CUDA by Example

AN INTRODUCTION TO GENERAL-PURPOSE GPU PROGRAMMING

JASON SANDERS
EDWARD KANDROT

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

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

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

NVIDIA makes no warranty or representation that the techniques described herein are free from any Intellectual Property claims. The reader assumes all risk of any such claims based on his or her use of these techniques.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Sanders, Jason.
CUDA by example : an introduction to general-purpose GPU programming /
Jason Sanders, Edward Kandrot.
p. cm.
Includes index.
ISBN 978-0-13-138768-3 (pbk. : alk. paper)
1. Application software—Development. 2. Computer architecture. 3.
Parallel programming (Computer science) I. Kandrot, Edward. II. Title.
QA76.76.A65S255 2010
005.2'75—dc22
2010017618

Copyright © 2011 NVIDIA Corporation

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. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-13: 978-0-13-138768-3
ISBN-10: 0-13-138768-5
Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
First printing, July 2010

Contents

Foreword . . . xiii

Preface . . . xv

Acknowledgments . . . xvii

About the Authors . . . xix

1 WHY CUDA? WHY NOW? 1

1.1 Chapter Objectives . . . 2

1.2 The Age of Parallel Processing . . . 2

1.2.1 Central Processing Units . . . 2

1.3 The Rise of GPU Computing . . . 4

1.3.1 A Brief History of GPUs . . . 4

1.3.2 Early GPU Computing . . . 5

1.4 CUDA . . . 6

1.4.1 What Is the CUDA Architecture? . . . 7

1.4.2 Using the CUDA Architecture . . . 7

1.5 Applications of CUDA . . . 8

1.5.1 Medical Imaging . . . 8

1.5.2 Computational Fluid Dynamics . . . 9

1.5.3 Environmental Science . . . 10

1.6 Chapter Review . . . 11

2 GETTING STARTED 13

2.1 Chapter Objectives 14

2.2 Development Environment 14

2.2.1 CUDA-Enabled Graphics Processors 14

2.2.2 NVIDIA Device Driver 16

2.2.3 CUDA Development Toolkit 16

2.2.4 Standard C Compiler 18

2.3 Chapter Review 19

3 INTRODUCTION TO CUDA C 21

3.1 Chapter Objectives 22

3.2 A First Program 22

3.2.1 Hello, World! 22

3.2.2 A Kernel Call 23

3.2.3 Passing Parameters 24

3.3 Querying Devices 27

3.4 Using Device Properties 33

3.5 Chapter Review 35

4 PARALLEL PROGRAMMING IN CUDA C 37

4.1 Chapter Objectives 38

4.2 CUDA Parallel Programming 38

4.2.1 Summing Vectors 38

4.2.2 A Fun Example 46

4.3 Chapter Review 57

5 THREAD COOPERATION 59

5.1 Chapter Objectives 60

5.2 Splitting Parallel Blocks 60

5.2.1 Vector Sums: Redux 60

5.2.2 GPU Ripple Using Threads 69

5.3 Shared Memory and Synchronization 75

5.3.1 Dot Product 76

5.3.1 Dot Product Optimized (Incorrectly) 87

5.3.2 Shared Memory Bitmap 90

5.4 Chapter Review 94

6 CONSTANT MEMORY AND EVENTS 95

6.1 Chapter Objectives 96

6.2 Constant Memory 96

6.2.1 Ray Tracing Introduction 96

6.2.2 Ray Tracing on the GPU 98

6.2.3 Ray Tracing with Constant Memory 104

6.2.4 Performance with Constant Memory 106

6.3 Measuring Performance with Events 108

6.3.1 Measuring Ray Tracer Performance 110

6.4 Chapter Review 114

7 TEXTURE MEMORY 115

7.1 Chapter Objectives 116

7.2 Texture Memory Overview 116

7.3 Simulating Heat Transfer 117
7.3.1 Simple Heating Model 117
7.3.2 Computing Temperature Updates 119
7.3.1 Animating the Simulation 121
7.3.2 Using Texture Memory 125
7.3.3 Using Two-Dimensional Texture Memory 131
7.4 Chapter Review 137

8 GRAPHICS INTEROPERABILITY 139

8.1 Chapter Objectives 140
8.2 Graphics Interoperation 140
8.3 GPU Ripple with Graphics Interoperability 147
8.3.1 The GPUAnimBitmap Structure 148
8.3.2 GPU Ripple Redux 152
8.4 Heat Transfer with Graphics Interop 154
8.5 DirectX Interoperability 160
8.6 Chapter Review 161

9 ATOMICS 163

9.1 Chapter Objectives 164
9.2 Compute Capability 164
9.2.1 The Compute Capability of NVIDIA GPUs 164
9.2.2 Compiling for a Minimum Compute Capability 167
9.3 Atomic Operations Overview 168
9.4 Computing Histograms 170
9.4.1 CPU Histogram Computation 171
9.4.2 GPU Histogram Computation 173
9.5 Chapter Review 183

10 STREAMS 185

10.1 Chapter Objectives 186

10.2 Page-Locked Host Memory 186

10.3 CUDA Streams 192

10.4 Using a Single CUDA Stream 192

10.5 Using Multiple CUDA Streams 198

10.6 GPU Work Scheduling 205

10.7 Using Multiple CUDA Streams Effectively 208

10.8 Chapter Review 211

11 CUDA C ON MULTIPLE GPUS 213

11.1 Chapter Objectives 214

11.2 Zero-Copy Host Memory 214

11.2.1 Zero-Copy Dot Product 214

11.2.2 Zero-Copy Performance 222

11.3 Using Multiple GPUs 224

11.4 Portable Pinned Memory 230

11.5 Chapter Review 235

12 THE FINAL COUNTDOWN 237

12.1 Chapter Objectives 238

12.2 CUDA Tools 238

12.2.1 CUDA Toolkit 238

12.2.2 CUFFT 239

12.2.3 CUBLAS 239

12.2.4 NVIDIA GPU Computing SDK 240

12.2.5 NVIDIA Performance Primitives 241

12.2.6 Debugging CUDA C 241

12.2.7 CUDA Visual Profiler 243

12.3 Written Resources 244

12.3.1 Programming Massively Parallel Processors:
A Hands-On Approach 244

12.3.2 CUDA U 245

12.3.3 NVIDIA Forums 246

12.4 Code Resources 246

12.4.1 CUDA Data Parallel Primitives Library 247

12.4.2 CULAtools 247

12.4.3 Language Wrappers 247

12.5 Chapter Review 248

A ADVANCED ATOMICS 249

A.1 Dot Product Revisited 250

A.1.1 Atomic Locks 251

A.1.2 Dot Product Redux: Atomic Locks 254

A.2 Implementing a Hash Table 258

A.2.1 Hash Table Overview 259

A.2.2 A CPU Hash Table 261

A.2.3 Multithreaded Hash Table 267

A.2.4 A GPU Hash Table 268

A.2.5 Hash Table Performance 276

A.3 Appendix Review 277

Index 279

Preface

This book shows how, by harnessing the power of your computer's graphics process unit (GPU), you can write high-performance software for a wide range of applications. Although originally designed to render computer graphics on a monitor (and still used for this purpose), GPUs are increasingly being called upon for equally demanding programs in science, engineering, and finance, among other domains. We refer collectively to GPU programs that address problems in nongraphics domains as *general-purpose*. Happily, although you need to have some experience working in C or C++ to benefit from this book, you need not have any knowledge of computer graphics. None whatsoever! GPU programming simply offers you an opportunity to build—and to build mightily—on your existing programming skills.

To program NVIDIA GPUs to perform general-purpose computing tasks, you will want to know what CUDA is. NVIDIA GPUs are built on what's known as the *CUDA Architecture*. You can think of the CUDA Architecture as the scheme by which NVIDIA has built GPUs that can perform *both* traditional graphics-rendering tasks *and* general-purpose tasks. To program CUDA GPUs, we will be using a language known as *CUDA C*. As you will see very early in this book, CUDA C is essentially C with a handful of extensions to allow programming of massively parallel machines like NVIDIA GPUs.

We've geared *CUDA by Example* toward experienced C or C++ programmers who have enough familiarity with C such that they are comfortable reading and writing code in C. This book builds on your experience with C and intends to serve as an example-driven, "quick-start" guide to using NVIDIA's CUDA C programming language. By no means do you need to have done large-scale software architecture, to have written a C compiler or an operating system kernel, or to know all the ins and outs of the ANSI C standards. However, we do not spend time reviewing C syntax or common C library routines such as `malloc()` or `memcpy()`, so we will assume that you are already reasonably familiar with these topics.

You will encounter some techniques that can be considered general parallel programming paradigms, although this book does not aim to teach general parallel programming techniques. Also, while we will look at nearly every part of the CUDA API, this book does not serve as an extensive API reference nor will it go into gory detail about every tool that you can use to help develop your CUDA C software. Consequently, we highly recommend that this book be used in conjunction with NVIDIA's freely available documentation, in particular the *NVIDIA CUDA Programming Guide* and the *NVIDIA CUDA Best Practices Guide*. But don't stress out about collecting all these documents because we'll walk you through everything you need to do.

