

Programming in Parallel with CUDA

CUDA is now the dominant language used for programming GPUs; it is one of the most exciting hardware developments of recent decades. With CUDA, you can use a desktop PC for work that would have previously required a large cluster of PCs or access to an HPC facility. As a result, CUDA is increasingly important in scientific and technical computing across the whole STEM community, from medical physics and financial modelling to big data applications and beyond.

This unique book on CUDA draws on the author's passion for and long experience of developing and using computers to acquire and analyse scientific data. The result is an innovative text featuring a much richer set of examples than found in any other comparable book on GPU computing. Much attention has been paid to the C++ coding style, which is compact, elegant and efficient. A code base of examples and supporting material is available online, which readers can build on for their own projects.

RICHARD ANSORGE is Emeritus University Senior Lecturer at the Cavendish Laboratory, University of Cambridge and Emeritus Tutor and Fellow at Fitzwilliam College, Cambridge. He is the author of over 170 peer-reviewed publications and co-author of the book *The Physics and Mathematics of MRI* (2016).

Cambridge University Press & Assessment
978-1-108-47953-0 — Programming in Parallel with CUDA
Richard Ansorge
Frontmatter
[More Information](#)

Programming in Parallel with CUDA

A Practical Guide

Richard Ansorge



CAMBRIDGE
UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom

One Liberty Plaza, 20th Floor, New York, NY 10006, USA

477 Williamstown Road, Port Melbourne, VIC 3207, Australia

314–321, 3rd Floor, Plot 3, Splendor Forum, Jasola District Centre, New Delhi – 110025, India

103 Penang Road, #05–06/07, Visioncrest Commercial, Singapore 238467

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning, and research at the highest international levels of excellence.

www.cambridge.org

Information on this title: www.cambridge.org/9781108479530

DOI: 10.1017/9781108855273

© Richard Ansorge 2022

This publication is in copyright. Subject to statutory exception and to the provisions of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of Cambridge University Press.

First published 2022

Printed in the United Kingdom by TJ Books Limited, Padstow Cornwall

A catalogue record for this publication is available from the British Library.

ISBN 978-1-108-47953-0 Hardback

Cambridge University Press has no responsibility for the persistence or accuracy of URLs for external or third-party internet websites referred to in this publication and does not guarantee that any content on such websites is, or will remain, accurate or appropriate.

To Catherine and Lydia

Cambridge University Press & Assessment
978-1-108-47953-0 — Programming in Parallel with CUDA
Richard Ansorge
Frontmatter
[More Information](#)

Contents

| | |
|--|---------------|
| <i>List of Figures</i> | <i>page</i> x |
| <i>List of Tables</i> | xiii |
| <i>List of Examples</i> | xv |
| <i>Preface</i> | xix |
| 1 Introduction to GPU Kernels and Hardware | 1 |
| 1.1 Background | 1 |
| 1.2 First CUDA Example | 2 |
| 1.3 CPU Architecture | 10 |
| 1.4 CPU Compute Power | 11 |
| 1.5 CPU Memory Management: Latency Hiding Using Caches | 12 |
| 1.6 CPU: Parallel Instruction Set | 13 |
| 1.7 GPU Architecture | 14 |
| 1.8 Pascal Architecture | 15 |
| 1.9 GPU Memory Types | 16 |
| 1.10 Warps and Waves | 18 |
| 1.11 Blocks and Grids | 19 |
| 1.12 Occupancy | 20 |
| 2 Thinking and Coding in Parallel | 22 |
| 2.1 Flynn's Taxonomy | 22 |
| 2.2 Kernel Call Syntax | 30 |
| 2.3 3D Kernel Launches | 31 |
| 2.4 Latency Hiding and Occupancy | 37 |
| 2.5 Parallel Patterns | 39 |
| 2.6 Parallel Reduce | 40 |
| 2.7 Shared Memory | 51 |
| 2.8 Matrix Multiplication | 53 |
| 2.9 Tiled Matrix Multiplication | 61 |
| 2.10 BLAS | 65 |
| 3 Warps and Cooperative Groups | 72 |
| 3.1 CUDA Objects in Cooperative Groups | 75 |
| 3.2 Tiled Partitions | 80 |

