



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF CIVIL ENGINEERING

FAKULTA STAVEBNÍ

INSTITUTE OF STRUCTURAL MECHANICS

ÚSTAV STAVEBNÍ MECHANIKY

**A DYNAMICAL PARTICLE SYSTEM AS A DRIVER
FOR OPTIMAL STATISTICAL SAMPLING**

DYNAMICKÝ ČÁSTICOVÝ SYSTÉM JAKO ÚČINNÝ NÁSTROJ PRO STATISTICKÉ
VZORKOVÁNÍ

DOCTORAL THESIS

DIZERTAČNÍ PRÁCE

AUTHOR

AUTOR PRÁCE

Ing. Jan Mašek

SUPERVISOR

VEDOUCÍ PRÁCE

prof. Ing. Miroslav Vořechovský, Ph.D.

BRNO 2018



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA STAVEBNÍ

Studijní program	P3607 Stavební inženýrství
Typ studijního programu	Doktorský studijní program s prezenční formou studia
Studijní obor	3607V009 Konstrukce a dopravní stavby
Pracoviště	Ústav stavební mechaniky

ZADÁNÍ DISERTAČNÍ PRÁCE

Student	Ing. Jan Mašek
Název	Dynamický částicový systém jako účinný nástroj pro optimální statistické vzorkování
Vedoucí práce	prof. Ing. Miroslav Vořechovský, Ph.D.
Datum zadání	23.1.2018
Datum odevzdání	30.1.2020

V Brně dne 30.7.2018

prof. Ing. Drahomír Novák, DrSc.
Vedoucí ústavu

prof. Ing. Miroslav Bajer, CSc.
Děkan Fakulty stavební VUT

PODKLADY A LITERATURA

- [1] J. M. Hammersley. „Monte Carlo methods for solving multivariable problems“. In: *Annals of the New York Academy of Sciences* 86.1 (1960), pp. 844–874.
- [2] P. Audze and Vilnis Eglajs. „New approach for planning out of experiments“. *Problems of Dynamics and Strengths*, 35: 104–107, 1977. (Rusky).
- [3] M. Johnson, L. Moore, and D. Ylvisaker. „Minimax and maximin distance designs“. In: *Journal of Statistical Planning and Inference* 2.26 (1990), pp. 131–148. ISSN: 0378-3758.
- [4] Thomas J. Santner, Brian J. Williams, and William I. Notz. „The Design and Analysis of Computer Experiments“. *Springer Series in Statistics*. Springer-Verlag New York, 2003. ISBN 978-0-387-95420-2.
- [5] M. Vořechovský and D. Novák. „Correlation control in small sample Monte Carlo type simulations I: A Simulated Annealing approach“. In: *Probabilistic Engineering Mechanics* 24.3 (2009), pp. 452–462. ISSN: 0266-8920.
- [6] J. Eliáš and M. Vořechovský. „Modification of the Audze–Eglajs criterion to achieve a uniform distribution of sampling points“. In: *Advances in Engineering Software* 100 (2016), pp. 82–96.
- [7] Další publikace a informační zdroje podle pokynů školitele

ZÁSADY PRO VYPRACOVÁNÍ

Cílem je práce vyvinutí metodiky a počítačového programu pro tvorbu optimalizovaného vzorku pro počítačové modely, jejichž vstupem jsou náhodné veličiny. Vzorky mají být optimalizovány vzhledem ke schopnosti přesně a s malým rozptylem odhadovat integrály pomocí metod typu Monte Carlo.

Student využije analogie mezi chováním soustavy nabitých částic s odpuzivými silami a tvorbou optimalizovaného návrhu pro statistické vzorkování. Odvodí pohybové rovnice takového systému. Vytvoří počítačový program, který bude dynamickou interakci simulovat za využití GPU – masivně paralelní platformy Nvidia CUDA.

Nastuduje dostupné metody pro tvorbu optimalizovaných vzorků a vyvinutý program použije pro tvorbu návrhů, jejichž účinnost porovná s účinností vzorků získaných vybranými známými postupy. Provede kritické zhodnocení pro úlohy statistické i pravděpodobnostní analýzy úloh stavební mechaniky.