Without further ado, the world of programming NVIDIA GPUs with CUDA C awaits!

Chapter 4

Parallel Programming in CUDA C

In the previous chapter, we saw how simple it can be to write code that executes on the GPU. We have even gone so far as to learn how to add two numbers together, albeit just the numbers 2 and 7. Admittedly, that example was not immensely impressive, nor was it incredibly interesting. But we hope you are convinced that it is easy to get started with CUDA C and you're excited to learn more. Much of the promise of GPU computing lies in exploiting the massively parallel structure of many problems. In this vein, we intend to spend this chapter examining how to execute parallel code on the GPU using CUDA C.

4.1 Chapter Objectives

Through the course of this chapter, you will accomplish the following:

- You will learn one of the fundamental ways CUDA exposes its parallelism.
- You will write your first parallel code with CUDA C.

4.2 CUDA Parallel Programming

Previously, we saw how easy it was to get a standard C function to start running on a device. By adding the `__global__` qualifier to the function and by calling it using a special angle bracket syntax, we executed the function on our GPU. Although this was extremely simple, it was also extremely inefficient because NVIDIA's hardware engineering minions have optimized their graphics processors to perform hundreds of computations in parallel. However, thus far we have only ever launched a kernel that runs serially on the GPU. In this chapter, we see how straightforward it is to launch a device kernel that performs its computations in parallel.

4.2.1 SUMMING VECTORS

We will contrive a simple example to illustrate threads and how we use them to code with CUDA C. Imagine having two lists of numbers where we want to sum corresponding elements of each list and store the result in a third list. Figure 4.1 shows this process. If you have any background in linear algebra, you will recognize this operation as summing two vectors.

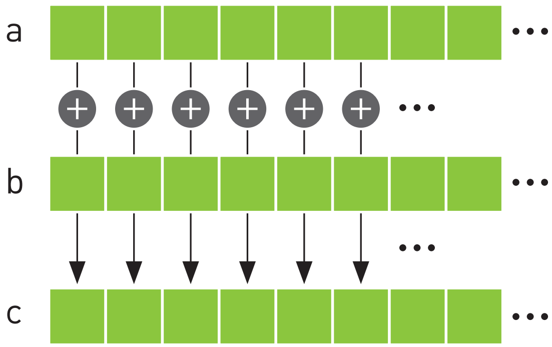

Figure 4.1 Summing two vectors

CPU VECTOR SUMS

First we'll look at one way this addition can be accomplished with traditional C code:

```
#include "../common/book.h"

#define N   10

void add( int *a, int *b, int *c ) {
    int tid = 0;    // this is CPU zero, so we start at zero
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1;   // we have one CPU, so we increment by one
    }
}

int main( void ) {
    int a[N], b[N], c[N];

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    add( a, b, c );
    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    return 0;
}
```

Most of this example bears almost no explanation, but we will briefly look at the `add()` function to explain why we overly complicated it.

```
void add( int *a, int *b, int *c ) {
    int tid = 0;    // this is CPU zero, so we start at zero
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1;   // we have one CPU, so we increment by one
    }
}
```

We compute the sum within a `while` loop where the index `tid` ranges from `0` to `N-1`. We add corresponding elements of `a[]` and `b[]`, placing the result in the corresponding element of `c[]`. One would typically code this in a slightly simpler manner, like so:

```
void add( int *a, int *b, int *c ) {
    for (i=0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
}
```

Our slightly more convoluted method was intended to suggest a potential way to parallelize the code on a system with multiple CPUs or CPU cores. For example, with a dual-core processor, one could change the increment to 2 and have one core initialize the loop with `tid = 0` and another with `tid = 1`. The first core would add the even-indexed elements, and the second core would add the odd-indexed elements. This amounts to executing the following code on each of the two CPU cores:

CPU CORE 1

```
void add( int *a, int *b, int *c )
{
    int tid = 0;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 2;
    }
}
```

CPU CORE 2

```
void add( int *a, int *b, int *c )
{
    int tid = 1;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 2;
    }
}
```

Of course, doing this on a CPU would require considerably more code than we have included in this example. You would need to provide a reasonable amount of infrastructure to create the worker threads that execute the function `add()` as well as make the assumption that each thread would execute in parallel, a scheduling assumption that is unfortunately not always true.

GPU VECTOR SUMS

We can accomplish the same addition very similarly on a GPU by writing `add()` as a device function. This should look similar to code you saw in the previous chapter. But before we look at the device code, we present `main()`. Although the GPU implementation of `main()` is different from the corresponding CPU version, nothing here should look new:

```
#include "../common/book.h"

#define N   10

int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }
    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int),
                              cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int),
                              cudaMemcpyHostToDevice ) );

    add<<<N,1>>>( dev_a, dev_b, dev_c );

    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int),
                              cudaMemcpyDeviceToHost ) );

    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    // free the memory allocated on the GPU
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );

    return 0;
}
```

You will notice some common patterns that we employ again:

- We allocate three arrays on the device using calls to `cudaMalloc()`: two arrays, `dev_a` and `dev_b`, to hold inputs, and one array, `dev_c`, to hold the result.
- Because we are environmentally conscientious coders, we clean up after ourselves with `cudaFree()`.
- Using `cudaMemcpy()`, we copy the input data to the device with the parameter `cudaMemcpyHostToDevice` and copy the result data back to the host with `cudaMemcpyDeviceToHost`.
- We execute the device code in `add()` from the host code in `main()` using the triple angle bracket syntax.

As an aside, you may be wondering why we fill the input arrays on the CPU. There is no reason in particular why we *need* to do this. In fact, the performance of this step would be faster if we filled the arrays on the GPU. But we intend to show how a particular operation, namely, the addition of two vectors, can be implemented on a graphics processor. As a result, we ask you to imagine that this is but one step of a larger application where the input arrays `a[]` and `b[]` have been generated by some other algorithm or loaded from the hard drive by the user. In summary, it will suffice to pretend that this data appeared out of nowhere and now we need to do something with it.

Moving on, our `add()` routine looks similar to its corresponding CPU implementation:

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;    // handle the data at this index
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

Again we see a common pattern with the function `add()`:

- We have written a function called `add()` that executes on the device. We accomplished this by taking C code and adding a `__global__` qualifier to the function name.

So far, there is nothing new in this example except it can do more than add 2 and 7. However, there *are* two noteworthy components of this example: The parameters within the triple angle brackets and the code contained in the kernel itself both introduce new concepts.

Up to this point, we have always seen kernels launched in the following form:

```
kernel<<<1,1>>>( param1, param2, … );
```

But in this example we are launching with a number in the angle brackets that is not 1:

```
add<<<N,1>>>( dev _ a, dev _ b, dev _ c );
```

What gives?

Recall that we left those two numbers in the angle brackets unexplained; we stated vaguely that they were parameters to the runtime that describe how to launch the kernel. Well, the first number in those parameters represents the number of parallel blocks in which we would like the device to execute our kernel. In this case, we're passing the value N for this parameter.

For example, if we launch with kernel<<<2,1>>>(), you can think of the runtime creating two copies of the kernel and running them in parallel. We call each of these parallel invocations a *block*. With kernel<<<256,1>>>(), you would get 256 *blocks* running on the GPU. Parallel programming has never been easier.

But this raises an excellent question: The GPU runs N copies of our kernel code, but how can we tell from within the code which block is currently running? This question brings us to the second new feature of the example, the kernel code itself. Specifically, it brings us to the variable blockIdx.x:

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;    // handle the data at this index
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

At first glance, it looks like this variable should cause a syntax error at compile time since we use it to assign the value of tid, but we have never defined it. However, there is no need to define the variable blockIdx; this is one of the built-in variables that the CUDA runtime defines for us. Furthermore, we use this variable for exactly what it sounds like it means. It contains the value of the block index for whichever block is currently running the device code.

Why, you may then ask, is it not just blockIdx? Why blockIdx.x? As it turns out, CUDA C allows you to define a group of blocks in two dimensions. For problems with two-dimensional domains, such as matrix math or image processing, it is often convenient to use two-dimensional indexing to avoid annoying translations from linear to rectangular indices. Don't worry if you aren't familiar with these problem types; just know that using two-dimensional indexing can sometimes be more convenient than one-dimensional indexing. But you never *have* to use it. We won't be offended.

When we launched the kernel, we specified N as the number of parallel blocks. We call the collection of parallel blocks a *grid*. This specifies to the runtime system that we want a one-dimensional *grid* of N blocks (scalar values are interpreted as one-dimensional). These threads will have varying values for blockIdx.x, the first taking value 0 and the last taking value N-1. So, imagine four blocks, all running through the same copy of the device code but having different values for the variable blockIdx.x. This is what the actual code being executed in each of the four parallel blocks looks like after the runtime substitutes the appropriate block index for blockIdx.x:

BLOCK 1

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 0;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 2

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 1;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 3

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 2;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 4

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 3;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

If you recall the CPU-based example with which we began, you will recall that we needed to walk through indices from 0 to N-1 in order to sum the two vectors. Since the runtime system is already launching a kernel where each block will have one of these indices, nearly all of this work has already been done for us. Because we're something of a lazy lot, this is a good thing. It affords us more time to blog, probably about how lazy we are.