| | | |
|----------|--|-----|
| 3.3 | Vector Loading | 85 |
| 3.4 | Warp-Level Intrinsic Functions and Sub-warps | 89 |
| 3.5 | Thread Divergence and Synchronisation | 90 |
| 3.6 | Avoiding Deadlock | 92 |
| 3.7 | Coalesced Groups | 96 |
| 3.8 | HPC Features | 103 |
| 4 | Parallel Stencils | 106 |
| 4.1 | 2D Stencils | 106 |
| 4.2 | Cascaded Calculation of 2D Stencils | 118 |
| 4.3 | 3D Stencils | 123 |
| 4.4 | Digital Image Processing | 126 |
| 4.5 | Sobel Filter | 134 |
| 4.6 | Median Filter | 135 |
| 5 | Textures | 142 |
| 5.1 | Image Interpolation | 143 |
| 5.2 | GPU Textures | 144 |
| 5.3 | Image Rotation | 146 |
| 5.4 | The Lerp Function | 147 |
| 5.5 | Texture Hardware | 151 |
| 5.6 | Colour Images | 156 |
| 5.7 | Viewing Images | 157 |
| 5.8 | Affine Transformations of Volumetric Images | 161 |
| 5.9 | 3D Image Registration | 167 |
| 5.10 | Image Registration Results | 175 |
| 6 | Monte Carlo Applications | 178 |
| 6.1 | Introduction | 178 |
| 6.2 | The cuRAND Library | 185 |
| 6.3 | Generating Other Distributions | 196 |
| 6.4 | Ising Model | 198 |
| 7 | Concurrency Using CUDA Streams and Events | 209 |
| 7.1 | Concurrent Kernel Execution | 209 |
| 7.2 | CUDA Pipeline Example | 211 |
| 7.3 | Thrust and cudaDeviceReset | 215 |
| 7.4 | Results from the Pipeline Example | 216 |
| 7.5 | CUDA Events | 218 |
| 7.6 | Disk Overheads | 225 |
| 7.7 | CUDA Graphs | 233 |
| 8 | Application to PET Scanners | 239 |
| 8.1 | Introduction to PET | 239 |
| 8.2 | Data Storage and Definition of Scanner Geometry | 241 |
| 8.3 | Simulating a PET Scanner | 247 |

Contents

ix

| | | |
|-----------|--|-----|
| 8.4 | Building the System Matrix | 259 |
| 8.5 | PET Reconstruction | 262 |
| 8.6 | Results | 266 |
| 8.7 | Implementation of OSEM | 268 |
| 8.8 | Depth of Interaction (DOI) | 270 |
| 8.9 | PET Results Using DOI | 273 |
| 8.10 | Block Detectors | 274 |
| 8.11 | Richardson–Lucy Image Deblurring | 286 |
| 9 | Scaling Up | 293 |
| 9.1 | GPU Selection | 295 |
| 9.2 | CUDA Unified Virtual Addressing (UVA) | 298 |
| 9.3 | Peer-to-Peer Access in CUDA | 299 |
| 9.4 | CUDA Zero-Copy Memory | 301 |
| 9.5 | Unified Memory (UM) | 302 |
| 9.6 | A Brief Introduction to MPI | 313 |
| 10 | Tools for Profiling and Debugging | 325 |
| 10.1 | The gpulog Example | 325 |
| 10.2 | Profiling with nvprof | 330 |
| 10.3 | Profiling with the NVIDIA Visual Profiler (NVVP) | 333 |
| 10.4 | Nsight Systems | 336 |
| 10.5 | Nsight Compute | 338 |
| 10.6 | Nsight Compute Sections | 339 |
| 10.7 | Debugging with Printf | 347 |
| 10.8 | Debugging with Microsoft Visual Studio | 349 |
| 10.9 | Debugging Kernel Code | 352 |
| 10.10 | Memory Checking | 354 |
| 11 | Tensor Cores | 358 |
| 11.1 | Tensor Cores and FP16 | 358 |
| 11.2 | Warp Matrix Functions | 360 |
| 11.3 | Supported Data Types | 365 |
| 11.4 | Tensor Core Reduction | 366 |
| 11.5 | Conclusion | 371 |
| | <i>Appendix A A Brief History of CUDA</i> | 373 |
| | <i>Appendix B Atomic Operations</i> | 382 |
| | <i>Appendix C The NVCC Compiler</i> | 387 |
| | <i>Appendix D AVX and the Intel Compiler</i> | 393 |
| | <i>Appendix E Number Formats</i> | 402 |
| | <i>Appendix F CUDA Documentation and Libraries</i> | 406 |
| | <i>Appendix G The CX Header Files</i> | 410 |
| | <i>Appendix H AI and Python</i> | 435 |
| | <i>Appendix I Topics in C++</i> | 438 |
| | <i>Index</i> | 448 |