STRUKTURA DISERTAČNÍ PRÁCE

VŠKP vypracujte a rozčleňte podle dále uvedené struktury:

1. Textová část VŠKP zpracovaná podle Směrnice rektora "Úprava, odevzdávání, zveřejňování a uchování vysokoškolských kvalifikačních prací" a Směrnice děkana "Úprava, odevzdávání, zveřejňování a uchování vysokoškolských kvalifikačních prací na FAST VUT" (povinná součást VŠKP).
2. Přílohy textové části VŠKP zpracované podle Směrnice rektora "Úprava, odevzdávání, zveřejňování a uchování vysokoškolských kvalifikačních prací" a Směrnice děkana "Úprava, odevzdávání, zveřejňování a uchování vysokoškolských kvalifikačních prací na FAST VUT" (nepovinná součást VŠKP v případě, že přílohy nejsou součástí textové části VŠKP, ale textovou část doplňují).

Abstract

The presented doctoral thesis aims at development a new efficient tool for optimization of uniformity of point samples. One of use-cases of these point sets is the usage as optimized sets of integration points in statistical analyses of computer models using Monte Carlo type integration. It is well known that the pursuit of uniformly distributed sets of integration points is the only possible way of decreasing the error of estimation of an integral over an unknown function. The tasks of the work concern a survey of currently used criteria for evaluation and/or optimization of uniformity of point sets. A critical evaluation of their properties is presented, leading to suggestions towards improvements in spatial and statistical uniformity of resulting samples. A refined variant of the general formulation of the ϕ_p optimization criterion has been derived by incorporating the periodically repeated design domain along with a scale-independent behavior of the criterion.

Based on a notion of a physical analogy between a set of sampling points and a dynamical system of mutually repelling particles, a hyper-dimensional N-body system has been selected to be the driver of the developed optimization tool. Because the simulation of such a dynamical system is known to be a computationally intensive task, an efficient solution using the massively parallel GPGPU platform Nvidia CUDA has been developed. An intensive study of properties of this complex architecture turned out as necessary to fully exploit the possible solution speedup.

Apart from statistical uniformity, the samples optimized by efficient dynamical simulations also tend to consist of uniform, self-similar point patterns. Furthermore, the desired self-similarity of samples also results in well optimized point layouts in all subspaces of the design domain. Due to the advantageneous formulation of the particle model, the possibility of additional sample size extension one-by-one is inherently possible. The performance of the dynamically optimized samples if used Monte Carlo estimation has proven to be superior to the commonly used sampling methods and comparable to samples optimized by brute-force combinatorial optimization. Due to the efficiency of the parallelized solution, obtaining the dynamically optimized samples requires orders of magnitude lower computational time.

Keywords

Optimization of statistical samples, dynamical particle system, general purpose computing on graphics processing units, Nvidia CUDA.

Abstrakt

Předložená dizertační práce se zabývá vývojem nového výkonného nástroje pro optimalizaci rovnoměrnosti bodových vzorků. Sekvence rovnoměrně rozmístěných bodů nacházejí uplatnění například jako sady integračních bodů při analýze počítačových modelů pomocí integrace metodami typu Monte Carlo. Pokud je zkoumaná funkce považována za neznámou, jedinou cestou, jak snížit horní mez chyby odhadu integrálu, je optimalizace rovnoměrnosti použitých integračních bodů.

Mezi cíle dizertační práce se předně řadí studium současně používaných kritérií pro ohodnocení nebo optimalizaci rovnoměrnosti bodových vzorků. Je předloženo kritické zhodnocení vybraných kritérií rovnoměrnosti. Pro nápravu vybraných nežádoucích vlastností bodových vzorků byla provedena úprava obecně formulovaného optimalizačního kritéria ϕ_p . Po zavedení periodického návrhového prostoru a odvození formulace kritéria, která nezávisí na měřítku úlohy, bylo docíleno soběpodobných bodových vzorků, které jsou statisticky i prostorově rovnoměrné.

Vyvinutý optimalizační algoritmus se opírá o představu o fyzikální podobnosti mezi sadou optimalizovaných bodů a dynamického systému vzájemně se odpuzujících částic. Simulace takového hyper-dimenzionálního částicového systému je ovšem značně výpočetně náročným úkolem. Proto bylo přikročeno k implementaci efektivního řešení pomocí masivně paralelní platformy Nvidia CUDA. Důležitou částí práce bylo proto též intenzivní studium této komplexní architektury, které bylo nezbytné k plnému využití jejího potenciálu.

Bodové vzorky optimalizované pomocí vyvinutého dynamického částicového systému se vyznačují rovnoměrným rozmístěním bodů do soběpodobných vzorů, a to i v dílčích podprostorech nižších dimenzí. Díky výhodné formulaci částicového systému je přirozeně možné provádět dodatečné rozšíření bodového vzorku o další jednotlivé integrační body.