The last remaining question to be answered is, why do we check whether tid is less than N? It *should* always be less than N, since we've specifically launched our kernel such that this assumption holds. But our desire to be lazy also makes us paranoid about someone breaking an assumption we've made in our code. Breaking code assumptions means broken code. This means bug reports, late nights tracking down bad behavior, and generally lots of activities that stand between us and our blog. If we didn't check that tid is less than N and subsequently fetched memory that wasn't ours, this would be bad. In fact, it could possibly kill the execution of your kernel, since GPUs have sophisticated memory management units that kill processes that seem to be violating memory rules.

If you encounter problems like the ones just mentioned, one of the HANDLE_ERROR() macros that we've sprinkled so liberally throughout the code will detect and alert you to the situation. As with traditional C programming, the lesson here is that functions return error codes for a reason. Although it is always tempting to ignore these error codes, we would love to save *you* the hours of pain through which *we* have suffered by urging that you *check the results of every operation that can fail*. As is often the case, the presence of these errors will not prevent you from continuing the execution of your application, but they will most certainly cause all manner of unpredictable and unsavory side effects downstream.

At this point, you're running code in parallel on the GPU. Perhaps you had heard this was tricky or that you had to understand computer graphics to do general-purpose programming on a graphics processor. We hope you are starting to see how CUDA C makes it much easier to get started writing parallel code on a GPU. We used the example only to sum vectors of length 10. If you would like to see how easy it is to generate a massively parallel application, try changing the 10 in the line #define N 10 to 10000 or 50000 to launch tens of thousands of parallel blocks. Be warned, though: No dimension of your launch of blocks may exceed 65,535. This is simply a hardware-imposed limit, so you will start to see failures if you attempt launches with more blocks than this. In the next chapter, we will see how to work within this limitation.

4.2.2 A FUN EXAMPLE

We don't mean to imply that adding vectors is anything less than fun, but the following example will satisfy those looking for some flashy examples of parallel CUDA C.

The following example will demonstrate code to draw slices of the Julia Set. For the uninitiated, the Julia Set is the boundary of a certain class of functions over complex numbers. Undoubtedly, this sounds even less fun than vector addition and matrix multiplication. However, for almost all values of the function's parameters, this boundary forms a fractal, one of the most interesting and beautiful curiosities of mathematics.

The calculations involved in generating such a set are quite simple. At its heart, the Julia Set evaluates a simple iterative equation for points in the complex plane. A point is *not* in the set if the process of iterating the equation diverges for that point. That is, if the sequence of values produced by iterating the equation grows toward infinity, a point is considered *outside* the set. Conversely, if the values taken by the equation remain bounded, the point *is* in the set.

Computationally, the iterative equation in question is remarkably simple, as shown in Equation 4.1.

Equation 4.1

$$Z_{n+1} = Z_n^2 + C$$

Computing an iteration of Equation 4.1 would therefore involve squaring the current value and adding a constant to get the next value of the equation.

CPU JULIA SET

We will examine a source listing now that will compute and visualize the Julia Set. Since this is a more complicated program than we have studied so far, we will split it into pieces here. Later in the chapter, you will see the entire source listing.

```
int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char *ptr = bitmap.get_ptr();

    kernel( ptr );

    bitmap.display_and_exit();
}
```

Our main routine is remarkably simple. It creates the appropriate size bitmap image using a utility library provided. Next, it passes a pointer to the bitmap data to the kernel function.

```
void kernel( unsigned char *ptr ){
    for (int y=0; y<DIM; y++) {
        for (int x=0; x<DIM; x++) {
            int offset = x + y * DIM;

            int juliaValue = julia( x, y );
            ptr[offset*4 + 0] = 255 * juliaValue;
            ptr[offset*4 + 1] = 0;
            ptr[offset*4 + 2] = 0;
            ptr[offset*4 + 3] = 255;
        }
    }
 }
```

The computation kernel does nothing more than iterate through all points we care to render, calling `julia()` on each to determine membership in the Julia Set. The function `julia()` will return 1 if the point is in the set and 0 if it is not in the set. We set the point's color to be red if `julia()` returns 1 and black if it returns 0. These colors are arbitrary, and you should feel free to choose a color scheme that matches your personal aesthetics.

```
int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}
```

This function is the meat of the example. We begin by translating our pixel coordinate to a coordinate in complex space. To center the complex plane at the image center, we shift by `DIM/2`. Then, to ensure that the image spans the range of -1.0 to 1.0, we scale the image coordinate by `DIM/2`. Thus, given an image point at `(x,y)`, we get a point in complex space at `( (DIM/2 – x)/(DIM/2), ((DIM/2 – y)/(DIM/2) )`.

Then, to potentially zoom in or out, we introduce a `scale` factor. Currently, the scale is hard-coded to be 1.5, but you should tweak this parameter to zoom in or out. If you are feeling really ambitious, you could make this a command-line parameter.

After obtaining the point in complex space, we then need to determine whether the point is in or out of the Julia Set. If you recall the previous section, we do this by computing the values of the iterative equation $Z_{n+1} = z_n^2 + C$. Since C is some arbitrary complex-valued constant, we have chosen `-0.8 + 0.156i` because it happens to yield an interesting picture. You should play with this constant if you want to see other versions of the Julia Set.

In the example, we compute 200 iterations of this function. After each iteration, we check whether the magnitude of the result exceeds some threshold (1,000 for our purposes). If so, the equation is diverging, and we can return 0 to indicate that the point is *not* in the set. On the other hand, if we finish all 200 iterations and the magnitude is still bounded under 1,000, we assume that the point is in the set, and we return 1 to the caller, `kernel()`.

Since all the computations are being performed on complex numbers, we define a generic structure to store complex numbers.

```
struct cuComplex {
    float   r;
    float   i;
    cuComplex( float a, float b ) : r(a), i(b)  {}
    float magnitude2( void ) { return r * r + i * i; }
    cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};
```

The class represents complex numbers with two data elements: a single-precision real component r and a single-precision imaginary component i. The class defines addition and multiplication operators that combine complex numbers as expected. (If you are completely unfamiliar with complex numbers, you can get a quick primer online.) Finally, we define a method that returns the magnitude of the complex number.

GPU JULIA SET

The device implementation is remarkably similar to the CPU version, continuing a trend you may have noticed.

```
int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char    *dev_bitmap;

    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                              bitmap.image_size() ) );

    dim3    grid(DIM,DIM);
    kernel<<<grid,1>>>( dev_bitmap );

    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(),
                              dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost ) );
    bitmap.display_and_exit();

    cudaFree( dev_bitmap );
}
```

This version of main() looks much more complicated than the CPU version, but the flow is actually identical. Like with the CPU version, we create a DIM x DIM bitmap image using our utility library. But because we will be doing computation on a GPU, we also declare a pointer called `dev_bitmap` to hold a copy of the data on the device. And to hold data, we need to allocate memory using `cudaMalloc()`.

We then run our `kernel()` function exactly like in the CPU version, although now it is a `__global__` function, meaning it will run on the GPU. As with the CPU example, we pass `kernel()` the pointer we allocated in the previous line to store the results. The only difference is that the memory resides on the GPU now, not on the host system.

The most significant difference is that we specify how many parallel blocks on which to execute the function `kernel()`. Because each point can be computed independently of every other point, we simply specify one copy of the function for each point we want to compute. We mentioned that for some problem domains, it helps to use two-dimensional indexing. Unsurprisingly, computing function values over a two-dimensional domain such as the complex plane is one of these problems. So, we specify a two-dimensional grid of blocks in this line:

```
    dim3  grid(DIM,DIM);
```

The type `dim3` is not a standard C type, lest you feared you had forgotten some key pieces of information. Rather, the CUDA runtime header files define some convenience types to encapsulate multidimensional tuples. The type `dim3` represents a three-dimensional tuple that will be used to specify the size of our launch. But why do we use a three-dimensional value when we oh-so-clearly stated that our launch is a *two-dimensional* grid?

Frankly, we do this because a three-dimensional, `dim3` value is what the CUDA runtime expects. Although a three-dimensional launch grid is not currently supported, the CUDA runtime still expects a `dim3` variable where the last component equals 1. When we initialize it with only two values, as we do in the statement `dim3 grid(DIM,DIM)`, the CUDA runtime automatically fills the third dimension with the value 1, so everything here will work as expected. Although it's possible that NVIDIA will support a three-dimensional grid in the future, for now we'll just play nicely with the kernel launch API because when coders and APIs fight, the API always wins.