Figures

| | | |
|------|--|---------------|
| 1.1 | How to enable OpenMP in Visual Studio | <i>page</i> 6 |
| 1.2 | Simplified CPU architecture | 10 |
| 1.3 | Moore's law for CPUs | 11 |
| 1.4 | Memory caching on 4-core Intel Haswell CPU | 13 |
| 1.5 | Hierarchical arrangement of compute cores in an NVIDIA GTX1080 | 16 |
| 1.6 | GPU memory types and caches | 18 |
| 2.1 | Latency hiding on GPUs | 38 |
| 2.2 | Pairwise reduction for the last 16 elements of x | 40 |
| 2.3 | Tiled matrix multiplication | 62 |
| 2.4 | Performance of matrix multiplication on an RTX 2070 GPU | 69 |
| 3.1 | Performance of the reduction kernels on a Turing RTX 2070 GPU | 88 |
| 3.2 | Performance differences between reduce kernels | 88 |
| 3.3 | Performance of the <code>reduce_coal_any_vl</code> device function | 102 |
| 4.1 | Performance of 2D 4-point and 9-point stencil codes | 111 |
| 4.2 | Approach to convergence for 512×512 arrays | 115 |
| 4.3 | Typical filters used for digital image processing | 127 |
| 4.4 | Result of filters applied to reference image | 127 |
| 4.5 | Noise reduction using a median filter | 136 |
| 4.6 | Batcher sorting networks for $N = 4$ and $N = 9$ | 138 |
| 4.7 | Modified Batcher network to find median of nine numbers | 138 |
| 5.1 | Pixel and image addressing | 143 |
| 5.2 | Bilinear interpolation for image pixels | 143 |
| 5.3 | Interpolation modes with NVIDIA textures | 145 |
| 5.4 | Image quality after rotation using nearest pixel and bilinear interpolations | 146 |
| 5.5 | Rotations and scaling of test image | 154 |
| 5.6 | Test image at 32×32 resolution | 156 |
| 5.7 | ImageJ dialogue for binary image IO | 158 |
| 5.8 | Affine transformations of a $256 \times 256 \times 256$ MRI head scan | 165 |
| 5.9 | Image registration results | 175 |
| 5.10 | Output from registration program | 176 |
| 6.1 | Calculation of π | 179 |
| 6.2 | 3D Ising model results showing 2D x-y slice at central z | 207 |
| 7.1 | Timelines for three-step pipeline code generated using NVVP | 217 |
| 7.2 | NVVP timelines for the event2 program | 226 |

List of Figures

xi

| | | |
|-------|--|-----|
| 7.3 | Scheme for asynchronous host disk IO | 227 |
| 7.4 | Possible topologies for CUDA graph objects | 234 |
| 8.1 | PET detector showing four rings of 48 detectors | 240 |
| 8.2 | Transverse views of coordinate systems used for PET | 240 |
| 8.3 | Encoding scheme for lines of response in PET scanner | 243 |
| 8.4 | PET (c, r) and (x, y) coordinates | 245 |
| 8.5 | PET detector spot maps for second gamma from LOR | 256 |
| 8.6 | Derenzo Phantom transverse and 3D views and generated counts per LOR | 266 |
| 8.7 | MLEM iteration time as a function of the number of thread blocks | 267 |
| 8.8 | PET reconstruction results for MLEM and OSEM with an RTX 2070 GPU | 269 |
| 8.9 | PET depth of interaction errors | 270 |
| 8.10 | LOR paths in blocked PET detectors | 274 |
| 8.11 | Ray tracing through a coordinate aligned block | 275 |
| 8.12 | Image deblurring using the Richardson–Lucy MLEM method | 290 |
| 9.1 | Topologies of HPC systems with multiple GPUs | 294 |
| 9.2 | CUDA unified virtual memory | 299 |
| 10.1 | NVVP timelines for gpulog example: 100 ms per step | 334 |
| 10.2 | NVVP timelines for gpulog example: 100 μ s per step | 335 |
| 10.3 | NVVP timelines for gpulog example: 2.5 μ s per step | 336 |
| 10.4 | Nsight Systems start-up screen | 337 |
| 10.5 | Nsight Systems timeline display | 338 |
| 10.6 | Timeline from Figure 10.6 expanded by a factor of $\sim 6 \times 10^5$ | 338 |
| 10.7 | Nsight Compute start-up dialog | 339 |
| 10.8 | Profiling results from Nsight Compute | 339 |
| 10.9 | GPU Speed of Light: kernel performance | 340 |
| 10.10 | GPU Speed of Light: roofline plot for two kernels | 340 |
| 10.11 | Compute workload analysis: chart for two kernels | 341 |
| 10.12 | Memory workload analysis: flow chart for <code>gpu_log</code> kernel | 342 |
| 10.13 | Scheduler statistics | 343 |
| 10.14 | Warp state statistics: showing data for two kernels | 343 |
| 10.15 | Instruction statistics: statistics for two kernels | 344 |
| 10.16 | Occupancy: theoretical and achieved values for gpulog program | 346 |
| 10.17 | Source counters: source and SASS code for gpulog program | 347 |
| 10.18 | Preparing a VS-debugging session | 350 |
| 10.19 | Start of VS debugging after pressing F5 | 351 |
| 10.20 | VS debugging at second break point | 352 |
| 10.21 | VS debugging: using Nsight for kernel code | 353 |
| 10.22 | VS CUDA kernel debugging with Nsight plugin | 353 |
| 11.1 | Floating-point formats supported by NVIDIA tensor cores | 359 |

