

**Computational Science, Engineering and Technology Series: 34**

**Patterns for  
Parallel Programming  
on GPUs**

## **Computational Science, Engineering and Technology Series**

### **Substructuring Techniques and Domain Decomposition Methods**

*Edited by: F. Magoulès*

### **Soft Computing in Civil and Structural Engineering**

*Edited by: B.H.V. Topping and Y. Tsompanakis*

### **Trends in Civil and Structural Engineering Computing**

*Edited by: B.H.V. Topping, L.F. Costa Neves and R.C. Barros*

### **Parallel, Distributed and Grid Computing for Engineering**

*Edited by: B.H.V. Topping and P. Iványi*

### **Trends in Engineering Computational Technology**

*Edited by: M. Papadrakakis and B.H.V. Topping*

### **Trends in Computational Structures Technology**

*Edited by: B.H.V. Topping and M. Papadrakakis*

### **Computational Methods for Acoustics Problems**

*Edited by: F. Magoulès*

### **Mesh Partitioning Techniques and Domain Decomposition Methods**

*Edited by: F. Magoulès*

### **Computational Mechanics using High Performance Computing**

*Edited by: B.H.V. Topping*

### **High Performance Computing for Computational Mechanics**

*Edited by: B.H.V. Topping, L. Lämmer*

### **Parallel and Distributed Processing for Computational Mechanics: Systems and Tools**

*Edited by: B.H.V. Topping*

## **Saxe-Coburg Publications:**

### **Programming Distributed Finite Element Analysis: An Object Oriented Approach**

*R.I. Mackie*

### **Object Oriented Methods and Finite Element Analysis**

*R.I. Mackie*

### **Domain Decomposition Methods for Distributed Computing**

*J. Kruis*

### **Computer Aided Design of Cable-Membrane Structures**

*B.H.V. Topping and P. Iványi*

# **Patterns for Parallel Programming on GPUs**

*Edited by*  
**F. Magoulès**



© Saxe-Coburg Publications, Kippen, Stirling, Scotland

published 2014 by

**Saxe-Coburg Publications**

Civil-Comp Ltd, Dun Eaglais, Station Brae

Kippen, Stirlingshire, FK8 3DY, Scotland

*Saxe-Coburg Publications is an imprint of Civil-Comp Ltd*

Computational Science, Engineering and Technology Series: 34

ISSN 1759-3158

ISBN 978-1-874672-57-9

**British Library Cataloguing in Publication Data**

A catalogue record for this book is available from the British Library

Publisher's production & editorial team: Steven Miller, Jane Tabor, Rosemary Brodie

Printed in Great Britain by Bell & Bain Ltd, Glasgow

# Contents

<b>Foreword by G. Colin de Verdière</b>	<b>xi</b>
<b>Preface by F. Magoulès</b>	<b>xiii</b>
<b>1 Evaluation of State-of-the-Art Parallelizing Compilers Generating CUDA Code for Heterogeneous CPU/GPU Computing</b>	<b>1</b>
J.C. Juega, S. Verdoolaege, A. Cohen, J.I. Gómez, C. Tenllado and F. Catthoor	
1.1 Introduction . . . . .	2
1.2 Related Work . . . . .	3
1.2.1 Performance Modeling and Tuning . . . . .	3
1.2.2 Programming Models . . . . .	3
1.2.3 Polyhedral Compilation . . . . .	4
1.2.4 Automatic Vectorization . . . . .	5
1.3 CUDA Programming Model . . . . .	6
1.4 Polyhedral Model Overview . . . . .	7
1.5 Tools for GPU Code Generation . . . . .	8
1.6 Experimental Results . . . . .	10
1.6.1 Matrix Multiply (gemm) . . . . .	11
1.6.2 Other Benchmarks . . . . .	15
1.6.2.1 bicg . . . . .	20
1.6.2.2 jacobi-2d . . . . .	22
1.6.2.3 mvt . . . . .	23
1.7 Lessons Learned . . . . .	24
<b>2 Data Size and Data Type Dynamic GPU Code Generation</b>	<b>31</b>
H.-P. Charles and V. Lomüller	
2.1 Introduction . . . . .	31
2.1.1 H4H ITEA2 Project . . . . .	33
2.1.2 Scilab Interpreter . . . . .	33
2.2 Code Generation for Accelerators . . . . .	34
2.2.1 CUDA Programming Language . . . . .	34
2.2.2 CUDA and PTX . . . . .	34

