashlight/
 main.cpp, 60–61
 files in flashlight app, 57
 listing for dist_v1/main.cpp, 267–268
 listing for dist_v2/main.cpp, 271
 listing for flashlight/main.cpp, 58–60
 listing for heat_2d/main.cpp, 92–94
 listing for parallel_dot/main.cpp, 126,
 135–136
 listing for sharpen_npp/main.cpp,
 194–195
 listing for sharpen/main.cpp, 103–104
 listing for vis_3d/main.cpp, 157–259
 make. See gmake
 Makefiles
 building Linux apps with, 37
 for compile/link process, 259–260
 for dd_1d_global, 84
 for dist_1d_thrust, 183
 for dist_v1, 19–20
 for dist_v1_cuda, 37
INDEX

for dist_v2, 22
for dist_v2_cuda, 39, 42
for flashlight, 65–66
for heat_2d, 100–102
for parallel_dot, 126–127, 136
for sharpen, 105
for sharpen_npp, 194
for stability, 73
for thrustpi, 192
malloc(), allocating host memory, 36
Managed arrays
allocating memory for, 44
unified memory and, 43–47
Mathematics, foundations for working with
CUDA, 3–4
Matlab, ArrayFire and, 207
Matrices
reduction and, 122
solving linear algebra problems with
cuSolver, 203
Memory
developments in memory access, 29
dynamic memory management, 276
kernel access to device memory not host
memory, 36
memory allocation in array declarations, 262
“nearer is faster” principle in data storage,
80–81
overhead of parallelizing, 42
shared memory. See Shared memory
unified memory and managed arrays, 43–47
Monte Carlo estimation, of Pi, 185–189
Mouse
callback function, 95
specifying interactions for flashlight app,
60

N
Namespaces, in resolution of naming conflicts,
176
Naming conventions, variables, 52
nbody sample
estimating acceleration, 17–18
running under Linux systems, 17
running under Windows systems, 14–17
“Nearer is faster” principle in data storage,
80–81
Neural networks, 207
Norms, computing using functions in Thrust
library, 176–180
Notation, C++, 176
Notebook computers, upgrading computing
capability of GPU, 225–226
NPP [NVIDIA Performance Primitives] library
image processing using, 198–201
linking to NPP files, 193–197
navigating NPP documentation, 197–198
overview of, 193
Nsight
debugging Linux systems, 305–308
debugging Windows systems, 298–304
profiling with Nsight Visual Studio, 288–292
requirements for running CUDA samples
under Windows systems, 14
Number or hash sign (#), syntax of C language,
247
nvcc
building apps in Linux, 258
handling uchar4 variables, 94
invoking NVIDIA C Compiler, 37
NVIDIA
checking for NVIDIA GPU in Linux, 223
checking for NVIDIA GPU in OS X, 222–223
checking for NVIDIA GPU in Windows, 221–222
debugging Linux systems, 305–308
debugging Windows systems, 298–304
DIGITS, 207
invoking NVIDIA C Compiler, 37
NVIDIA Performance Primitives [NPP] library
image processing using, 198–201
linking to NPP files, 193–197
navigating NPP documentation, 197–198
overview of, 193
NVIDIA Visual Profiler [NVVP]
cross platform visual profiling tool, 284–288
profiling to test for errors, 112–117
NVS, CUDA-enabled GPUs, 223
Nx acceleration, estimating acceleration, 17–18