Appendix Figures

| | | |
|-----|--|-----|
| A.1 | ToolKit version 10.2 install directory on Windows 10 | 379 |
| A.2 | CUDA samples directory on Windows 10 | 380 |
| D.1 | Normal scalar and AVX2 eight-component vector multiplication | 394 |

| | |
|--|-----|
| D.2 Visual Studio with ICC installed | 395 |
| E.1 16-bit pattern corresponding to AC05 in hexadecimal | 403 |
| E.2 IEEE 32-bit floating-point format | 405 |
| G.1 Interpretation of 2D array index as Morton and row-major order | 432 |
| G.2 2D array addresses in Morton and row-major order | 432 |

Tables

| | | |
|-----|---|----------------|
| 1.1 | CUDA built-in variables | <i>page</i> 20 |
| 2.1 | Flynn’s taxonomy | 23 |
| 2.2 | Kernel launch configurations for maximum occupancy | 38 |
| 2.3 | Features of GPU generations from Kepler to Ampere | 45 |
| 2.4 | Possible combinations of const and restrict for pointer arguments | 57 |
| 3.1 | Member functions for CG objects | 76 |
| 3.2 | Additional member functions for tiled thread blocks | 80 |
| 3.3 | Warp vote and warp match intrinsic functions | 90 |
| 3.4 | The warp shuffle functions | 91 |
| 3.5 | Return values from warp shuffle functions | 92 |
| 3.6 | Behaviour of synchronisation functions | 92 |
| 3.7 | Results from deadlock kernel in Example 3.8 | 96 |
| 4.1 | Convergence rates for the stencil2D kernel | 115 |
| 4.2 | Accuracy of stencil2D for arrays of size 1024×1024 | 119 |
| 4.3 | Results from cascade method using 4-byte floats and arrays of size 1024×1024 | 123 |
| 4.4 | Performance of 3D kernels for a $256 \times 256 \times 256$ array | 125 |
| 4.5 | Performance of <code>filter9PT</code> kernels on an RTX 2070 GPU | 134 |
| 5.1 | Maximum sizes for CUDA textures | 151 |
| 5.2 | Performance of Examples 5.1–5.3 on an RTX 2070 GPU | 153 |
| 5.3 | Performance of <code>affine3D</code> kernel using an RTX 2070 GPU | 165 |
| 6.1 | Times required for random number generators using an RTX 2070 GPU | 197 |
| 6.2 | Random number distribution functions in C++ and CUDA | 197 |
| 7.1 | CUDA stream and event management functions | 210 |
| 7.2 | C++ <code><threads></code> library | 226 |
| 7.3 | Results from <code>asyncDiskIO</code> example using 1 GB data sets | 232 |
| 7.4 | API functions needed for creation of CUDA graphs via capture | 238 |
| 8.1 | Coordinate ranges for PET simulation | 246 |
| 8.2 | Performance of event generators | 285 |
| 9.1 | CUDA device management functions | 297 |
| 9.2 | Values of the CUDA <code>cudaMemcpyKind</code> flag used with <code>cudaMemcpy</code> functions | 299 |
| 9.3 | CUDA host memory allocation functions | 301 |
| 9.4 | Timing results for CUDA memory management methods | 313 |
| 9.5 | Additional timing measurements using NVPROF | 313 |

| | | |
|------|---|-----|
| 9.6 | MPI version history | 314 |
| 9.7 | Core MPI functions | 316 |
| 9.8 | Additional MPI functions | 320 |
| 10.1 | Tuning the number of thread blocks for the gpulog program | 345 |
| 11.1 | CUDA warp matrix functions | 360 |
| 11.2 | Tensor cores supported data formats and tile dimensions | 366 |
| 11.3 | Tensor core performance | 366 |