2.2.3	OpenCL . . . . .	36
2.2.4	Using Libraries . . . . .	36
2.2.4.1	Linear Algebra Domain . . . . .	36
2.2.4.2	Message Passing Domain . . . . .	37
2.2.4.3	Graphical Libraries . . . . .	37
2.2.4.4	Other Domains . . . . .	38
2.3	deGoal Code Generator Generator . . . . .	39
2.3.1	Neutral but Rich Instruction Set . . . . .	39
2.3.2	Assembly and Expressions Interleaving . . . . .	40
2.3.3	Small Example . . . . .	41
2.3.4	Instructions Meta Information and Fast Code Generation . . . . .	42
2.3.5	Current Status . . . . .	43
2.4	Experimentations . . . . .	43
2.4.1	Experimental Compiette . . . . .	43
2.4.1.1	Reference Implementation . . . . .	44
2.4.1.2	deGoal Implementation . . . . .	45
2.4.1.3	Hardware and Software Setup . . . . .	47
2.4.1.4	Results . . . . .	47
2.4.2	Scilab Integration . . . . .	51
2.5	Related Works . . . . .	51
2.5.1	HMPP . . . . .	51
2.5.2	Scilab . . . . .	52
2.5.3	LLVM . . . . .	52
2.5.3.1	GCD Grand Central Dispatch . . . . .	52
2.5.3.2	Nvidia Compiler . . . . .	52
2.5.4	Ocelot . . . . .	53
2.5.5	Scipy . . . . .	53
2.6	Conclusion . . . . .	53

### **3 High Level GPGPU Programming with Parallel Skeletons** **57**

M. Bourgoïn, E. Chailloux and J.-L. Lamotte		
3.1	Introduction . . . . .	57
3.2	OCaml and SPOC . . . . .	58
3.2.1	OCaml . . . . .	59
3.2.2	GPGPU Programming . . . . .	59
3.2.3	SPOC . . . . .	61
3.2.4	A Small Example . . . . .	62
3.3	Design Patterns and Skeletons . . . . .	63
3.3.1	Example . . . . .	65
3.3.2	Benefits . . . . .	65
3.4	Skeletons Composition . . . . .	67
3.4.1	Back to the Example . . . . .	68

3.4.2	Benefits . . . . .	68
3.5	Related Works . . . . .	68
3.6	Conclusion & Future Work . . . . .	69
3.6.1	Conclusion . . . . .	69
3.6.2	Future Work . . . . .	69
<b>4</b>	<b>Programming GPUs from High Level Data Flow Models</b>	<b>73</b>
	M. Barreteau, R. Barrère and E. Lenormand	
4.1	Introduction . . . . .	73
4.2	Problem and Related Work . . . . .	74
4.3	The Adaptive Beam Forming Application . . . . .	75
4.3.1	Adaptive Beamforming Basic Description . . . . .	76
4.3.2	Description of the ABF Functions . . . . .	77
4.3.3	Corner-Turns . . . . .	79
4.4	Tool Design Flow . . . . .	80
4.4.1	Modeling . . . . .	80
4.4.1.1	Application Capture . . . . .	80
4.4.1.2	Execution Platform Capture . . . . .	80
4.4.2	Parallelisation . . . . .	81
4.4.2.1	Task Parallelism . . . . .	82
4.4.2.2	Data Parallelism . . . . .	83
4.4.2.3	Space and Time Optimisations . . . . .	84
4.4.3	Code Generation . . . . .	85
4.5	Host Code Generation . . . . .	85
4.5.1	Execution Platform-Independent Code . . . . .	86
4.5.1.1	Functional Code . . . . .	86
4.5.1.2	Asynchronous Code . . . . .	87
4.5.1.3	Data Transfer Optimisation . . . . .	89
4.5.1.4	Multiple Command Queues . . . . .	89
4.5.2	Execution Platform-Dependent Optimisations . . . . .	91
4.5.2.1	Execution Platform-Specific Code Generation . . . . .	92
4.5.2.2	Multiple Devices Code Generation . . . . .	92
4.5.2.3	Gridification . . . . .	93
4.5.3	Tooling Support . . . . .	95
4.5.3.1	Code Porting . . . . .	95
4.5.3.2	Debug . . . . .	95
4.5.3.3	Profiling . . . . .	95
4.6	Kernels Generation . . . . .	96
4.7	Code Optimisation Process . . . . .	97
4.7.1	Host Optimisations . . . . .	97
4.7.1.1	Optimised Gridification . . . . .	97
4.7.1.2	Software Pipelining . . . . .	98

4.7.2	Application Profiling . . . . .	98
4.7.3	Kernel Optimisations . . . . .	100
4.8	Results . . . . .	101
4.8.1	Basic OPENCL Code Generation . . . . .	101
4.8.2	Host Optimisation . . . . .	102
4.8.3	Kernel Optimisation . . . . .	103
4.8.4	Complete Optimisation . . . . .	105
4.8.5	Synthesis . . . . .	106
4.9	Conclusion and Perspectives . . . . .	108

