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).
Programming in Parallel with CUDA
A Practical Guide

Richard Ansorge
To Catherine and Lydia
# Contents

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>page x</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>xiii</td>
</tr>
<tr>
<td>List of Examples</td>
<td>xv</td>
</tr>
<tr>
<td>Preface</td>
<td>xix</td>
</tr>
</tbody>
</table>

1  **Introduction to GPU Kernels and Hardware**  
   1.1  Background  
   1.2  First CUDA Example  
   1.3  CPU Architecture  
   1.4  CPU Compute Power  
   1.5  CPU Memory Management: Latency Hiding Using Caches  
   1.6  CPU: Parallel Instruction Set  
   1.7  GPU Architecture  
   1.8  Pascal Architecture  
   1.9  GPU Memory Types  
   1.10  Warps and Waves  
   1.11  Blocks and Grids  
   1.12  Occupancy  

2  **Thinking and Coding in Parallel**  
   2.1  Flynn’s Taxonomy  
   2.2  Kernel Call Syntax  
   2.3  3D Kernel Launches  
   2.4  Latency Hiding and Occupancy  
   2.5  Parallel Patterns  
   2.6  Parallel Reduce  
   2.7  Shared Memory  
   2.8  Matrix Multiplication  
   2.9  Tiled Matrix Multiplication  
   2.10  BLAS  

3  **Warps and Cooperative Groups**  
   3.1  CUDA Objects in Cooperative Groups  
   3.2  Tiled Partitions
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

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

Appendix A A Brief History of CUDA 373
Appendix B Atomic Operations 382
Appendix C The NVCC Compiler 387
Appendix D AVX and the Intel Compiler 393
Appendix E Number Formats 402
Appendix F CUDA Documentation and Libraries 406
Appendix G The CX Header Files 410
Appendix H AI and Python 435
Appendix I Topics in C++ 438
Index 448
# Figures

1.1 How to enable OpenMP in Visual Studio | page 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 `reduce_coal_any_vl` device function | 102
4.1 Performance of 2D 4-point and 9-point stencil codes | 111
4.2 Approach to convergence for $512 \times 512$ 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 $32 \times 32$ 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 $\pi$ | 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

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

Appendix Figures

<table>
<thead>
<tr>
<th>Appendix</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>ToolKit version 10.2 install directory on Windows 10</td>
<td>379</td>
</tr>
<tr>
<td>A.2</td>
<td>CUDA samples directory on Windows 10</td>
<td>380</td>
</tr>
<tr>
<td>D.1</td>
<td>Normal scalar and AVX2 eight-component vector multiplication</td>
<td>394</td>
</tr>
</tbody>
</table>
List of Figures

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

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

xiii
List of Tables

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 cxbinio.h 416
G.3 Possible flags used in cudaTextureDesc 424
Examples