We then pass our `dim3` variable `grid` to the CUDA runtime in this line:

```
kernel<<<grid,1>>>( dev _ bitmap );
```

Finally, a consequence of the results residing on the device is that after executing `kernel()`, we have to copy the results back to the host. As we learned in previous chapters, we accomplish this with a call to `cudaMemcpy()`, specifying the direction `cudaMemcpyDeviceToHost` as the last argument.

```
    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(),
                              dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost ) );
```

One of the last wrinkles in the difference of implementation comes in the implementation of `kernel()`.

```
__global__ void kernel( unsigned char *ptr ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = blockIdx.x;
    int y = blockIdx.y;
    int offset = x + y * gridDim.x;

    // now calculate the value at that position
    int juliaValue = julia( x, y );
    ptr[offset*4 + 0] = 255 * juliaValue;
    ptr[offset*4 + 1] = 0;
    ptr[offset*4 + 2] = 0;
    ptr[offset*4 + 3] = 255;
}
```

First, we need `kernel()` to be declared as a `__global__` function so it runs on the device but can be called from the host. Unlike the CPU version, we no longer need nested `for()` loops to generate the pixel indices that get passed to `julia()`. As with the vector addition example, the CUDA runtime generates these indices for us in the variable `blockIdx`. This works because we declared our grid of blocks to have the same dimensions as our image, so we get one block for each pair of integers `(x,y)` between `(0,0)` and `(DIM-1, DIM-1)`.

Next, the only additional information we need is a linear offset into our output buffer, `ptr`. This gets computed using another built-in variable, `gridDim`. This variable is a constant across all blocks and simply holds the dimensions of the grid that was launched. In this example, it will always be the value `(DIM, DIM)`. So, multiplying the row index by the grid width and adding the column index will give us a unique index into `ptr` that ranges from `0` to `(DIM*DIM-1)`.

```
    int offset = x + y * gridDim.x;
```

Finally, we examine the actual code that determines whether a point is in or out of the Julia Set. This code should look identical to the CPU version, continuing a trend we have seen in many examples now.

```
__device__ int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}
```

Again, we define a `cuComplex` structure that defines a method for storing a complex number with single-precision floating-point components. The structure also defines addition and multiplication operators as well as a function to return the magnitude of the complex value.

```
struct cuComplex {
    float   r;
    float   i;
    cuComplex( float a, float b ) : r(a), i(b)  {}
    __device__ float magnitude2( void ) {
        return r * r + i * i;
    }
    __device__ cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    __device__ cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};
```

Notice that we use the same language constructs in CUDA C that we use in our CPU version. The one difference is the qualifier `__device__`, which indicates that this code will run on a GPU and not on the host. Recall that because these functions are declared as `__device__` functions, they will be callable only from other `__device__` functions or from `__global__` functions.

Since we've interrupted the code with commentary so frequently, here is the entire source listing from start to finish:

```
#include "../common/book.h"
#include "../common/cpu_bitmap.h"

#define DIM 1000
```

```
struct cuComplex {
    float   r;
    float   i;
    cuComplex( float a, float b ) : r(a), i(b)  {}
    __device__ float magnitude2( void ) {
        return r * r + i * i;
    }
    __device__ cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    __device__ cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};

__device__ int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}
```

```
__global__ void kernel( unsigned char *ptr ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = blockIdx.x;
    int y = blockIdx.y;
    int offset = x + y * gridDim.x;

    // now calculate the value at that position
    int juliaValue = julia( x, y );
    ptr[offset*4 + 0] = 255 * juliaValue;
    ptr[offset*4 + 1] = 0;
    ptr[offset*4 + 2] = 0;
    ptr[offset*4 + 3] = 255;
}

int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char    *dev_bitmap;

    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                               bitmap.image_size() ) );

    dim3    grid(DIM,DIM);
    kernel<<<grid,1>>>( dev_bitmap );

    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost ) );
    bitmap.display_and_exit();

    HANDLE_ERROR( cudaFree( dev_bitmap ) );
}
```

When you run the application, you should see an animating visualization of the Julia Set. To convince you that it has earned the title “A Fun Example,” Figure 4.2 shows a screenshot taken from this application.

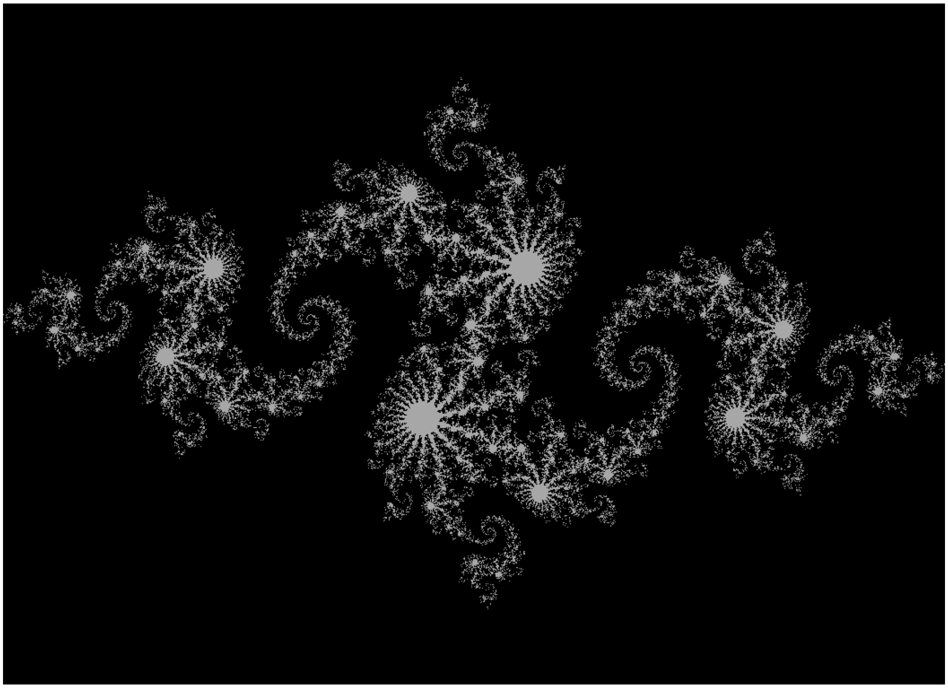

Figure 4.2 A screenshot from the GPU Julia Set application

4.3 Chapter Review

Congratulations, you can now write, compile, and run massively parallel code on a graphics processor! You should go brag to your friends. And if they are still under the misconception that GPU computing is exotic and difficult to master, they will be most impressed. The ease with which you accomplished it will be our secret. If they're people you trust with your secrets, suggest that they buy the book, too.

We have so far looked at how to instruct the CUDA runtime to execute multiple copies of our program in parallel on what we called *blocks*. We called the collection of blocks we launch on the GPU a *grid*. As the name might imply, a grid can be either a one- or two-dimensional collection of blocks. Each copy of the kernel can determine which block it is executing with the built-in variable `blockIdx`. Likewise, it can determine the size of the grid by using the built-in variable `gridDim`. Both of these built-in variables proved useful within our kernel to calculate the data index for which each block is responsible.

Index

A

add() function, CPU vector sums, 40–44
add_to_table() kernel, GPU hash table, 272
ALUs (arithmetic logic units)
  CUDA Architecture, 7
  using constant memory, 96
anim_and_exit() method, GPU ripples, 70
anim_gpu() routine, texture memory, 123, 129
animation
  GPU Julia Set example, 50–57
  GPU ripple using threads, 69–74
  heat transfer simulation, 121–125
animExit(), 149
asynchronous call
  cudaMemcpyAsync() as, 197
  using events with, 109
atomic locks
  GPU hash table, 274–275
  overview of, 251–254
atomicAdd()
  atomic locks, 251–254
  histogram kernel using global memory, 180
  not supporting floating-point numbers, 251
atomicCAS(), GPU lock, 252–253
atomicExch(), GPU lock, 253–254
atomics, 163–184
  advanced, 249–277
  compute capability of NVIDIA GPUs, 164–167
  dot product and, 248–251
  hash tables. *see* hash tables
  histogram computation, CPU, 171–173
  histogram computation, GPU, 173–179
  histogram computation, overview, 170
  histogram kernel using global memory atomics, 179–181
  histogram kernel using shared/global memory atomics, 181–183
  for minimum compute capability, 167–168
  locks, 251–254
  operations, 168–170
  overview of, 163–164, 249
  summary review, 183–184, 277