Appendix Tables

| | | |
|-----|---|-----|
| A.1 | NVIDIA GPU generations, 2007–2021 | 375 |
| A.2 | NVIDIA GPUs from Kepler to Ampere | 376 |
| A.3 | Evolution of the CUDA toolkit | 378 |
| B.1 | Atomic functions | 383 |
| D.1 | Evolution of the SIMD instruction set on Intel processors | 394 |
| E.1 | Intrinsic types in C++ (for current Intel PCs) | 404 |
| G.1 | The CX header files | 411 |
| G.2 | IO functions supplied by <code>cxbinio.h</code> | 416 |
| G.3 | Possible flags used in <code>cudaTextureDesc</code> | 424 |

Examples

| | | |
|------|---|---------------|
| 1.1 | <code>cpusum</code> single CPU calculation of a sin integral | <i>page</i> 2 |
| 1.2 | <code>ompsum</code> OMP CPU calculation of a sin integral | 4 |
| 1.3 | <code>gpusum</code> GPU calculation of a sin integral | 7 |
| 2.1 | Modifications to Example 1.3 to implement thread-linear addressing | 29 |
| 2.2 | <code>gpu_sin</code> kernel alternative version using a for loop | 30 |
| 2.3 | <code>grid3D</code> using a 3D grid of thread blocks | 31 |
| 2.4 | <code>grid3D_linear</code> thread-linear processing of 3D array | 34 |
| 2.5 | <code>reduce0</code> kernel and associated host code | 41 |
| 2.6 | <code>reduce1</code> kernel using thread-linear addressing | 44 |
| 2.7 | <code>reduce2</code> kernel showing use of shared memory | 46 |
| 2.8 | <code>reduce3</code> kernel permitting non-power of two thread blocks | 48 |
| 2.9 | <code>reduce4</code> kernel with explicit loop unrolling | 49 |
| 2.10 | <code>shared_example</code> kernel showing multiple array allocations | 52 |
| 2.11 | <code>hostmult0</code> matrix multiplication on host CPU | 54 |
| 2.12 | <code>hostmult1</code> showing use of <code>restrict</code> keyword | 56 |
| 2.13 | <code>gpumult0</code> kernel simple matrix multiplication on the GPU | 58 |
| 2.14 | <code>gpumult1</code> kernel using <code>restrict</code> keyword on array arguments | 60 |
| 2.15 | <code>gpumult2</code> kernel using lambda function for 2D array indexing | 61 |
| 2.16 | <code>gputild</code> kernel: tiled matrix multiplication using shared memory | 62 |
| 2.17 | <code>gputild1</code> kernel showing explicit loop unrolling | 65 |
| 2.18 | Host code showing matrix multiplication using cuBLAS | 66 |
| 3.1 | <code>reduce5</code> kernel using <code>syncwarp</code> for device of CC = 7 and higher | 73 |
| 3.2 | <code>coop3D</code> kernel illustrating use of cooperative groups with 3D grids | 77 |
| 3.3 | <code>cgwarp</code> kernel illustrating use of tiled partitions | 79 |
| 3.4 | <code>reduce6</code> kernel using <code>warp_shfl</code> functions | 81 |
| 3.5 | <code>reduce7</code> kernel using solely intra-warp communication | 83 |
| 3.6 | <code>reduce8</code> kernel showing use of <code>cg::reduce</code> warp-level function | 85 |
| 3.7 | <code>reduce7_v1</code> kernel with vector loading | 86 |
| 3.8 | <code>deadlock</code> kernel showing deadlock on thread divergence | 94 |
| 3.9 | <code>deadlock_coalesced</code> revised deadlock kernel using coalesced groups | 97 |
| 3.10 | <code>reduce7_v1_coal</code> kernel which uses subsets of threads in each warp | 98 |
| 3.11 | <code>reduce_coal_any_v1</code> kernel using coalesced groups of any size | 100 |
| 4.1 | <code>stencil2D</code> kernel for Laplace's equation | 107 |
| 4.2 | <code>stencil2D_sm</code> kernel, tiled shared memory version of <code>stencil2d</code> | 112 |