O
ODEs [Ordinary Differential Equations], 66–67
Online courses, CUDA resources, 213–214
OpenGL Extension Wrangler [GLEW], 66
OpenGL graphics package
standard graphics APIs, 49
use in graphics interoperability app, 56–57
OpenGL Utility Toolkit (GLUT)
  explanation of functionality of flashlight/
  main.cpp, 60–61
  in heat_2d app, 92
  use in graphics interoperability app, 56–57
Windows system requirements for
  flashlight app, 66
Operating systems
  Linux. See Linux systems
  Mac OS X. See OS X systems
  Windows. See Windows systems
Optimus, 225
Order of execution, functions for synchronizing
  and coordinating, 30
Ordinary differential equations (ODEs), 66–67
Orthographic projection, as viewing transform, 60
Oscillators
  adding stability app to flashlight app,
  66–69
  numerical simulations of dynamics of, 49
  running stability visualizer, 74–76
  velocity of 1DOF oscillators, 70
Overloading, function, 175
OS X hardware setup
  checking for NVIDIA GPU in OS X, 222–223
  determining computing capability of GPU,
  223–225
  upgrading computing capability of GPU,
  225–226
OS X software setup
  installing CUDA toolkit, 239–240
  overview of, 238–239
OS X systems, CUDA gdb deprecated on, 305
P
Parallel computing
  basic tasks, 28–30
  comparing control of order of execution with
  serial computing, 38
  CUDA model for, 1–2, 25–27
  hardware setup for. See Hardware setup
  threads in, 148
Parallel Forall, CUDA resources, 134, 213
Parallel functions, in NPP, 193
Parallel patterns, uses of libraries, 174
parallel_dot app
  associative nature of floating-point addition,
  130
  atomics as solution to race conditions,
  128–129
  building and executing, 127–128
  listing for parallel_dot/kernel.cu,
  124–125
  listing for parallel_dot/kernel.h, 125
  listing for parallel_dot/main.cpp, 126
  listing for parallel_dot/Makefile,
  126–127
  parallelizing computation of the dot product of
  pair of vectors, 123–124
  parentheses ( ), syntax of C language, 247
  Passing by reference, arguments, 246
  Passing by value, arguments, 246
  Patterns, uses of parallel patterns, 174
  PCI, 226–228
Performance
  choosing execution configuration, 37
  costs of atomic functions, 129
Pi
  estimating using cuRAND, 190–192
  estimating using Thrust library, 185–189
Pixels [picture elements]
  buffers, 61
  describing pixel values via RGBA, 53–56
  in digital raster images, 50
  image processing using NPP, 199
  locating centroid, 130, 137
  scaling pixel values, 70
Placeholders, in Thrust transform(), 182
Pointer arithmetic, C programming, 273
Pointers, use in C, 246
pos, as reference position, 146
Preprocessors, syntax of C language, 247
Primitive data types, in C language, 248
printf() statement, support for using from a
  kernel, 37–38
Profiling. See also Debugging; Error handling
  with cuSolver and cuBLAS, 207
  with Nsight Visual Studio, 288–292
  with NVIDIA visual profiler, 284–288
  to test for errors, 112–117
Programming Massively Parallel Processors
  with CUDA, online courses, 214
Property Pages, Visual Studio, 309–311
Q
Quadro, CUDA-enabled GPUs, 223
INDEX

R
Race conditions, resulting in undefined behavior, 128
Radar systems, uses of 3D data grids, 141
Radius, stencil, 85
Random number generation, 190–192
Raster images, uses of 2D grids, 50
Ray tracing, impact on surface appearance, 156
Ray-box intersection tests, 148–149
Raycasting
  3D data, 154–156
  overview of, 147
  sample images of 3D distance field, 143
  screenshot of vis_3d raycast visualization, 170
rayCastShader(), 153
Ray-plane intersection tests, 152
Read-modify-write sequence, atomic functions performing, 128–129
reduce(), estimating Pi using built-in Thrust functionality, 185–189
Reduction
  associative nature of floating-point addition, 130
  atomics as solution to race conditions, 128–129
  computing dot product of pair of vectors, 123–124
  computing integral properties, 130–131
  listing for centroid_2d/kernel.cu, 131–133
  listing for centroid_2d/kernel.h, 135
  listing for centroid_2d/main.cpp, 135–136
  listing for centroid_2d/Makefile, 136
  listing for parallel_dot/kernel.cu, 124–125
  listing for parallel_dot/kernel.h, 125
  listing for parallel_dot/main.cpp, 126
  listing for parallel_dot/Makefile, 126–127
  output image with axes locating centroid, 136–137
  overview of, 121
  threads interacting globally, 121–122
Redundancy, stencil radius and, 85
Regression, solving linear regression, 203–206
ResetKernel(), in heat2d app, 97
resetTemperature(), in heat2d app, 97
Resources. See CUDA ecosystem
RGB (red, green, blue) color model, image processing using NPP, 198–199
RGBA
  describing pixel values, 53–56
  initializing GLUT library and, 60
Rootfinding, locating point where function takes on given value, 155
Runtime API
  error handling and, 292
  functions of CUDA Runtime API, 30
S
Samples, CUDA resources, 217
Scalability, of SIMT approach, 26
Scalar products, applications of reduction. See parallel_dot app
Scoping, defining functions, 250
Seismic exploration, uses of 3D data grids, 141
Separate compilation. See Relocatable device code
Serial computing
  comparing control of order of execution with parallel computing, 38
  comparing serial and parallel computing, 25
  running serial apps, 19–22
Serialization, costs of atomic functions, 129
Shading functions
  rayCastShader(), 153
  sliceShader(), 151–153
  volumeRenderShader(), 153
Shared arrays
  declaring, 86, 133
  index of items in, 85
  setting size of, 87
Shared memory
  bridging gap between memory access and speed, 81
  computing derivatives on 1D grid, 81–82
  declaring shared array, 133
  declaring threads as shared, 123
  image sharpening app, 103
  implementing dd_1d_global app, 82–85
  implementing dd_1d_shared app, 85
  profiling, 114
  sharing information between blocks, 79
Sharing models, impact on surface appearance, 156