## **5 Optimization methodology for Parallel Programming of Homogeneous or Hybrid Clusters** **111**

S. Vialle and S. Contassot-Vivier		
5.1	Motivations and Objectives . . . . .	112
5.1.1	Programming Modern Distributed and Parallel Architectures	112
5.1.2	Benchmark Application . . . . .	113
5.1.3	Experimental Context . . . . .	115
5.2	Interest and Difficulties of Computations and Communications	
	Overlapping . . . . .	116
5.2.1	Decision Criteria to Implement Overlapping . . . . .	116
5.2.2	Attempting to Use Non-Blocking MPI Communications . .	121
5.2.3	Synchronous MPI Communications inside Dedicated Threads	121
5.2.4	Overlapping MPI Communications and GPU Computations	123
5.2.4.1	Natural Overlapping of GPU Computations with Blocking MPI Communications . . . . .	125
5.2.4.2	Inserting the CPU/GPU Data Transfers in the Overlapping Mechanism . . . . .	127
5.2.5	Experimental Comparison and Analysis of the Overlapping Schemes . . . . .	129
5.3	Impact of Optimization Degree in Computing Kernels . . . . .	131
5.3.1	Typical Degrees of Optimization . . . . .	131
5.3.1.1	Optimization of CPU Computing Kernels . . . . .	131
5.3.1.2	Optimization of GPU Computing Kernels . . . . .	133
5.3.2	Experimental Highlighting of the Kernel Optimization . . .	135
5.3.3	Decision Chain for Optimization of Computing Kernels . .	137
5.3.3.1	Technical Criterion . . . . .	137
5.3.3.2	Required Expertise Criterion . . . . .	138
5.3.3.3	Use Context Criterion . . . . .	138
5.3.3.4	Complete Decision Chain . . . . .	139
5.4	Global Experiments and Analysis . . . . .	139
5.4.1	Experimentation Strategy . . . . .	139
5.4.2	Experimental Results . . . . .	140



5.4.2.1	Performance on Multicore CPU Cluster . . . . .	140
5.4.2.2	Performance on GPU Cluster . . . . .	142
5.4.3	Discussion . . . . .	143
5.4.3.1	Assessment of Overlapping Strategy on CPU Clusters . . . . .	143
5.4.3.2	Assessment of Overlapping Strategy on GPU Clusters . . . . .	144
5.4.3.3	Looking for the Most Interesting Solution . . . . .	145
5.5	Conclusion . . . . .	146
<b>6</b>	<b>Program Sequentially, Carefully, and Benefit from Compiler Advances for Parallel Heterogeneous Computing</b>	<b>149</b>
	M. Amini, C. Ancourt, B. Creusillet, F. Irigoien and R. Keryell	
6.1	Introduction . . . . .	150
6.2	Program Structure and Control Flow . . . . .	151
6.2.1	Well-Formed Loops... . . . .	152
6.2.2	... and Loop Bodies . . . . .	153
6.2.3	Testing Error Conditions . . . . .	153
6.2.4	Declaration Scope . . . . .	154
6.2.5	Interprocedural Control Flow . . . . .	155
6.3	Data Structures and Types . . . . .	155
6.3.1	Pointers . . . . .	156
6.3.2	Arrays . . . . .	158
6.3.2.1	Providing and Respecting Array Bounds . . . . .	158
6.3.2.2	Linearization . . . . .	159
6.3.2.3	Successive Array Element References . . . . .	159
6.3.3	Casts . . . . .	159
6.4	Directives and Pragma (OpenMP ...) . . . . .	160
6.5	Using Libraries . . . . .	162
6.5.1	Relying on Efficient Libraries . . . . .	162
6.5.2	Quarantined Functions: Stubs . . . . .	162
6.6	Object Oriented Programming . . . . .	163
6.7	Conclusion . . . . .	164
<b>7</b>	<b>Using MELT to Improve or Explore your GCC-Compiled Source Code</b>	<b>171</b>
	B. Starynkevitch	
7.1	Introduction to MELT . . . . .	171
7.1.1	GCC and MELT . . . . .	171
7.1.2	A Glimpse into <i>Gimple</i> . . . . .	172
7.1.3	MELT Features . . . . .	176
7.2	MELT Usage for HPC Customization . . . . .	177