| | | |
|------|--|-----|
| xvi | <i>List of Examples</i> | |
| 4.3 | stencil9PT kernel generalisation of stencil2D using all eight nearest neighbours | 113 |
| 4.4 | reduce_maxdiff kernel for finding maximum difference between two arrays | 115 |
| 4.5 | Modification of Example 4.1 to use array_max_diff | 117 |
| 4.6 | zoomfrom kernel for cascaded iterations of stencil2D | 119 |
| 4.7 | stencil3D kernels (two versions) | 124 |
| 4.8 | filter9PT kernel implementing a general 9-point filter | 128 |
| 4.9 | filter9PT_2 kernel using GPU constant memory for filter coefficients | 130 |
| 4.10 | filter9PT_3 kernel with vector loading to shared memory | 131 |
| 4.11 | sobel6PT kernel based on filter9PT_3 | 135 |
| 4.12 | The device function a_less | 136 |
| 4.13 | median9 device function | 137 |
| 4.14 | batcher9 kernel for per-thread median of nine numbers | 139 |
| 5.1 | Bilinear and nearest device and host functions for 2D image interpolation | 148 |
| 5.2 | rotate1 kernel for image rotation and simple main routine | 149 |
| 5.3 | rotate2 kernel demonstrating image rotation using CUDA textures | 151 |
| 5.4 | rotate3 kernel for simultaneous image rotation and scaling | 154 |
| 5.5 | rotate4 kernel for processing RGBA images | 157 |
| 5.6 | rotate4CV with OpenCV support for image display | 158 |
| 5.7 | affine3D kernel used for 3D image transformations | 163 |
| 5.8 | interp3D function for trilinear interpolation | 166 |
| 5.9 | costfun_sumsq kernel: A modified version of affine3D | 167 |
| 5.10 | The struct paramset used for affine image registration | 169 |
| 5.11 | functor cost_functor for evaluation of image registration cost function | 169 |
| 5.12 | Simple host-based optimiser which uses cost_functor | 171 |
| 5.13 | Image registration main routine fragment showing iterative optimisation process | 173 |
| 6.1 | piH host calculation of π using random sampling | 180 |
| 6.2 | piH2 with faster host RNG | 182 |
| 6.3 | piOMP version | 183 |
| 6.4 | piH4 with cuRand Host API | 186 |
| 6.5 | piH5 with cuRand Host API and pinned memory | 188 |
| 6.6 | piH6 with cudaMemcpyAsync | 188 |
| 6.7 | piG kernel for calculation of π using the cuRand Device API | 193 |
| 6.8 | 3D Ising model setup_randstates and init_spins kernels | 200 |
| 6.9 | 3D Ising 2D model flip_spins kernel | 201 |
| 6.10 | 3D Ising model main routine | 203 |
| 7.1 | Pipeline data processing | 212 |
| 7.2 | event1 program showing use of CUDA events with default stream | 219 |
| 7.3 | event2 program CUDA events with multiple streams | 221 |
| 7.4 | asyncDiskIO program support functions | 227 |
| 7.5 | asyncDiskIO program main routine | 229 |
| 7.6 | CUDA graph program | 234 |
| 8.1 | structs used in fullsim | 248 |
| 8.2 | voxgen kernel for PET event generation | 249 |
| 8.3 | ray_to_cyl device function for tracking gammas to cylinder | 252 |

List of Examples

xvii

| | | |
|------|---|-----|
| 8.4 | <code>find_spot</code> kernel used to compress <code>full_sim</code> results | 257 |
| 8.5 | <code>smPart</code> object with <code>key2lor</code> and <code>lor2key</code> utility functions | 260 |
| 8.6 | <code>smTab</code> structure used for indexing the system matrix | 261 |
| 8.7 | <code>forward_project</code> kernel used for MLEM PET reconstruction | 262 |
| 8.8 | <code>backward_project</code> and <code>rescale</code> kernels | 265 |
| 8.9 | <code>ray_to_cyl_doi</code> and <code>voxgen_doi</code> device functions | 271 |
| 8.10 | <code>ray_to_block</code> device function | 276 |
| 8.11 | <code>ray_to_block2</code> illustrating C++11 lambda function to reduce code duplication | 279 |
| 8.12 | <code>track_ray</code> device function which handles calls to <code>ray_to_block2</code> | 281 |
| 8.13 | <code>voxgen_block</code> kernel for event generation in blocked PET detector | 283 |
| 8.14 | Richardson–Lucy FP and BP device functions | 286 |
| 8.15 | <code>rl_deconv</code> host function | 288 |
| 9.1 | Using multiple GPUs on single host | 295 |
| 9.2 | <code>p2ptest</code> kernel demonstrating P2P operations between two GPUs | 299 |
| 9.3 | Managed memory timing tests <code>reduce_warp_vl</code> kernel and main routine | 303 |
| 9.4 | Managed memory test 0 using <code>cudaMalloc</code> | 305 |
| 9.5 | Managed memory test 1 using <code>cudaMallocHost</code> | 306 |
| 9.6 | Managed memory test 3 using <code>thrust</code> for memory allocation | 307 |
| 9.7 | Managed memory test 5 using <code>cudaHostMallocMapped</code> | 308 |
| 9.8 | Managed memory test 6 using <code>cudaMallocManaged</code> | 310 |
| 9.9 | Extended versions of tests 1 and 5 | 312 |
| 9.10 | Reduction using MPI | 316 |
| 9.11 | Compiling and running an MPI program in Linux | 319 |
| 9.12 | Use of <code>mpialltoall</code> to transpose a matrix | 321 |
| 9.13 | Results of matrix transposition program | 323 |
| 10.1 | <code>gpulog</code> program for evaluation of $\log(1+x)$ | 326 |
| 10.2 | Results of running <code>gpulog</code> on an RTX 2070 GPU | 330 |
| 10.3 | <code>nvprof</code> output for <code>gpulog</code> example | 331 |
| 10.4 | <code>nvprof</code> with <code>cudaProfilerStart</code> and <code>Stop</code> | 332 |
| 10.5 | Checking the return code from a CUDA call | 348 |
| 10.6 | Use of <code>cuda-memcheck</code> | 355 |
| 11.1 | <code>matmult</code> kernel for matrix multiplication with tensor cores | 361 |
| 11.2 | <code>matmultS</code> kernel for matrix multiplication with tensor cores and shared memory | 363 |
| 11.3 | <code>reduceT</code> kernel for reduction using tensor cores | 367 |
| 11.4 | <code>reduce_half_vl</code> kernel for reduction using the FP16 data type | 369 |