sharpen app
  listing for sharpen/kernel.cu with global memory implementation, 105–107
  listing for sharpen/kernel.cu with input and output shared memory arrays, 109–112
  listing for sharpen/kernel.cu with shared memory implementation, 107–109
  listing for sharpen/main.cpp, 103–104
  listing for sharpen/Makefile, 105
  profiling to test for errors, 112–117
  sharpening image edges, 102–103

sharpen_npp app
  listing for sharpen_npp/main.cpp, 194–195
  listing for sharpen_npp/Makefile, 194
  reimplement of sharpen app using NPP, 194

Signal processing, NPP functions for, 193
Signed distance function, 155
Single instruction multiple thread (SIMT)
  limits on thread accessibility, 123
  model of parallelization, 26
Slash (/), syntax of C language, 248
sliceShader() function, 151–153

Slicing
  overview of, 147
  sample images of 3D distance field, 143
  screenshot of vis_3d slicing, 169
  slice visualization of 3D data, 149–153

SMs. See Streaming multiprocessors (SMs)

Software
  requirements for changing from serial to parallel computing, 25
  requirements for SIMT implement of parallelization, 27
  setup for Mac OS X. See Mac OS X software setup
  setup for Windows OSs. See Windows software setup

stability app
  explanation of functionality of stability/interactions.h, 71–72
  listing for stability/kernel.cu, 69–70
  running the stability visualizer app, 73–76
  stabilizing visualization app, 69–70

Stack (or stratum), integer variable used in indexing, 144

Stack overflow error, 146

StackOverflow, CUDA resources, 213

Standard Template Library (STL), Thrust based on, 175–176

State map, locating centroid on. See centroid_2d app

Steady-state temperature distribution. See also heat_2d app
  overview of, 91
  reduction and, 122

Stencils
  5-point stencil, 91
  computing derivatives on 1D grid, 81–82
    implementing dd_1d_global app, 82–85
    implementing dd_1d_shared app, 85–88
  kernel functions, 97–98
    listing for heat_2d/interactions.h, 95–96
    listing for heat_2d/kernel.cu, 98–100
    listing for heat_2d/kernel.h, 94
    listing for heat_2d/main.cpp, 92–94
    listing for heat_2d/Makefile, 100–102
  overview of, 79
  radius, 85
  sharpening image edges. See sharpen app
  solving Laplace’s equation in 2D [heat_2d], 88–92
  thread interdependence and, 80–81

STL (Standard Template Library), Thrust based on, 175–176

Stratum (or stack), integer variable used in indexing, 144

Streaming multiprocessors (SMs)
  choosing execution configuration, 37
  in CUDA model for parallelization, 26–27
  thread interdependences and, 80

Streams, specifying computational stream number, 87

Synchronous operations
  error handling and, 292
  execution timing and, 282
  __syncthreads(), synchronization of threads, 88
Temperature, equilibrium temperature distribution. See heat_2d app
Temperature distribution, equilibrium. See heat_2d app
tempKernel(), kernel functions of heat2d app

Templates
C++ template library, 175–176
flashlight app as, 58, 66
Tesla, CUDA-enabled GPUs, 223
Tests, for errors, 112–117. See also Profiling
Text editors, for Linux and OS X, 258
Threadblocks, launching kernel and, 27
threadIdx, 29
Thread
about, 26
bottlenecks, 79
breaking tasks into subtasks, 26
choosing execution configuration, 37
debugging Linux systems, 307
execution configuration for parallelizing dist_v2, 41
execution configuration of kernel launch, 28
interacting globally, 121–122
interdependence, 80–81
launching kernel and, 27
limits on accessibility, 123
in parallel computing, 148
synchronization of, 88
threads per block (TPB), defining constant values in parallel_dot app, 124