7.2.1	Traversing GCC Internal Representations with MELT . . . . .	177
7.2.2	Taming HPC Parallelism with GRAPHITE and MELT . . . . .	179
7.2.3	Why HPC needs GCC Extensions and Customizations? . . . . .	181
7.3	Some Hints and Advice for Taming GCC with MELT . . . . .	182
7.3.1	Choosing the Right Passes . . . . .	183
7.3.2	Using the Available Representations . . . . .	184
7.4	Conclusion and Future Work . . . . .	185
<b>8</b>	<b>OpenCL: A Suitable Solution to Simplify and Unify High Performance Computing Developments</b>	<b>187</b>
	J. Passerat-Palmbach and D.R.C. Hill	
8.1	Introduction . . . . .	187
8.2	On the Need for an Abstraction Layer for OpenCL . . . . .	189
8.2.1	OpenCL in Brief . . . . .	189
8.2.2	OpenCL: A Constrained API . . . . .	190
8.3	OpenCL Support on Various Platforms: The FPGAs Case Study . . . . .	190
8.3.1	Source-to-Source Transformation Approach . . . . .	191
8.3.1.1	At the Beginning, there was CUDA . . . . .	191
8.3.1.2	Then comes OpenCL . . . . .	193
8.4	High-Level APIs for OpenCL: Two Philosophies . . . . .	194
8.4.1	Ease OpenCL Development through High-Level APIs . . . . .	194
8.4.1.1	Standard API C++ Wrapper . . . . .	194
8.4.1.2	QtOpenCL . . . . .	195
8.4.1.3	PyOpenCL . . . . .	197
8.4.2	Generating OpenCL Source Code from High-Level APIs . . . . .	198
8.4.2.1	ScalaCL . . . . .	198
8.4.2.2	Aparapi . . . . .	199
8.4.3	A Complete Solution: JavaCL . . . . .	200
8.4.4	Summary Table of the Solutions . . . . .	204
8.5	Perspectives . . . . .	204
8.6	Conclusion . . . . .	205
<b>9</b>	<b>Parallel Preconditioned Conjugate Gradient Algorithm on GPU</b>	<b>209</b>
	F. Andzembe and J. Koko	
9.1	Introduction . . . . .	209
9.2	Conjugate Gradient Algorithm . . . . .	210
9.2.1	Motivation . . . . .	211
9.2.2	Conjugate Gradient . . . . .	212
9.2.3	Preconditioned Conjugate Gradient . . . . .	213
9.3	Preconditioners . . . . .	214
9.3.1	Incomplete Factorizations . . . . .	214

9.3.2	Jacobi Preconditioner . . . . .	215
9.3.3	Approximate Inverses . . . . .	215
9.3.4	SSOR Preconditioner . . . . .	216
9.4	GPU Implementation . . . . .	217
9.4.1	Matrix Storage . . . . .	217
9.4.2	CUDA . . . . .	218
9.5	Numerical Experiments . . . . .	221
9.6	Conclusion . . . . .	223
<b>10</b>	<b>Solving Sparse Linear Systems with CG and GMRES Methods on a GPU and GPU Clusters</b>	<b>227</b>
	R. Couturier and L. Ziane Khodja	
10.1	Introduction . . . . .	227
10.2	Iterative Methods . . . . .	228
10.2.1	Conjugate Gradient Method . . . . .	229
10.2.2	Generalized Minimal Residual Method . . . . .	231
10.3	Sequential Sparse Linear Solvers on a GPU . . . . .	232
10.3.1	GPU Implementation . . . . .	232
10.3.2	Performances on a GPU . . . . .	235
10.4	Parallel Sparse Linear Solvers on GPU Clusters . . . . .	238
10.4.1	Parallel Implementation on GPU Clusters . . . . .	238
10.4.2	Performances on a GPU Cluster . . . . .	241
10.5	Conclusion . . . . .	245
<b>11</b>	<b>Bioinformatics of Non-Coding RNAs and GPUs, A Case Study: Prediction at Large Scale of MicroRNAs in Genomes</b>	<b>249</b>
	F. Tahi, V.D. Tran, S. Tempel and E. Mahé	
11.1	Introduction . . . . .	250
11.2	The World of Non-Coding RNAs . . . . .	251
11.2.1	ncRNA Structure . . . . .	251
11.2.2	Examples of ncRNAs . . . . .	253
11.2.2.1	Transfer RNA . . . . .	253
11.2.2.2	Ribosomal RNA . . . . .	253
11.2.2.3	Small Nucleolar RNA . . . . .	254
11.2.2.4	MicroRNA . . . . .	254
11.3	Bioinformatics and Non-Coding RNAs . . . . .	255
11.3.1	ncRNA Secondary Structure Prediction . . . . .	256
11.3.2	ncRNA Pseudoknot Prediction . . . . .	256
11.3.3	ncRNA Tertiary Structure Prediction . . . . .	257
11.3.4	ncRNA Identification . . . . .	257
11.3.5	ncRNA Structure Comparison . . . . .	258