Appendix Examples

| | | |
|-----|---|-----|
| B.1 | Use of <code>atomicCAS</code> to implement <code>atomicAdd</code> for <code>ints</code> | 384 |
| B.2 | Use of <code>atomicCAS</code> to implement <code>atomicAdd</code> for <code>floats</code> | 385 |
| C.1 | Build command generated by Visual Studio | 387 |
| D.1 | Comparison of Intel ICC and VS compilers | 395 |
| D.2 | Intel intrinsic functions for AVX2 | 397 |

| | | |
|------|--|-----|
| D.3 | Multithreaded version of D.2 using OpenMP | 399 |
| D.4 | gpusaxpy kernel for comparison with host-based versions | 399 |
| G.1 | Header file <code>cx.h</code> , part 1 | 410 |
| G.2 | Header file <code>cx.h</code> , part 2 | 411 |
| G.3 | Header file <code>cx.h</code> , part 3 | 414 |
| G.4 | Use of <code>cxbinio.h</code> to merge a set of binary files | 416 |
| G.5 | Header file <code>cxbinio.h</code> , part 1 | 418 |
| G.6 | Header file <code>cxtimers.h</code> | 422 |
| G.7 | Header file <code>cxttextures.h</code> , part 1 | 425 |
| G.8 | Header file <code>cxttextures.h</code> , part 2 – class <code>txs2D</code> | 426 |
| G.9 | Header file <code>cxttextures.h</code> , part 3 – class <code>txs3D</code> | 428 |
| G.10 | Header file <code>cxconfun.h</code> | 433 |
| I.1 | Iterators in C++ | 442 |

Preface

This book has been primarily written for people who need lots of computing power, including those engaged in scientific research who need this power to acquire, process, analyse or model their data. People working with medical data who need to process ever-larger data sets and more complicated image data are also likely to find this book helpful.

Complicated and demanding computations are something I have been doing for my entire research career, firstly in experimental high-energy physics and more recently in various applications of medical imaging. The advent of GPU computing is one of the most exciting developments I have yet seen, and one reason for writing this book is to share that excitement with readers.

It seems to be a corollary of Moore's law that the demand for computing power increases to always exceed what is currently available. Since the dawn of the PC age in the early 1980s, vendors have been providing supplementary cards to improve the speed of rendering displays. These cards are now known as graphics processing units or GPUs, and, driven by the demands of the PC gaming industry, they have become very powerful computing engines in their own right. The arrival in 2007 of the NVIDIA CUDA Toolkit for writing software that exploits the power of GPUs for scientific applications was a game changer. Suddenly we got a step up in computing power by a factor of 100 instead of the usual doubling every 18 months or so. Since then, the power of GPUs has also continued to grow exponentially over time, following and even exceeding Moore's law. Thus, knowing how to program GPUs is just as useful today as it was in 2007. In fact, today, if you want to engage with high-performance computing (HPC) perhaps on world-class supercomputers, knowing how to use GPUs is essential.