1.1 cpusum single CPU calculation of a sin integral \hspace{1cm} page 2
1.2 ompsum OMP CPU calculation of a sin integral \hspace{1cm} 4
1.3 gpusum GPU calculation of a sin integral \hspace{1cm} 7
2.1 Modifications to Example 1.3 to implement thread-linear addressing \hspace{1cm} 29
2.2 gpu_sin kernel alternative version using a for loop \hspace{1cm} 30
2.3 grid3D using a 3D grid of thread blocks \hspace{1cm} 31
2.4 grid3D_linear thread-linear processing of 3D array \hspace{1cm} 34
2.5 reduce0 kernel and associated host code \hspace{1cm} 41
2.6 reduce1 kernel using thread-linear addressing \hspace{1cm} 44
2.7 reduce2 kernel showing use of shared memory \hspace{1cm} 46
2.8 reduce3 kernel permitting non-power of two thread blocks \hspace{1cm} 48
2.9 reduce4 kernel with explicit loop unrolling \hspace{1cm} 49
2.10 shared_example kernel showing multiple array allocations \hspace{1cm} 52
2.11 hostmult0 matrix multiplication on host CPU \hspace{1cm} 54
2.12 hostmult1 showing use of restrict keyword \hspace{1cm} 56
2.13 gpumult0 kernel simple matrix multiplication on the GPU \hspace{1cm} 58
2.14 gpumult1 kernel using restrict keyword on array arguments \hspace{1cm} 60
2.15 gpumult2 kernel using lambda function for 2D array indexing \hspace{1cm} 61
2.16 gputiled kernel: tiled matrix multiplication using shared memory \hspace{1cm} 62
2.17 gputiled1 kernel showing explicit loop unrolling \hspace{1cm} 65
2.18 Host code showing matrix multiplication using cuBLAS \hspace{1cm} 66
3.1 reduce5 kernel using syncwarp for device of CC = 7 and higher \hspace{1cm} 73
3.2 coop3D kernel illustrating use of cooperative groups with 3D grids \hspace{1cm} 77
3.3 cgwarp kernel illustrating use of tiled partitions \hspace{1cm} 79
3.4 reduce6 kernel using warp_shfl functions \hspace{1cm} 81
3.5 reduce7 kernel using solely intra-warp communication \hspace{1cm} 83
3.6 reduce8 kernel showing use of cg::reduce warp-level function \hspace{1cm} 85
3.7 reduce7_vl kernel with vector loading \hspace{1cm} 86
3.8 deadlock kernel showing deadlock on thread divergence \hspace{1cm} 94
3.9 deadlock_coalesced revised deadlock kernel using coalesced groups \hspace{1cm} 97
3.10 reduce7_vl_coal kernel which uses subsets of threads in each warp \hspace{1cm} 98
3.11 reduce_coal_any_vl kernel using coalesced groups of any size \hspace{1cm} 100
4.1 stencil2D kernel for Laplace’s equation \hspace{1cm} 107
4.2 stencil2D_sm kernel, tiled shared memory version of stencil2d \hspace{1cm} 112
List of Examples

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 filter9PT_2 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 \(\pi\) 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 cudaMempcpy/Async 188
6.7 piG kernel for calculation of \(\pi\) 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

8.4 find_spot kernel used to compress full_sim results 257
8.5 smPart object with key2lor and lor2key utility functions 260
8.6 smTab structure used for indexing the system matrix 261
8.7 forward_project kernel used for MLEM PET reconstruction 262
8.8 backward_project and rescale kernels 265
8.9 ray_to_cyl_doi and voxgen_doi device functions 271
8.10 ray_to_block device function 276
8.11 ray_to_block2 illustrating C++11 lambda function to reduce code duplication 279
8.12 track_ray device function which handles calls to ray_to_block2 281
8.13 voxgen_block kernel for event generation in blocked PET detector 283
8.14 Richardson–Lucy FP and BP device functions 286
8.15 rl_deconv host function 288
9.1 Using multiple GPUs on single host 295
9.2 p2ptest kernel demonstrating P2P operations between two GPUs 299
9.3 Managed memory timing tests reduce_warp_vl kernel and main routine 303
9.4 Managed memory test 0 using cudaMalloc 305
9.5 Managed memory test 1 using cudaMallocHost 306
9.6 Managed memory test 3 using thrust for memory allocation 307
9.7 Managed memory test 5 using cudaHostMallocMapped 308
9.8 Managed memory test 6 using cudaMallocManaged 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 mpiAlltoall to transpose a matrix 321
9.13 Results of matrix transposition program 323
10.1 gpulog program for evaluation of log(1+x) 326
10.2 Results of running gpulog on an RTX 2070 GPU 330
10.3 nvprof output for gpulog example 333
10.4 nvprof with cudaProfilerStart and Stop 332
10.5 Checking the return code from a CUDA call 348
10.6 Use of cuda-memcheck 355
11.1 matmulT kernel for matrix multiplication with tensor cores 361
11.2 matmulTS kernel for matrix multiplication with tensor cores and shared memory 363
11.3 reduceT kernel for reduction using tensor cores 367
11.4 reduce_half_vl kernel for reduction using the FP16 data type 369

Appendix Examples

B.1 Use of atomicCAS to implement atomicAdd for ints 384
B.2 Use of atomicCAS to implement atomicAdd for floats 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
## List of Examples

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

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.