11.4	EvryRNA Bioinformatics Platform . . . . .	258
11.4.1	<i>P-DCFold</i> Algorithm . . . . .	259
11.4.2	<i>SSCA</i> Algorithm . . . . .	260
11.4.3	<i>Tfold</i> Algorithm . . . . .	261
11.4.4	<i>ncRNAclassifier</i> Algorithm . . . . .	262
11.4.5	<i>BoostSVM</i> Algorithm . . . . .	263
11.4.6	<i>miRNAFold</i> Algorithm . . . . .	264
11.5	Search for miRNA Precursors in Genomes . . . . .	264
11.5.1	Introduction . . . . .	264
11.5.2	<i>miRNAFold</i> Method . . . . .	265
11.5.2.1	The Approach . . . . .	265
11.5.2.2	The Algorithm . . . . .	265
11.5.3	Results . . . . .	267
11.6	Search at Large Scale for miRNA Precursors in Genomes: Use of GPUs . . . . .	267
11.6.1	The GPU Version of <i>miRNAFold</i> Algorithm . . . . .	268
11.6.2	Results Obtained by the GPU Version of <i>miRNAFold</i> . . . . .	270
11.6.3	Conclusion . . . . .	270
11.7	GPU Application in Bioinformatics . . . . .	271
11.7.1	Next Generation Sequencing . . . . .	272
11.7.2	Sequence Identification . . . . .	272
11.7.3	RNA Structure Prediction . . . . .	273

**12 Migrating a Big-Data Grade Application to Large GPU Clusters 281**

D. Tello, V. Ducrot, J.-M. Batto, S. Monot, F. Boumezbeur, V. Arslan and T. Saidani		
12.1	Introduction . . . . .	282
12.1.1	Metagenomics Data Flood . . . . .	282
12.1.2	Selection of Candidate Codes . . . . .	282
12.2	Porting Metaprof to Multi-Core Architectures . . . . .	283
12.2.1	First Optimizations . . . . .	284
12.2.2	Profiling . . . . .	284
12.2.2.1	Hotspots . . . . .	284
12.2.3	Porting to Multi-Core Based Clusters . . . . .	285
12.2.4	Single Node Benchmarks . . . . .	286
12.2.4.1	Scalability . . . . .	286
12.2.4.2	About Bindings . . . . .	288
12.2.4.3	Occupancy . . . . .	289
12.2.4.4	Concurrency . . . . .	289
12.2.4.5	Locks and Waits . . . . .	289
12.3	Porting Metaprof to GPU . . . . .	292
12.3.1	Cuda-MPI v0 Implementation (Shared Memory) . . . . .	292

12.3.1.1	MPI Load Balancing . . . . .	292
12.3.1.2	Results . . . . .	292
12.3.1.3	Limitations . . . . .	294
12.3.2	MetaProf Cuda-MPI v1 . . . . .	294
12.3.2.1	Limiting the Number of Registers . . . . .	294
12.3.2.2	Getting Rid of Memory Constraints . . . . .	295
12.3.2.3	Load Balancing . . . . .	295
12.3.2.4	Transfer/Calculation Overlapping . . . . .	295
12.3.2.5	Improving Memory Access: Shared Memory vs Texture . . . . .	296
12.3.2.6	Results . . . . .	296
12.3.3	MPI-OpenCL Implementation . . . . .	297
12.3.4	Further Optimizations . . . . .	299
12.3.4.1	Input File Format . . . . .	299
12.3.4.2	Further Cuda Optimisations . . . . .	300
12.3.4.3	MPI Processes Interleaving . . . . .	301
12.3.5	Parallelization Assistance Tools . . . . .	301
12.3.5.1	Approach . . . . .	301
12.3.5.2	Results . . . . .	302
12.3.5.3	Conclusion . . . . .	302
12.3.6	Summary . . . . .	302
12.3.7	Power Efficiency . . . . .	302
12.3.7.1	OpenMP Implementation Power Efficiency . . . . .	303
12.3.7.2	MPI+OpenMP Implementation Power Efficiency . . . . .	303
12.3.7.3	GPU Implementation Power Efficiency . . . . .	304
12.3.7.4	Conclusion . . . . .	304
12.3.8	What about Workstations . . . . .	304
12.3.8.1	Scalability . . . . .	305
12.3.8.2	GPU Occupancy Rate . . . . .	305
12.3.8.3	System Global Monitoring . . . . .	305
12.3.8.4	Out of Memory Issue . . . . .	306
12.3.8.5	Influence of Other Parameters . . . . .	307
12.4	Conclusion . . . . .	308