Up till about 2002 the exponential growth in PC computing power was largely due to increasing clock speeds. However, since then, clock speeds have plateaued at around 3.5 GHz, but the number of cores in a CPU chip has steadily increased. Thus, *parallel programming*, which uses many cooperating cores running simultaneously to share the computing load for a single task, is now essential to get the benefit from modern hardware. GPUs take parallel programming to the next level, allowing thousands or even millions of parallel threads to cooperate in a calculation.

Scientific research is difficult, and competitive, available computing power is often a limiting factor. Speeding up an important calculation by a factor of, say, 200 can be a game changer. A running time of a week is reduced to less than one hour, allowing for same-day analysis of results. A running time of one hour would be reduced to 18 seconds, allowing for exploration of the parameter space of complex models. A running time of seconds is reduced

to milliseconds, allowing for interactive investigation of computer models. This book should be particularly useful to individual researchers and small groups who can equip their own in-house PCs with GPUs and get these benefits. Even groups with good access to large HPC facilities would benefit from very rapid tools on their own desktop machine to explore features of their results.

Of course, this book is also suitable for any reader interested in finding out more about GPUs and parallel programming. Even if you already know a little about the subject, we think you will find studying our coding style and choice of examples rewarding.

To be specific, this book is about programming NVIDIA GPUs in C++. I make no apology for concentrating on a specific vendor's products. Since 2007 NVIDIA have become a dominant force in HPC and, more recently, also AI. This is due to both the cost-effectiveness of their GPUs and, just as importantly, the elegance of the C++-like CUDA language. I know that some scientific programming is still carried out in various dialects of Fortran (including Fortran IV, a language I was very fond of in the early 1980s). But C++ is, in my opinion, more expressive. Fans of Fortran may point out that there is a technical problem with optimising C++ code using pointers, but that problem was overcome in C++11 with the introduction of the `restrict` keyword in C11. This keyword is also supported by modern C++ compilers, and it is used in many of our examples.

The examples are one feature that distinguishes this book from other current books on CUDA. Our examples have been carefully crafted from interesting real-world applications, including physics and medical imaging, rather than the rather basic (and frankly boring) problems often found elsewhere. Another difference between this book and others is that we have taken a lot of care over the appearance of our code, using modern C++ where appropriate, to reduce verbosity while retaining simplicity. I feel this is really important; in my experience most scientific PhD students learn computing by modifying other people's code, and, while much of the CUDA example code currently circulating works, it is far from elegant. This may be because in 2007 CUDA was launched as an extension to C, not C++, and most of the original SDK examples were written in a verbose C style. It is unfortunate that that style still persists in many of the online CUDA tutorials and books. The truth is that CUDA always supported some C++, and nowadays CUDA fully supports up to C++17 (albeit with a few restrictions). In November 2019 the venerable "NVIDIA C Programmers Guide" was renamed the "NVIDIA C++ Programmers Guide", and, although then there was no significant change to the content of the guide, it did signal a change in NVIDIA's attitude to their code, and since 2020 some more advanced uses of C++ have started to appear in the SDK examples.

This book does not aim to teach you C++ from scratch; some basic knowledge of C++ is assumed. However Appendix I discusses some of the C++ features used in our examples. Modern C++ is actually something of a monster, with many newer features to support object-orientated and other high-level programming styles. We do not use such features in this book, as, in our view, they are not appropriate for implementing the algorithmic code we run on GPUs. We also favour template functions over virtual functions.

To get the most out of our book, you will need access to a PC equipped with an NVIDIA GPU supporting CUDA (many of them do). The examples were developed using a Windows 10 PC with a 4-core Intel CPU and an NVIDIA RTX 2070 GPU (costing £480 in 2019). A Linux system is also fine, and all our examples should run without modification. Whatever

system you have, you will need a current version of the (free) NVIDIA CUDA Toolkit. On Windows, you will also need Visual Studio C++ (the free community version is fine). On Linux, gcc or g++ is fine.

Sadly, we cannot recommend CUDA development on macOS, since Apple do not use NVIDIA cards on their hardware and their drivers do not support recent NVIDIA cards. In addition, NVIDIA have dropped support for macOS starting with their Toolkit version 11.0, released in May 2020.

All of the example code can be downloaded from <https://github.com/RichardAns/CUDA-Programs>. This site will also contain errata for the inevitable bugs that some of you may find in my code. By the way, I welcome reader feedback about bugs or any other comments. My email address is real@cam.ac.uk. The site will be maintained, and I also hope to add some additional examples from time to time.

I hope you enjoy reading my book as much as I have enjoyed writing it.

Cambridge University Press & Assessment
978-1-108-47953-0 — Programming in Parallel with CUDA
Richard Ansorge
Frontmatter
[More Information](#)