Výsledná účinnost optimalizovaných bodových vzorků při numerické integraci byla prokázána značně vyšší než účinnost běžně užívaných vzorkovacích metod a srovnatelná se vzorky optimalizované kombinatoricky pomocí hrubé síly. Díky efektivitě vyvinutého paralelního řešení je však časová náročnost dynamické optimalizace řádově nižší.

Klíčová slova

Optimalizace statistického vzorkování, dynamický částicový systém, využití grafických procesorů pro řešení obecných algoritmů, Nvidia CUDA.

© 2018 Jan Mašek

Institute of Structural Mechanics
Faculty of Civil Engineering
Brno University of Technology

Typeset by L^AT_EX

MAŠEK, Jan. A dynamical particle system as a driver for optimal statistical sampling. Brno, 2018, 185 p. Doctoral thesis. Brno University of Technology, Faculty Civil Engineering, Department of Structural Mechanics. Supervised by prof. Ing. Miroslav Vořechovský, Ph.D.

Declaration of originality

I declare that I have written the Doctoral Thesis titled “A dynamical particle system as a driver for optimal statistical sampling” independently, under the guidance of the supervisor and using exclusively the technical references and other sources of information cited in the thesis and listed in the comprehensive bibliography at the end of the thesis.

As the author I furthermore declare that, with respect to the creation of this Doctoral Thesis, I have not infringed any copyright or violated anyone’s personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation S 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll., Section 2, Head VI, Part 4.

Brno

.....

author’s signature

Jan Mašek

Acknowledgement

The completion of this work would not be possible without the support that I was receiving from many individuals, to whom I would like to express my gratitude.

I would like to sincerely appreciate the supervising of my work by prof. Ing. Miroslav Vořechovský, Ph.D. I am grateful that I have been given such a freedom to pursue my own ideas and, at the same time, provided with an intense and enthusiastic guidance whenever these ideas turned out to be questionable. In the very same manner, my gratitude goes to doc. Ing. Petr Frantík, Ph.D., my supervisor-specialist and advisor, who supported me immensely long before my doctoral studies. It was his unique fondness to mechanics and programming that determined the future direction of my studies.

Apart from these gentlemen, I would like to thank to the researchers I had the pleasure to closely collaborate with at the Institute of Structural Mechanics. I especially appreciate the valuable discussions and help of doc. Ing. Jan Eliáš, Ph.D. and Ing. Václav Sadílek, Ph.D.

Finally and most importantly, I take this opportunity to express my heartfelt gratitude to my parents. Their genuine, unconditional love and support helped me crucially in times of doubt and allowed me to share my joy in times of success. Without their care and work, my life would be not this simple. Last, and certainly not least, I am grateful to my brother Pavel for being the best friend I can ask for in any situation we encounter.

Thank you.

The results presented in this work were achieved thanks to financial support by:

- Czech Science Foundation under project **GA16-22230S**
- Ministry of Education, Sports and Youth of under Specific University Research projects **FAST-J-16-3486**, **FAST-J-17-4564**, and **FAST-J-18-5254**.

This support is gratefully acknowledged.

Brno

.....

author's signature

Jan Mašek

Contents

1	Introduction to thesis structure	1
I	STATISTICAL SAMPLING	3
2	Introduction to Design of Experiments	5
3	The Monte Carlo method	7
3.1	Approximation of a definite integral	7
3.2	Estimation of properties of functions of random vectors	9
3.3	General requirements on point samples	12
4	Measures of uniformity and optimization criteria	15
4.1	Capturing sample uniformity	16
4.1.1	Discrepancy measures	16
4.1.2	Distance-based criteria	16
4.2	Variance reduction techniques	18
4.2.1	Lattice samples	18
4.2.2	Low-discrepancy sequences	18
4.2.3	Latin Hypercube sampling	19
4.2.4	Optimization criteria combined with LHS	19
5	Refinement of the ϕ-criterion	21
5.1	Periodic extension of design domain	22
5.2	Selection of value of the distance exponent p	24
5.3	On the similar effect of rising p and increasing number of periodic envelopes	28
6	Physical analogy between the ϕ_p criterion and a dynamical N-body system	31
6.1	Enhancing numerical stability of solution	34
II	NVIDIA COMPUTE UNIFIED DEVICE ARCHITECTURE	39
7	General Purpose Computing on Graphics Processing Units	41
7.1	Architecture of a CPU-GPU system	44
8	Nvidia GPU architecture	49
8.1	CUDA execution model	50