B

bandwidth, constant memory saving, 106–107
Basic Linear Algebra Subprograms (BLAS), CUBLAS library, 239–240
bin counts, CPU histogram computation, 171–173
BLAS (Basic Linear Algebra Subprograms), CUBLAS library, 239–240
blend_kernel()
  2D texture memory, 131–133
  texture memory, 127–129
blockDim variable
  2D texture memory, 132–133
  dot product computation, 76–78, 85
  dot product computation, incorrect optimization, 88
  dot product computation with atomic locks, 255–256
  dot product computation, zero-copy memory, 221–222
  GPU hash table implementation, 272
  GPU ripple using threads, 72–73
  GPU sums of a longer vector, 63–65
  GPU sums of arbitrarily long vectors, 66–67
  graphics interoperability, 145
  histogram kernel using global memory atomics, 179–180
  histogram kernel using shared/global memory atomics, 182–183
  multiple CUDA streams, 200
  ray tracing on GPU, 102
  shared memory bitmap, 91
  temperature update computation, 119–120

blockIdx variable
  2D texture memory, 132–133
  defined, 57
  dot product computation, 76–77, 85
  dot product computation with atomic locks, 255–256
  dot product computation, zero-copy memory, 221–222
  GPU hash table implementation, 272
  GPU Julia Set, 53
  GPU ripple using threads, 72–73
  GPU sums of a longer vector, 63–64
  GPU vector sums, 44–45
  graphics interoperability, 145
  histogram kernel using global memory atomics, 179–180
  histogram kernel using shared/global memory atomics, 182–183
  multiple CUDA streams, 200
  ray tracing on GPU, 102
  shared memory bitmap, 91
  temperature update computation, 119–121
blocks
  defined, 57
  GPU Julia Set, 51
  GPU vector sums, 44–45
  hardware-imposed limits on, 46
  splitting into threads. *see* parallel blocks, splitting into threads
breast cancer, CUDA applications for, 8–9
bridges, connecting multiple GPUs, 224
buckets, hash table
  concept of, 259–260
  GPU hash table implementation, 269–275
  multithreaded hash tables and, 267–268
bufferObj variable
  creating GPUAnimBitmap, 149
  registering with CUDA runtime, 143
  registering with cudaGraphicsGLRegisterBuffer(), 151
  setting up graphics interoperability, 141, 143–144
buffers, declaring shared memory, 76–77

C

cache[] shared memory variable
  declaring buffer of shared memory named, 76–77
  dot product computation, 79–80, 85–86
  dot product computation with atomic locks, 255–256
cacheIndex, incorrect dot product optimization, 88
caches, texture, 116–117
callbacks, GPUAnimBitmap user registration for, 149
Cambridge University, CUDA applications, 9–10
camera
  ray tracing concepts, 97–98
  ray tracing on GPU, 99–104
cellular phones, parallel processing in, 2
central processing units. *see* CPUs (central processing units)
cleaning agents, CUDA applications for, 10–11
clickDrag(), 149
clock speed, evolution of, 2–3
code, breaking assumptions, 45–46
code resources, CUDa, 246–248
collision resolution, hash tables, 260–261
color
  CPU Julia Set, 48–49
  early days of GPU computing, 5–6
  ray tracing concepts, 98
compiler
  for minimum compute capability, 167–168
  standard C, for GPU code, 18–19
complex numbers
  defining generic class to store, 49–50
  storing with single-precision floating-point components, 54
computational fluid dynamics, CUDA applications for, 9–10
compute capability
  compiling for minimum, 167–168
  cudaChooseDevice() and, 141
  defined, 164
  of NVIDIA GPUs, 164–167
  overview of, 141–142
computer games, 3D graphic development for, 4–5
constant memory
  accelerating applications with, 95
  measuring performance with events, 108–110
  measuring ray tracer performance, 110–114
  overview of, 96
  performance with, 106–107
  ray tracing introduction, 96–98
  ray tracing on GPU, 98–104
  ray tracing with, 104–106
  summary review, 114
__constant__function
  declaring memory as, 104–106
  performance with constant memory, 106–107
copy_const_kernel() kernel
  2D texture memory, 133
  using texture memory, 129–130

copy_constant_kernel(), computing temperature updates, 119–121
CPUAnimBitmap class, creating GPU ripple, 69–70, 147–148
CPUs (central processing units)
  evolution of clock speed, 2–3
  evolution of core count, 3
  freeing memory. *see* free(), C language
  hash tables, 261–267
  histogram computation on, 171–173
  as host in this book, 23
  thread management and scheduling in, 72
  vector sums, 39–41
  verifying GPU histogram using reverse CPU histogram, 175–176
CUBLAS library, 239–240
cuComplex structure, CPU Julia Set, 48–49
cuComplex structure, GPU Julia Set, 53–55
CUDA, Supercomputing for the Masses , 245–246
CUDA Architecture
  computational fluid dynamic applications, 9–10
  defined, 7
  environmental science applications, 10–11
  first application of, 7
  medical imaging applications, 8–9
  resource for understanding, 244–245
  using, 7–8
CUDA C
  computational fluid dynamic applications, 9–10
  CUDA development toolkit, 16–18
  CUDA-enabled graphics processor, 14–16
  debugging, 241–242
  development environment setup. *see* development environment setup
  development of, 7
  environmental science applications, 10–11
  getting started, 13–20
  medical imaging applications, 8–9
  NVIDIA device driver, 16
  on multiple GPUs. *see* GPUs (graphics processing units), multi-system
  overview of, 21–22
  parallel programming in. *see* parallel programming, CUDA
  passing parameters, 24–27
  querying devices, 27–33
  standard C compiler, 18–19
  summary review, 19, 35
  using device properties, 33–35
  writing first program, 22–24
CUDA Data Parallel Primitives Library (CUDPP), 246
CUDA event API, and performance, 108–110
CUDA Memory Checker, 242
CUDA streams
  GPU work scheduling with, 205–208
  multiple, 198–205, 208–210
  overview of, 192
  single, 192–198
  summary review, 211
CUDA Toolkit, 238–240
  in development environment, 16–18
CUDA tools
  CUBLAS library, 239–240
  CUDA Toolkit, 238–239
  CUFFT library, 239
  debugging CUDA C, 241–242
  GPU Computing SDK download, 240–241
  NVIDIA Performance Primitives, 241
  overview of, 238
  Visual Profiler, 243–244
CUDA Zone, 167
cuda_malloc_test(), page-locked memory, 189
cudaBindTexture(), texture memory, 126–127
cudaBindTexture2D(), texture memory, 134
cudaChannelFormatDesc(), binding 2D textures, 134
cudaChooseDevice()
  defined, 34
  GPUAnimBitmap initialization, 150
  for valid ID, 141–142
cudaD39SetDirect3DDevice(), DirectX interoperability, 160–161
cudaDeviceMapHost(), zero-copy memory dot product, 221
cudaDeviceProp structure
  cudaChooseDevice() working with, 141
  multiple CUDA streams, 200
  overview of, 28–31
  single CUDA streams, 193–194
  using device properties, 34
CUDA-enabled graphics processors, 14–16
cudaEventCreate()
  2D texture memory, 134
  CUDA streams, 192, 194, 201
  GPU hash table implementation, 274–275
  GPU histogram computation, 173, 177
  measuring performance with events, 108–110, 112
  page-locked host memory application, 188–189
  performing animation with GPUAnimBitmap, 158
  ray tracing on GPU, 100
  standard host memory dot product, 215
  texture memory, 124
  zero-copy host memory, 215, 217

cudaEventDestroy()
- defined, 112
- GPU hash table implementation, 275
- GPU histogram computation, 176, 178
- heat transfer simulation, 123, 131, 137
- measuring performance with events, 111–113
- page-locked host memory, 189–190
- texture memory, 136
- zero-copy host memory, 217, 220

cudaEventElapsedTime()
- 2D texture memory, 130
- CUDA streams, 198, 204
- defined, 112
- GPU hash table implementation, 275
- GPU histogram computation, 175, 178
- heat transfer simulation animation, 122
- heat transfer using graphics interoperability, 157
- page-locked host memory, 188, 190
- standard host memory dot product, 216
- zero-copy memory dot product, 219

cudaEventRecord()
- CUDA streams, 194, 198, 201
- CUDA streams and, 192
- GPU hash table implementation, 274–275
- GPU histogram computation, 173, 175, 177
- heat transfer simulation animation, 122
- heat transfer using graphics interoperability, 156–157
- measuring performance with events, 108–109
- measuring ray tracer performance, 110–113
- page-locked host memory, 188–190
- ray tracing on GPU, 100
- standard host memory dot product, 216
- using texture memory, 129–130

cudaEventSynchronize()
- 2D texture memory, 130
- GPU hash table implementation, 275
- GPU histogram computation, 175, 178
- heat transfer simulation animation, 122
- heat transfer using graphics interoperability, 157
- measuring performance with events, 109, 111, 113
- page-locked host memory, 188, 190
- standard host memory dot product, 216

cudaFree()
- allocating portable pinned memory, 235
- CPU vector sums, 42
- CUDA streams, 198, 205
- defined, 26–27
- dot product computation, 84, 87
- dot product computation with atomic locks, 258
- GPU hash table implementation, 269–270, 275
- GPU ripple using threads, 69
- GPU sums of arbitrarily long vectors, 69
- multiple CPUs, 229
- page-locked host memory, 189–190
- ray tracing on GPU, 101
- ray tracing with constant memory, 105
- shared memory bitmap, 91
- standard host memory dot product, 217

cudaFreeHost()
- allocating portable pinned memory, 233
- CUDA streams, 198, 204
- defined, 190
- freeing buffer allocated with cudaHostAlloc(), 190
- zero-copy memory dot product, 220