## **13 Testing Random Numbers: When OpenCL is the Right Choice 311**

J.M. Chauvet and E. Mahé

13.1	Introduction . . . . .	311
13.2	Related Works . . . . .	312
13.3	Porting MonoBit and Poker Tests to OpenCL and CUDA . . . . .	313
13.3.1	The MonoBit Test . . . . .	313
13.3.1.1	Choice of the Right Data Format . . . . .	313
13.3.1.2	Working with Fixed Size . . . . .	314

13.3.1.3	OpenCL Implementation . . . . .	314
13.3.2	The Poker Test . . . . .	316
13.3.2.1	OpenCL Implementation . . . . .	317
13.4	Results . . . . .	319
13.4.1	Hardware Configuration . . . . .	319
13.4.2	Serial versus Parallel Code . . . . .	320
13.4.3	Scalability . . . . .	320
13.4.3.1	Monobit Scalability . . . . .	320
13.4.3.2	Poker Scalability . . . . .	321
13.4.4	Choosing the Right Work-Group Size . . . . .	322
13.4.4.1	GPU Work-Group Size . . . . .	322
13.4.4.2	CPU Work-Group Size . . . . .	323
13.4.5	IGP vs CPU . . . . .	324
13.4.6	CUDA vs OpenCL . . . . .	324
13.5	Conclusion and Future Work . . . . .	325

**Index** **327**

# Foreword

Our modern society is increasingly relying on computers. They are ubiquitous in our daily life even if they are hidden in most cases (in appliances, cars, telephones, USB keys, *etc.*). Digital television (TV) would not be the unique broadcasting system today without important compute resources at the recording end and in the display. TV sets nowadays use advanced graphics processing units (GPU) to upscale digital Versatile Disc (DVD) images (not to mention BluRays<sup>®</sup>) or pan and zoom your favorite show. The anti-lock braking system (ABS) of your car is controlled by a computer. The latest three-dimensional scan of a baby would not be possible without computers and powerful GPUs. This list would be too tedious to make exhaustive.

In the scientific and technical world, computers also play an ever growing role, frequently replacing experiments. Numerical simulation has an increasing economical weight in various domains ranging from car crashes to the evaluation of the biological effects of a new pharmaceutical molecule. Simulations have replaced experiments in many cases either because they reduce costs by allowing many variations of experimental conditions (in destructive experiments such as car crashes) or simply because experiments are not feasible (*e.g.* seeing a protein unfolding in which case *in vitro* experimentation has been replaced by “*in silico*” analysis).

In all cases, using simulations help the scientists to understand complex phenomena. Yet this understanding is closely related to the size of the simulation: the finer the simulation, the greater the insight can be. A finer simulation is for example to predict the weather for areas of 10km<sup>2</sup> instead of 100km<sup>2</sup>. As a consequence the larger a simulation is, the more compute capability is required. For example, dividing the size of a cubic simulation by two in each dimension of space increases the number of elements to process by 8 (which means that the processing time also has increased by a factor of 8 or more, depending on the type of simulation). Users (or customers) are not always ready to pay for such an increase in simulation time and they dream of speedier computers solving larger problems in a constant time. Since computers are powered by electricity, deploying more compute power leads to more electrical consumption. This cannot be an endless process for the price of electricity will eventually limit the growth of the supercomputers. Current petaflop machines are in the range of 5MW. With a simple extrapolation, an exaflop machine could be in the order of 100MW or more without a technology change.

To overcome this practical (infrastructure size) and economic limit (*i.e.* the budget to run the installation in the long run), a true disruptive technology is required and is in fact already emerging. The evolution of supercomputers processing elements is parallel to the one of commodity hardware (*e.g.* smartphones): the number of process-

ing units per chip is increasing to deliver greater performances within a constrained energy budget (limit the number of megawatts for the former, longer battery life for the latter). This is especially true of the usage of graphical processing units for general purpose computations (GPGPU).

Envisioned at CEA as early as 2007, the use of GPU computing is becoming widespread (medicine, avionics, HPC) if not mainstream (Apple is harnessing the GPU power through OpenCL and its OS X, your smartphone relies of its GPU to playback videos and so on). GPUs solve part of the equation for a number of applications: they are of an order of magnitude more efficient than regular processors for the same electrical needs, on some algorithms. An increasing number of success stories demonstrates that GPUs are here to stay for a long time in high performance computing (either embedded or not). In that respect, the Titane and Curie machines hosted at CEA (see <http://www-hpc.cea.fr>) are good examples of advanced architectures which have found a community of users for their ability to produce results for the hardest problems yet in a limited time.