Thrust library
computing distances, 180–185
computing norms, 176–180
documentation of, 175–176
estimating Pi, 185–189
passing CUDA device array as argument to Thrust function, 189–190
passing Thrust device vector as argument to CUDA API function or kernel, 190
using with arrays, 178
thrustpi app, 186–188, 190–192
<time.h>, timing methods in C, 280
Timing. See Execution timing
Tools, CUDA resources, 212

TPB [threads per block], defining constant values in parallel_dot app, 124
Traces, Nsight Visual Studio profiling, 289–290
Training, CUDA resources, 212–214
transform()
computing distances using built-in Thrust functionality, 180–185
estimating Pi using built-in Thrust functionality, 185–189
Triggers, Nsight Visual Studio profiling, 289–290
Type qualifiers, functions, 28
Typed arguments, defining functions in C, 250
Typed languages, C as, 246

U
uchar4 variables, nvcc handling of, 94
Udacity CS344: Intro to Parallel Programming, online courses, 213
uint3 variable, CUDA supported vector types, 30
Underscore (_), variable names and, 249
Unified memory
creating a simplified workflow, 43–47
notes regarding use of, 44
Unsharp masks, 102–103
Upgrades, computing capability of GPU, 225–228

V
van der Pol oscillator
adding stability app to flashlight app, 66
phase plane of, 76
Variables
capitalization of variable names, 52
declaring and naming in C, 248–249
Vectors
container in C++ standard library, 175–176
CUDA supported data types, 30
parallelizing computation of the dot product of pair of. See parallel_dot app
passing Thrust device vector as argument to CUDA API function or kernel, 189–190
using Thrust functions with arrays instead of vectors, 178
viewing thrust vectors in debugger, 190
Video processing
NPP functions for, 193
uses of 3D data grids, 142
vis_3d app
creating, 156
listing for vis_3d/device_funcs.cu, 164–167
listing for vis_3d/device_funcs.cuh, 164
listing for vis_3d/interactions.h, 159–161
listing for vis_3d/kernel.cu, 162–163
listing for vis_3d/kernel.h, 161
listing for vis_3d/main.cpp, 157–259
listing for vis_3d/Makefile, 168–169
viewing and interacting with 3D data, 146–149
Visual Profiler. See NVIDIA Visual Profiler (NVVP)
Visual Studio
building apps in Windows, 252–258
Debug execution, 270
limitations of debugging tools, 38
messages and warnings, 275
Property Pages, 309–311
Visual Studio Community 2013, 14
Visualization of data
slice visualization of, 149–153
viewing and interacting with 3D data. See vis_3d app
void, mandatory return type for kernel, 36
Volume elements (voxels)
length scale set by voxel spacing, 155
moving from 2D grids to 3D grids, 144
Volume rendering
3D data, 153
overview of, 147
sample images of 3D distance field, 143
screenshot of vis_3d volume rendering, 170
volumeRenderShader(), 153
Voxels (volume elements)
length scale set by voxel spacing, 155
moving from 2D grids to 3D grids, 144
W
Warp, dividing blocks into, 26–27
Watch window, 277
Web resources
CUDA Zone website, 211–212
primary web sources, 212–213
while loops
termination by exiting bounding box, 155
types of control statements in C, 267
White spaces, in C language, 249
Wikipedia, CUDA resources, 213
Windows hardware setup
checking for NVIDIA GPU in Windows, 221–222
determining computing capability of GPU, 223–225
Windows software setup
creating restore point, 230
initial test run, 235–238
installing CUDA toolkit, 230–234
installing IDE, 230
steps in, 229
Windows systems
building apps using C, 252–258
converting dist_v1 to executable app, 269
debugging in, 298–304
invoking NVIDIA C Compiler, 37
running CUDA samples under, 14–17
viewing thrust vectors in debugger, 190
viewing values in managed arrays, 46–47
Workflow
analyzing costs and benefits of CUDA, 42–43
unified memory simplifying, 43–47
Wrapper functions
about, 41
in vis_3d app, 161