CUDA-GDB debugging tool, 241–242

cudaGetDevice()
- CUDA streams, 193, 200
- device properties, 34
- zero-copy memory dot product, 220

cudaGetDeviceCount()
- device properties, 34
- getting count of CUDA devices, 28
- multiple CPUs, 224–225

cudaGetDeviceProperties()
- determining if GPU is integrated or discrete, 223
- multiple CUDA streams, 200
- querying devices, 33–35
- zero-copy memory dot product, 220

cudaGLSetGLDevice()
- graphics interoperation with OpenGL, 150
- preparing CUDA to use OpenGL driver, 142

cudaGraphicsGLRegisterBuffer(), 143, 151

cudaGraphicsMapFlagsNone(), 143

cudaGraphicsMapFlagsReadOnly(), 143

cudaGraphicsMapFlagsWriteDiscard(), 143

cudaGraphicsUnapResources(), 144

cudaHostAlloc()
- CUDA streams, 195, 202
- malloc() versus, 186–187
- page-locked host memory application, 187–192
- zero-copy memory dot product, 217–220

cudaHostAllocDefault()
- CUDA streams, 195, 202
- default pinned memory, 214
- page-locked host memory, 189–190

cudaHostAllocMapped() flag
- default pinned memory, 214
- portable pinned memory, 231
- zero-copy memory dot product, 217–218

cudaHostAllocPortable(), portable pinned memory, 230–235

cudaHostAllocWriteCombined() flag
- portable pinned memory, 231
- zero-copy memory dot product, 217–218

cudaHostGetDevicePointer()
  portable pinned memory, 234
  zero-copy memory dot product, 218–219
cudaMalloc(), 124
  2D texture memory, 133–135
  allocating device memory using, 26
  CPU vector sums application, 42
  CUDA streams, 194, 201–202
  dot product computation, 82, 86
  dot product computation, standard host memory, 215
  dot product computation with atomic locks, 256
  GPU hash table implementation, 269, 274–275
  GPU Julia Set, 51
  GPU lock function, 253
  GPU ripple using threads, 70
  GPU sums of arbitrarily long vectors, 68
  measuring ray tracer performance, 110, 112
  portable pinned memory, 234
  ray tracing on GPU, 100
  ray tracing with constant memory, 105
  shared memory bitmap, 90
  using multiple CPUs, 228
  using texture memory, 127
cuda-memcheck, 242
cudaMemcpy()
  2D texture binding, 136
  copying data between host and device, 27
  CPU vector sums application, 42
  dot product computation, 82–83, 86
  dot product computation with atomic locks, 257
  GPU hash table implementation, 270, 274–275
  GPU histogram computation, 174–175
  GPU Julia Set, 52
  GPU lock function implementation, 253
  GPU ripple using threads, 70
  GPU sums of arbitrarily long vectors, 68
  heat transfer simulation animation, 122–125
  measuring ray tracer performance, 111
  page-locked host memory and, 187, 189
  ray tracing on GPU, 101
  standard host memory dot product, 216
  using multiple CPUs, 228–229
cudaMemcpyAsync()
  GPU work scheduling, 206–208
  multiple CUDA streams, 203, 208–210
  single CUDA streams, 196
  timeline of intended application execution using multiple streams, 199
cudaMemcpyDeviceToHost()
  CPU vector sums application, 42
  dot product computation, 82, 86–87
  GPU hash table implementation, 270
  GPU histogram computation, 174–175
  GPU Julia Set, 52
  GPU sums of arbitrarily long vectors, 68
  multiple CUDA streams, 204
  page-locked host memory, 190
  ray tracing on GPU, 101
  shared memory bitmap, 91
  standard host memory dot product, 216
  using multiple CPUs, 229
cudaMemcpyHostToDevice()
  CPU vector sums application, 42
  dot product computation, 86
  GPU sums of arbitrarily long vectors, 68
  implementing GPU lock function, 253
  measuring ray tracer performance, 111
  multiple CPUs, 228
  multiple CUDA streams, 203
  page-locked host memory, 189
  standard host memory dot product, 216
cudaMemcpyToSymbol(), constant memory, 105–106
cudaMemset()
  GPU hash table implementation, 269
  GPU histogram computation, 174
CUDA.NET project, 247
cudaSetDevice()
  allocating portable pinned memory, 231–232, 233–234
  using device properties, 34
  using multiple CPUs, 227–228
cudaSetDeviceFlags()
  allocating portable pinned memory, 231, 234
  zero-copy memory dot product, 221
cudaStreamCreate(), 194, 201
cudaStreamDestroy(), 198, 205
cudaStreamSynchronize(), 197–198, 204
cudaThreadSynchronize(), 219
cudaUnbindTexture(), 2D texture memory, 136–137
CUDPP (CUDA Data Parallel Primitives Library), 246
CUFFT library, 239
CULAtools, 246
current animation time, GPU ripple using threads, 72–74

D

debugging CUDA C, 241–242
detergents, CUDA applications, 10–11
dev_bitmap pointer, GPU Julia Set, 51
development environment setup
  CUDA Toolkit, 16–18
  CUDA-enabled graphics processor, 14–16
  NVIDIA device driver, 16
  standard C compiler, 18–19
  summary review, 19

device drivers, 16
device overlap, GPU, 194, 198–199
__device__function
  GPU hash table implementation, 268–275
  GPU Julia Set, 54
devices
  getting count of CUDA, 28
  GPU vector sums, 41–46
  passing parameters, 25–27
  querying, 27–33
  use of term in this book, 23
  using properties of, 33–35
devPtr, graphics interoperability, 144
dim3 variable grid, GPU Julia Set, 51–52
DIMxDIM bitmap image, GPU Julia Set, 49–51, 53
direct memory access (DMA), for page-locked
    memory, 186
DirectX
  adding standard C to, 7
  breakthrough in GPU technology, 5–6
  GeForce 8800 GTX, 7
  graphics interoperability, 160–161
discrete GPUs, 222–224
display accelerators, 2D, 4
DMA (direct memory access), for page-locked
    memory, 186
dot product computation
  optimized incorrectly, 87–90
  shared memory and, 76–87
  standard host memory version of, 215–217
  using atomics to keep entirely on GPU, 250–251,
    254–258
dot product computation, multiple GPUs
  allocating portable pinned memory, 230–235
  using, 224–229
  zero-copy, 217–222
  zero-copy performance, 223
Dr. Dobb's CUDA, 245–246
DRAMs, discrete GPUs with own dedicated, 222–223
draw_func, graphics interoperability, 144–146

E

end_thread(), multiple CPUs, 226
environmental science, CUDA applications for, 10–11
event timer. *see* timer, event
events
  computing elapsed time between recorded. *see*
    cudaEventElapsedTime()
  creating. *see* cudaEventCreate()
  GPU histogram computation, 173
  measuring performance with, 95
  measuring ray tracer performance, 110–114
  overview of, 108–110
  recording. *see* cudaEventRecord()
  stopping and starting. *see*
    cudaEventDestroy()
  summary review, 114
EXIT_FAILURE(), passing parameters, 26

F

fAnim(), storing registered callbacks, 149
Fast Fourier Transform library, NVIDIA, 239
first program, writing, 22–24
flags, in graphics interoperability, 143
float_to_color() kernels, in graphics
    interoperability, 157
floating-point numbers
  atomic arithmetic not supported for, 251
  CUDA Architecture designed for, 7
  early days of GPU computing not able to handle, 6
FORTRAN applications
  CUBLAS compatibility with, 239–240
  language wrapper for CUDA C, 246
forums, NVIDIA, 246
fractals. *see* Julia Set example
free(), C language
  cudaFree( ) versus, 26–27
  dot product computation with atomic locks, 258
  GPU hash table implementation, 275
  multiple CPUs, 227
  standard host memory dot product, 217

G

GeForce 256, 5
GeForce 8800 GTX, 7
generate_frame(), GPU ripple, 70, 72–73, 154
generic classes, storing complex numbers with,
    49–50
GL_PIXEL_UNPACK_BUFFER_ARB target, OpenGL
    interoperation, 151
glBindBuffer()
  creating pixel buffer object, 143
  graphics interoperability, 146
glBufferData(), pixel buffer object, 143
glDrawPixels()
  graphics interoperability, 146
  overview of, 154–155
glGenBuffers(), pixel buffer object, 143
global memory atomics
  GPU compute capability requirements, 167
  histogram kernel using, 179–181
  histogram kernel using shared and, 181–183