8.1.1	SIMD/SIMT operation	51
8.1.2	Handling conditionally divergent code	52
8.2	CUDA memory model	56
8.2.1	Global memory	57
8.2.2	L1 and L2 caches	58
8.2.3	Accessing global memory	59
8.2.4	Texture memory and texture cache	62
8.2.5	Constant memory	62
8.2.6	Local memory	63
8.2.7	Shared memory	63
8.2.8	Registers	66
8.3	CUDA hardware model	70
8.3.1	Streaming Multiprocessors (SMs)	70
8.3.2	CUDA Cores	75
8.3.3	Double Precision Units	75
8.3.4	Special Function Units	76
8.3.5	Load/Store Units	77
8.3.6	Texture Units	77
8.3.7	Register File	77
8.3.8	Giga-Thread Engine	78
8.3.9	Warp Schedulers	78
8.3.10	Dispatch Units	78
8.3.11	Instruction cache and Instruction buffer	79
8.4	Notes on GPGPU performance	81
8.4.1	Terminology of execution performance	81
8.4.2	Instruction throughput	83
8.4.3	Instruction level parallelism	84
8.4.4	Bounds on execution performance	86
9	Parallel implementation of hyper-dimensional N-body system	89
9.1	Particle simulation algorithms	90
9.1.1	GPGPU computing of particle systems	90
9.2	Requirements on solution implementation	91
9.3	Presented approaches to GPU solution	92
9.3.1	Implementation using on-chip resources	92
9.3.2	Implementation using global memory	94
9.4	Performance of parallelized solution	97
9.4.1	Bounds on parallel solution	97
9.5	Notes on parallel solution	98
III	NUMERICAL EXPERIMENTS	101
10	Selected properties of the dynamically optimized samples	103
11	Numerical integration performance	107
11.1	Approximation of definite integral	109

11.1.1 Discussion of numerical results	110
11.2 Sum of exponentials of normal variables	115
11.2.1 Discussion of numerical results	116
11.3 Engineering example I - Failure of a truss model	121
11.3.1 Discussion of numerical results	122
11.4 Engineering example I - A refined approach	126
11.4.1 Dimension reduction	126
11.4.2 Importance sampling approach	127
11.4.3 Discussion of numerical results	129
11.4.4 Comparison of presented approaches to estimation of p_f	133
11.5 Engineering example II - FEM model of fracture	136
11.5.1 Discussion of numerical results	137
12 Related topics	141
12.1 Fast parallelized approximation of Voronoï diagram	141
12.2 Sample size extension	144
13 Concluding remarks	147
13.1 Future work	149
About author	151

List of Figures

3.1	Monte Carlo approximation of a deterministic integral.	8
3.2	Computation of the mean value μ_{f_i} of random variable X_i	11
4.1	A system of mutually repelling particles within a hypercube $[0, 1]^2$	18
5.1	a) Potential energy of repulsive interaction between a pair of particles. b) Repulsive force acting on particles.	22
5.2	Illustration of periodically repeated planar domain. a) the original two-dimensional design domain with pale colored distances L_{ij} (Equation (5.2)). b) periodically repeated design domain with eight additional images of each particle. Periodic distances \bar{L}_{ij} (Equation (5.7)) are rich colored. c) folding the design domain into a torus is another possible illustration of a periodical domain. Note that the computed distances are not defined on the toroidal surface.	23
5.3	1D example of the periodic extension of level $k_{\max} = 2$	24
5.4	Geometrical illustration of the characteristic length, ℓ_{char}	27
5.5	Convergence of the normalized potential energy $\phi_{(\bar{L}, p)}$ depending on the exponent p . Coloring that designate N_{var} is identical for the two bundles of curves. Solid circles are accompanied by the sample count corresponding to the ℓ_{char} and N_{var}	28
5.6	Disappearing of the quality of a self-similar design and the effect of remedies proposed.	29
5.7	3D designs: a) the original PAE ($p = 2$), b) the corrected potential exponent ($p = N_{\text{var}} + 1$).	30
6.1	Illustration of the optimization process of $N_{\text{sim}}=24$, $N_{\text{var}}=2$. a) initial randomized sample, b) optimized sample.	31
6.2	Interaction of two charged particles.	32
6.3	A possibility of regularization of repulsive forces by an asymptotical softening of the interaction law.	37
6.4	A possibility of regularization of repulsive forces by setting a constant bound (red) or by linearization (blue).	37
6.5	Soft start of the numerical simulation.	37
7.1	A diagram of a CPU-GPU system connected by northbridge.	44
7.2	Connection of CPU and single GPU via motherboard chipset and PCI-express bus.	45
7.3	Connection of CPU with multiple GPUs via multiple PCI-express buses.	46