The introduction of new hardware compute technologies forces evolutions in the software stack used by programmers. Operating systems, compiler chains as well as libraries are not immune to the nature of the underlying hardware. But the development of a comprehensive software stack takes a significant amount of time. It is therefore very important to start the developments as soon as possible to make sure that the code developers have all the necessary tools for their daily job, when the given technology will spread to the masses. It means also that those developers will have to change their programming habits and adapt themselves to the new technologies.

GPU computing is still a vast field of research which encompasses various domains such as programming languages, compiler technologies, parallelization techniques or linear algebra methods. Such research is fundamental in every field of computer science. They aim to increase the productivity of the developers, yield a better maintainability of the code and reach the best possible performances. The impact of the results achieved is enhanced if the underlying technologies rely on open specifications and are themselves made available open to the whole community of developers and users. Along this line, OpenCL is emerging as the only viable low level technology to drive many forms of compute units, yet in a portable manner.

OpenGPU, a project of the systematic cluster of competitiveness, is clearly at the crossroads of all these needs (such as high performance or portability) and techniques (use of a GPU, CUDA or OpenCL programming). OpenGPU has proved to be a good platform to go forward in the understanding of the pros and cons of GPUs and, at the same time, to promote GPUs' diffusion through the development of a viable ecosystem. This book, entitled *Patterns for Parallel Programming on GPU* edited by Frédéric Magoulès, summarizes some of the achievements of the OpenGPU project. It is an invitation for the reader to deep dive into the fascinating world of GPU programming and usage.

Guillaume Colin de Verdière  
CEA, France



# Preface

At present, multi-core and many-core platforms lead the computer industry, forcing software developers to adopt new programming paradigms, in order to fully exploit their computing capabilities. Graphics Processing Units (GPUs) are one representative of many-core architectures, and certainly the most widespread.

GPU-based application development requires a great effort from application programmers. On one hand, they must take advantage of the massively parallel platform in the problem modeling. On the other, the applications have to make an efficient use of the heterogeneous memory system, managing several levels that are software or hardware controlled. Generally the programmers' methodology consists in evaluating several mapping alternatives, guided by their experience and intuition, which becomes inefficient for software development and maintenance.

In order to help the programmers to make the good choices, this book presents in thirteen chapters some methodologies and design patterns for parallel programming on GPUs. Each chapter, written by different authors, presents a state-of-the-art of some innovative methods, techniques or algorithms, useful for GPU computing. A bibliography is included at the end of each chapter for the reader who wishes to go further.

This book starts at Chapter 1 with an evaluation of state-of-the-art parallelizing compilers generating CUDA code for heterogeneous computing. This chapter evaluates and compares tool frameworks that automatically generate code for GPUs, saving time and effort to programmers. These frameworks take advantage of polyhedral model techniques to exploit parallelism and to satisfy the specific GPU constraints. The authors show the key features of some of these source-to-source compilers and analyze the codes generated. Finally, the authors discuss the importance of some key aspects such as data mapping and code quality.

In Chapter 2, the authors present an unusual strategy to perform dynamic code generation for GPUs. Usual compiler relies on assumption that are not always true, which can lead to sub-optimal code due to a lack of information available to the compiler. By using a code generator called deGoal that can produce a code in a pseudo-assembly code representation for GPUs, the authors show how to dynamically generate code usable by the GPU. This chapter illustrates the tool usage by the matrix multiplication on various configurations. This example show that it allows to develop an application and get near optimal results faster than developping a specialized version.

Chapter 3 shows how to achieve on GPUs great performance with applications commonly handled by CPU only. This hybrid system leads to complex programming designs combining multiple paradigms to manage each hardware architecture. The

authors present how parallel skeletons can help tackle this challenge by abstracting some of these programming designs while automating optimizations. Through a simple example using the OCaml programming language to develop and compose skeletons, this chapter demonstrates how simple modifications in using parallel skeletons can ease GPU programming while offering good performance speed-ups.

Chapter 4 shows how data flow applications can be efficiently programmed on GPUs from a unique high level capture. The authors rely on a tooling approach, through the SPEAR design environment, to point out the underlying productivity gain with regard to performance. For efficient code generation purposes, several optimisations at different levels are detailed, followed by some numerical experiments performed on a representative radar application.

Chapter 5 proposes a study of the optimization process of parallel applications run on modern hybrid architectures. Different optimization schemes are proposed for overlapping computations with communications; and for computation kernels. Development methodologies are introduced to obtain different optimization degrees and specific criteria are defined to help developers to find the most suited degree of optimization according to the application and parallel system considered. Both the performance and code complexity increase are analyzed. This last point is an important issue, as it directly impacts on development and maintenance costs. Experiments are performed to evaluate the different variants of a benchmark application that consists in a dense matrix product.