__global__ function
  add function, 43
  kernel call, 23–24
  running kernel() in GPU Julia Set application, 51–52
GLUT (GL Utility Toolkit)
  graphics interoperability setup, 144
  initialization of, 150
  initializing OpenGL driver by calling, 142
glutIdleFunc(), 149
glutInit(), 150
glutMainLoop(), 144
GPU Computing SDK download, 18, 240–241
GPU ripple
  with graphics interoperability, 147–154
  using threads, 69–74
GPU vector sums
  application, 41–46
  of arbitrarily long vectors, using threads, 65–69
  of longer vector, using threads, 63–65
  using threads, 61–63
gpu_anim.h, 152–154
GPUAnimBitmap structure
  creating, 148–152
  GPU ripple performing animation, 152–154
  heat transfer with graphics interoperability, 156–160
GPUs (graphics processing units)
  called "devices" in this book, 23
  developing code in CUDA C with CUDA-enabled, 14–16
  development of CUDA for, 6–8
  discrete versus integrated, 222–223
  early days of, 5–6
  freeing memory. *see* cudaFree()
  hash tables, 268–275
  histogram computation on, 173–179
  histogram kernel using global memory atomics, 179–181
  histogram kernel using shared/global memory atomics, 181–183
  history of, 4–5
  Julia Set example, 50–57
  measuring performance with events, 108–110
  ray tracing on, 98–104
  work scheduling, 205–208
GPUs (graphics processing units), multiple, 213–236
  overview of, 213–214
  portable pinned memory, 230–235
  summary review, 235–236
  using, 224–229
  zero-copy host memory, 214–222
  zero-copy performance, 222–223
graphics accelerators, 3D graphics, 4–5
graphics interoperability, 139–161
  DirectX, 160–161
  generating image data with kernel, 139–142
  GPU ripple with, 147–154
  heat transfer with, 154–160
  overview of, 139–140
  passing image data to Open GL for rendering, 142–147
  summary review, 161
graphics processing units. *see* GPUs (graphics processing units)
grey(), GPU ripple, 74
grid
  as collection of parallel blocks, 45
  defined, 57
  three-dimensional, 51
gridDim variable
  2D texture memory, 132–133
  defined, 57
  dot product computation, 77–78
  dot product computation with atomic locks, 255–256
  GPU hash table implementation, 272
  GPU Julia Set, 53
  GPU ripple using threads, 72–73
  GPU sums of arbitrarily long vectors, 66–67
  graphics interoperability setup, 145
  histogram kernel using global memory atomics, 179–180
  histogram kernel using shared/global memory atomics, 182–183
  ray tracing on GPU, 102
  shared memory bitmap, 91
  temperature update computation, 119–120
  zero-copy memory dot product, 222

H

half-warps, reading constant memory, 107
HANDLE_ERROR() macro
  2D texture memory, 133–136
  CUDA streams, 194–198, 201–204, 209–210
  dot product computation, 82–83, 86–87
  dot product computation with atomic locks, 256–258
  GPU hash table implementation, 270
  GPU histogram computation completion, 175
  GPU lock function implementation, 253
  GPU ripple using threads, 70
  GPU sums of arbitrarily long vectors, 68

HANDLE_ERROR() macro, *continued*
- heat transfer simulation animation, 122–125
- measuring ray tracer performance, 110–114
- page-locked host memory application, 188–189
- passing parameters, 26
- paying attention to, 46
- portable pinned memory, 231–235
- ray tracing on GPU, 100–101
- ray tracing with constant memory, 104–105
- shared memory bitmap, 90–91
- standard host memory dot product, 215–217
- texture memory, 127, 129
- zero-copy memory dot product, 217–222

hardware
- decoupling parallelization from method of executing, 66
- performing atomic operations on memory, 167

hardware limitations
- GPU sums of arbitrarily long vectors, 65–69
- number of blocks in single launch, 46
- number of threads per block in kernel launch, 63

hash function
- CPU hash table implementation, 261–267
- GPU hash table implementation, 268–275
- overview of, 259–261

hash tables
- concepts, 259–261
- CPU, 261–267
- GPU, 268–275
- multithreaded, 267–268
- performance, 276–277
- summary review, 277

heat transfer simulation
- 2D texture memory, 131–137
- animating, 121–125
- computing temperature updates, 119–121
- with graphics interoperability, 154–160
- simple heating model, 117–118
- using texture memory, 125–131

"Hello, World" example
- kernel call, 23–24
- passing parameters, 24–27
- writing first program, 22–23

Highly Optimized Object-oriented Many-particle Dynamics (HOOMD), 10–11

histogram computation
- on CPUs, 171–173
- on GPUs, 173–179
- overview, 170

histogram kernel
- using global memory atomics, 179–181
- using shared/global memory atomics, 181–183

hit() method, ray tracing on GPU, 99, 102

HOOMD (Highly Optimized Object-oriented Many-particle Dynamics), 10–11

hosts
- allocating memory to. *see* malloc()
- CPU vector sums, 39–41
- CUDA C blurring device code and, 26
- page-locked memory, 186–192
- passing parameters, 25–27
- use of term in this book, 23
- zero-copy host memory, 214–222

I

idle_func() member, GPUAnimBitmap, 154

IEEE requirements, ALUs, 7

increment operator (x++), 168–170

initialization
- CPU hash table implementation, 263, 266
- CPU histogram computation, 171
- GLUT, 142, 150, 173–174
- GPUAnimBitmap, 149

inner products. *see* dot product computation

integrated GPUs, 222–224

interleaved operations, 169–170

interoperation. *see* graphics interoperability

J

julia() function, 48–49, 53

Julia Set example
- CPU application of, 47–50
- GPU application of, 50–57
- overview of, 46–47

K

kernel
- 2D texture memory, 131–133
- blockIdx.x variable, 44
- call to, 23–24
- defined, 23
- GPU histogram computation, 176–178
- GPU Julia Set, 49–52
- GPU ripple performing animation, 154
- GPU ripple using threads, 70–72
- GPU sums of a longer vector, 63–65
- graphics interoperability, 139–142, 144–146
- "Hello, World" example of call to, 23–24
- launching with number in angle brackets that is not 1, 43–44
- passing parameters to, 24–27
- ray tracing on GPU, 102–104
- texture memory, 127–131

key_func, graphics interoperability, 144–146

keys
  CPU hash table implementation, 261–267
  GPU hash table implementation, 269–275
  hash table concepts, 259–260

L

language wrappers, 246–247
LAPACK (Linear Algebra Package), 246
light effects, ray tracing concepts, 97
Linux, standard C compiler for, 19
`Lock` structure, 254–258, 268–275
locks, atomic, 251–254

M

Macintosh OS X, standard C compiler, 19
`main()` routine
  2D texture memory, 133–136
  CPU hash table implementation, 266–267
  CPU histogram computation, 171
  dot product computation, 81–84
  dot product computation with atomic locks, 255–256
  GPU hash table implementation, 273–275
  GPU histogram computation, 173
  GPU Julia Set, 47, 50–51
  GPU ripple using threads, 69–70
  GPU vector sums, 41–42
  graphics interoperability, 144
  page-locked host memory application, 190–192
  ray tracing on GPU, 99–100
  ray tracing with constant memory, 104–106
  shared memory bitmap, 90
  single CUDA streams, 193–194
  zero-copy memory dot product, 220–222
`malloc()`
  `cudaHostAlloc()` versus, 186
  `cudaHostAlloc()` versus, 190
  `cudaMalloc( )` versus, 26
  ray tracing on GPU, 100
mammograms, CUDA applications for medical imaging, 9
`maxThreadsPerBlock` field, device properties, 63
media and communications processors (MCPs), 223
medical imaging, CUDA applications for, 8–9
`memcpy()`, C language, 27
memory
  allocating device. *see* `cudaMalloc()`
  constant. *see* constant memory
  CUDA Architecture creating access to, 7
  early days of GPU computing, 6
  executing device code that uses allocated, 70
  freeing. *see* `cudaFree()`; `free()`, C language
  GPU histogram computation, 173–174
  page-locked host (pinned), 186–192
  querying devices, 27–33
  shared. *see* shared memory
  texture. *see* texture memory
  use of term in this book, 23
Memory Checker, CUDA, 242
`memset()`, C language, 174
Microsoft Windows, Visual Studio C compiler, 18–19
Microsoft.NET, 247
multicore revolution, evolution of CPUs, 3
multiplication, in vector dot products, 76
multithreaded hash tables, 267–268
`mutex`, GPU lock function, 252–254

N

nForce media and communications processors (MCPs), 222–223
NVIDIA
  compute capability of various GPUs, 164–167
  creating 3D graphics for consumers, 5
  creating CUDA C for GPU, 7
  creating first GPU built with CUDA Architecture, 7
  CUBLAS library, 239–240
  CUDA-enabled graphics processors, 14–16
  CUDA-GDB debugging tool, 241–242
  CUFFT library, 239
  device driver, 16
  GPU Computing SDK download, 18, 240–241
  Parallel NSight debugging tool, 242
  Performance Primitives, 241
  products containing multiple GPUs, 224
  Visual Profiler, 243–244
NVIDIA CUDA Programming Guide, 31

O

`offset`, 2D texture memory, 133
on-chip caching. *see* constant memory; texture memory
one-dimensional blocks
  GPU sums of a longer vector, 63
  two-dimensional blocks versus, 44
online resources. *see* resources, online
OpenGL
  creating `GPUAnimBitmap`, 148–152
  in early days of GPU computing, 5–6
  generating image data with kernel, 139–142
  interoperation, 142–147
  writing 3D graphics, 4
operations, atomic, 168–170
optimization, incorrect dot product, 87–90