Porting and maintaining multiple versions of a code base require different skills; and the efforts required in the process, as well as the increased complexity in debugging and testing, are time consuming. Some solutions based on compilers emerge. They are based either on directives added to *C* in *openhmp* or *openacc* or on automatic solutions like *pocc*, *Pluto*, *ppcg*, or *par4all*. However compilers cannot retarget in an efficient way any program written in a low-level language such as unconstrained *C*. Programmers should follow good practices when writing code so that compilers have more room to perform the transformations required for efficient execution on heterogeneous targets.

Chapter 6 explores the impact of different patterns used by programmers, and defines a set of good practices allowing a compiler to generate efficient code.

Chapter 7 introduces the Melt framework and domain-specific language to extend the Gcc compiler. It explains the major internal representations (*Gimple*, *Tree-s*, *etc.*) and the overall organization of Gcc. It shows the major features of Melt and illustrates why extending and customizing the Gcc compiler using Melt is useful; for instance, to use GPUs through OpenCL. It gives some concrete advice and guidelines for the development of such extensions with Melt.

In Chapter 8, the authors study the opportunities that OpenCL offers to high performance computing applications to provide a solution to unify new developments. In order to overcome the lack of native OpenCL support for some architectures, the authors survey the third-party research work that propose a source-to-source approach to transform OpenCL into other parallel programming languages. For instance, FPGAs are considered as a case study, because of their dramatic OpenCL support compared

to GPUs. These transformation approaches could also lead to potential works in the model driven engineering (MDE) field as conceptualized in this chapter. Moreover, OpenCL's standard API is quite rough. Thus, the authors introduce several APIs from the simple high-level binder to the source code generator to ease and boost the development process of any OpenCL application.

In Chapter 9, the authors propose a parallel implementation of the preconditioned conjugate gradient algorithm on a GPU platform. The preconditioning matrix is a first order approximate inverse derived from the SSOR preconditioner. Used through sparse matrix-vector multiplication, the preconditioner proposed by the authors is well-suited to massively parallel architecture like GPUs. As compared to CPU implementation of the conjugate gradient algorithm, the GPU preconditioned conjugate gradient implementation is between eight and sixteen times faster.

In Chapter 10, the authors present the implementation on GPUs of two classical methods for solving sparse linear systems. Those methods are the conjugate gradient method which is usually used for solving symmetric linear systems, and the generalized minimal residual method (GMRES), which is commonly used for solving unsymmetric linear systems. For each method, the authors describe how they have adapted the sequential version for the GPU with the CUDA programming environment. The performances of these parallel algorithms are tested and analyzed on GPU and on CPU clusters.

Non-coding ribonucleic acid (RNA) are functional RNAs that are not translated into proteins. Computational studies of non-coding RNAs have recently become an important challenge in bioinformatics, including their identification and structure prediction. In Chapter 11, the authors present several algorithms for these applications. With the development of next-generation sequencing technologies, huge amounts of genomic and RNA sequences data have been produced, and a parallelization of such tools is required to overcome the long execution time. In this chapter, miRNAFold, an algorithm developed by the authors for the search for microRNAs in genomic sequences, is described together with its implementation in CUDA on GPUs.

Chapter 12 aims to provide both a feedback and a methodological approach on how to port a legacy application to GPU clusters for quantitative metagenomic data, *i.e.*, studying many organisms with whole deoxyribonucleic acid (DNA) information, without getting access to the single and pure species information. The proposed approach is based on the clustering principles that rely on a large correlation matrix computation on polled data. To allow fast computation and get the best of the GPU, many optimizations to cover several optimization strategies are investigated by the authors. This chapter explores dynamic and static optimizations and presents benchmarks on supercomputers and small clusters of GPUs.

Testing random numbers relies on a heavy battery of statistical tests that can take hours of computations or small samplings of numbers. In Chapter 13, the authors demystify this slow checking process by showing the parallel nature of two representative tests and how to implement them on GPU with OpenCL. The authors illustrates these concepts with two examples of test codes translated from C to OpenCL. In addition, an original algorithm, made in the context of testing the quality of the true

random number generator is illustrated, and provides a totally new and safe way of dealing with random number generator.

Naturally, the present book cannot provide a complete record of the many approaches, applications, features and schemes related to GPUs. However, it does provide a useful synopsis of the recent methodologies used in academic circles and in industry to handle efficiently hybrid architectures. This book will be of interest to engineers, computer scientists and applied mathematicians. The editor wishes to thank the authors for their willingness to contribute to this book dedicated to patterns for parallel programming on GPUs.

Frédéric Magoulès  
Ecole Centrale Paris, France