P

page-locked host memory
- allocating as portable pinned memory, 230–235
- overview of, 186–187
- restricted use of, 187
- single CUDA streams with, 195–197

parallel blocks
- GPU Julia Set, 51
- GPU vector sums, 45

parallel blocks, splitting into threads
- GPU sums of arbitrarily long vectors, 65–69
- GPU sums of longer vector, 63–65
- GPU vector sums using threads, 61–63
- overview of, 60
- vector sums, 60–61

Parallel NSight debugging tool, 242

parallel processing
- evolution of CPUs, 2–3
- past perception of, 1

parallel programming, CUDA
- CPU vector sums, 39–41
- example, CPU Julia Set application, 47–50
- example, GPU Julia Set application, 50–57
- example, overview, 46–47
- GPU vector sums, 41–46
- overview of, 38
- summary review, 56
- summing vectors, 38–41

parameter passing, 24–27, 40, 72

PC gaming, 3D graphics for, 4–5

PCI Express slots, adding multiple GPUs to, 224

performance
- constant memory and, 106–107
- evolution of CPUs, 2–3
- hash table, 276
- launching kernel for GPU histogram computation, 176–177
- measuring with events, 108–114
- page-locked host memory and, 187
- zero-copy memory and, 222–223

pinned memory
- allocating as portable, 230–235
- `cudaHostAllocDefault()` getting default, 214
- as page-locked memory. *see* page-locked host memory

pixel buffer objects (PBO), OpenGL, 142–143

pixel shaders, early days of GPU computing, 5–6

pixels, number of threads per block, 70–74

portable computing devices, 2

Programming Massively Parallel Processors: A Hands-on Approach (Kirk, Hwu), 244

properties
- `cudaDeviceProp` structure. *see* `cudaDeviceProp` structure
- `maxThreadsPerBlock` field for device, 63
- reporting device, 31
- using device, 33–35

PyCUDA project, 246–247

Python language wrappers for CUDA C, 246

Q

querying, devices, 27–33

R

rasterization, 97

ray tracing
- concepts behind, 96–98
- with constant memory, 104–106
- on GPU, 98–104
- measuring performance, 110–114

read-modify-write operations
- atomic operations as, 168–170, 251
- using atomic locks, 251–254

read-only memory. *see* constant memory; texture memory

reductions
- dot products as, 83
- overview of, 250
- shared memory and synchronization for, 79–81

references, texture memory, 126–127, 131–137

registration
- `bufferObj` with `cudaGraphicsGLRegisterBuffer()`, 151
- callback, 149

rendering, GPUs performing complex, 139

`resource` variable
- creating `GPUAnimBitmap`, 148–152
- graphics interoperation, 141

resources, online
- CUDA code, 246–248
- CUDA Toolkit, 16
- CUDA University, 245
- CUDPP, 246
- CULAtools, 246
- Dr. Dobb's CUDA, 246
- GPU Computing SDK code samples, 18
- language wrappers, 246–247
- NVIDIA device driver, 16
- NVIDIA forums, 246
- standard C compiler for Mac OS X, 19
- Visual Studio C compiler, 18

resources, written
  CUDA U, 245–246
  forums, 246
  programming massive parallel processors, 244–245
ripple, GPU
  with graphics interoperability, 147–154
  producing, 69–74
routine()
  allocating portable pinned memory, 232–234
  using multiple CPUs, 226–228
Russian nesting doll hierarchy, 164

S
scalable link interface (SLI), adding multiple GPUs with, 224
scale factor, CPU Julia Set, 49
scientific computations, in early days, 6
screenshots
  animated heat transfer simulation, 126
  GPU Julia Set example, 57
  GPU ripple example, 74
  graphics interoperation example, 147
  ray tracing example, 103–104
  rendered with proper synchronization, 93
  rendered without proper synchronization, 92
shading languages, 6
shared data buffers, kernel/OpenGL rendering interoperation, 142
shared memory
  atomics, 167, 181–183
  bitmap, 90–93
  CUDA Architecture creating access to, 7
  dot product, 76–87
  dot product optimized incorrectly, 87–90
  and synchronization, 75
Silicon Graphics, OpenGL library, 4
simulation
  animation of, 121–125
  challenges of physical, 117
  computing temperature updates, 119–121
  simple heating model, 117–118
SLI (scalable link interface), adding multiple GPUs with, 224
spatial locality
  designing texture caches for graphics with, 116
  heat transfer simulation animation, 125–126
split parallel blocks. *see* parallel blocks, splitting into threads
standard C compiler
  compiling for minimum compute capability, 167–168
  development environment, 18–19
  kernel call, 23–24
start event, 108–110
start_thread(), multiple CPUs, 226–227
stop event, 108–110
streams
  CUDA, overview of, 192
  CUDA, using multiple, 198–205, 208–210
  CUDA, using single, 192–198
  GPU work scheduling and, 205–208
  overview of, 185–186
  page-locked host memory and, 186–192
  summary review, 211
supercomputers, performance gains in, 3
surfactants, environmental devastation of, 10
synchronization
  of events. *see* cudaEventSynchronize()
  of streams, 197–198, 204
  of threads, 219
synchronization, and shared memory
  dot product, 76–87
  dot product optimized incorrectly, 87–90
  overview of, 75
  shared memory bitmap, 90–93
__syncthreads()
  dot product computation, 78–80, 85
  shared memory bitmap using, 90–93
  unintended consequences of, 87–90

T
task parallelism, CPU versus GPU applications, 185
TechniScan Medical Systems, CUDA applications, 9
temperatures
  computing temperature updates, 119–121
  heat transfer simulation, 117–118
  heat transfer simulation animation, 121–125
Temple University research, CUDA applications, 10–11
tex1Dfetch() compiler intrinsic, texture memory, 127–128, 131–132
tex2D() compiler intrinsic, texture memory, 132–133
texture, early days of GPU computing, 5–6
texture memory
  animation of simulation, 121–125
  defined, 115
  overview of, 115–117
  simulating heat transfer, 117–121
  summary review, 137
  two-dimensional, 131–137
  using, 125–131

threadIdx variable
  2D texture memory, 132–133
  dot product computation, 76–77, 85
  dot product computation with atomic locks, 255–256
  GPU hash table implementation, 272
  GPU Julia Set, 52
  GPU ripple using threads, 72–73
  GPU sums of a longer vector, 63–64
  GPU sums of arbitrarily long vectors, 66–67
  GPU vector sums using threads, 61
  histogram kernel using global memory atomics, 179–180
  histogram kernel using shared/global memory atomics, 182–183
  multiple CUDA streams, 200
  ray tracing on GPU, 102
  setting up graphics interoperability, 145
  shared memory bitmap, 91
  temperature update computation, 119–121
  zero-copy memory dot product, 221
threads
  coding with, 38–41
  constant memory and, 106–107
  GPU ripple using, 69–74
  GPU sums of a longer vector, 63–65
  GPU sums of arbitrarily long vectors, 65–69
  GPU vector sums using, 61–63
  hardware limit to number of, 63
  histogram kernel using global memory atomics, 179–181
  incorrect dot product optimization and divergence of, 89
  multiple CPUs, 225–229
  overview of, 59–60
  ray tracing on GPU and, 102–104
  read-modify-write operations, 168–170
  shared memory and. *see* shared memory
  summary review, 94
  synchronizing, 219
threadsPerBlock
  allocating shared memory, 76–77
  dot product computation, 79–87
three-dimensional blocks, GPU sums of a longer vector, 63
three-dimensional graphics, history of GPUs, 4–5
three-dimensional scenes, ray tracing producing 2-D image of, 97
tid variable
  blockIdx.x variable assigning value of, 44
  checking that it is less than N, 45–46
  dot product computation, 77–78
  parallelizing code on multiple CPUs, 40
time, GPU ripple using threads, 72–74
timer, event. *see* cudaEventElapsedTime()
Toolkit, CUDA, 16–18
two-dimensional blocks
  arrangement of blocks and threads, 64
  GPU Julia Set, 51
  GPU ripple using threads, 70
  gridDim variable as, 63
  one-dimensional indexing versus, 44
two-dimensional display accelerators, development of GPUs, 4
two-dimensional texture memory
  defined, 116
  heat transfer simulation, 117–118
  overview of, 131–137

U

ultrasound imaging, CUDA applications for, 9
unified shader pipeline, CUDA Architecture, 7
university, CUDA, 245

V

values
  CPU hash table implementation, 261–267
  GPU hash table implementation, 269–275
  hash table concepts, 259–260
vector dot products. *see* dot product computation
vector sums
  CPU, 39–41
  GPU, 41–46
  GPU sums of arbitrarily long vectors, 65–69
  GPU sums of longer vector, 63–65
  GPU sums using threads, 61–63
  overview of, 38–39, 60–61
verify_table(), GPU hash table, 270
Visual Profiler, NVIDIA, 243–244
Visual Studio C compiler, 18–19

W

warps, reading constant memory with, 106–107
while() loop
  CPU vector sums, 40
  GPU lock function, 253
work scheduling, GPU, 205–208

Z

zero-copy memory
  allocating/using, 214–222
  defined, 214
  performance, 222–223