7.4	Connection of CPU with multiple GPUs mutually connected by a multi-GPU bridge.	46
8.1	Illustration of a two dimensional grid of two dimensional thread blocks.	50
8.2	Assembly code of the SIMD execution of Code 8.3.	53
8.3	Assembly code of the SIMT execution of Code 8.4.	53
8.4	A flow diagram of the management of a group of active threads through a conditionally divergent code.	55
8.5	Examples of coalesced and non-coalesced memory requests.	60
8.6	Examples of various memory loads cached in L1 and L2 cache.	61
8.7	Examples of various memory loads cached only in L2 cache.	61
8.8	Data arrangement into (a) an array of structures or (b) a structure of arrays.	62
8.9	Conflictless shared memory requests serviced by a single memory transaction.	65
8.10	Examples of shared memory bank conflicts and broadcasting of a value requested by multiple threads.	65
8.11	Conflictless arrangement of data in shared memory banks by memory padding.	66
8.12	Register shuffle methods.	68
8.13	Scheme of the Pascal GP100 core.	70
8.14	Scheme of the Kepler GK110 core.	71
8.15	A single Streaming Multiprocessor (SM) of the Pascal GP100 core.	73
8.16	A single Streaming Multiprocessor (SMX) of the Kepler GK110 chip.	74
8.17	Scheme of a single CUDA core.	75
8.18	Analogy between CUDA programming and hardware models.	80
8.19	Pipelined ILP execution.	84
8.20	Bounds on warp throughput.	87
9.1	Solution procedure of the on-chip memory implementation.	92
9.2	Performance comparison of the all-pairs $\mathcal{O}(N^2)$ solution while scaling N_{sim} in constant dimension of $N_{\text{var}} = 2$	96
9.3	a) Performance comparison of the $\mathcal{O}(N^2)$ solution as executed by CPU (dashed) and GPU (solid and dotted lines) while scaling both N_{sim} and N_{var} . b) The achieved speedup of solution. c) Comparison of performance in interactions solved per second.	98
10.1	Optimal ϕ_p designs of $N_{\text{sim}} = 14, N_{\text{var}} = 2$ for various exponents, p , and the intersite Euclidean distance. Note the increased frequency of the shortest distances.	103
10.2	Samples optimized by using the refined ϕ_p criterion by dynamical simulation and LHS-switching.	105
10.3	Rectification of collapsible grids obtained by dynamical simulation using latinization and pre-rotation.	106
11.1	Estimated mean value of the integral J_A (ave, ave \pm ssd).	112
11.2	Estimated standard deviation of the integral J_A (ave, ave \pm ssd).	112
11.3	Estimated mean value of the integral J_B (ave, ave \pm ssd).	113
11.4	Estimated standard deviation of the integral J_B (ave, ave \pm ssd).	113
11.5	Estimated mean value of the integral J_C (ave, ave \pm ssd).	114

11.6	Estimated standard deviation of the integral J_C (ave, ave \pm ssd).	114
11.7	Estimated mean value of $g_{\text{exp},2\text{d}}$ (ave, ave \pm ssd).	117
11.8	Standard deviation of the estimated mean value of $g_{\text{exp},2\text{d}}$	117
11.9	Estimated standard deviation of $g_{\text{exp},2\text{d}}$ (ave, ave \pm ssd).	118
11.10	Sample standard deviation of the estimated standard deviation of $g_{\text{exp},2\text{d}}$	118
11.11	Estimated mean value of $g_{\text{exp},5\text{d}}$ (ave, ave \pm ssd).	119
11.12	Sample standard deviation of the estimated mean value of $g_{\text{exp},5\text{d}}$	119
11.13	Estimated standard deviation of $g_{\text{exp},5\text{d}}$ (ave, ave \pm ssd).	120
11.14	Sample standard deviation of the estimated standard deviation of $g_{\text{exp},5\text{d}}$	120
11.15	An illustration of the studied model of truss structure.	121
11.16	Estimated mean value of deflection w (ave, ave \pm ssd).	123
11.17	Sample standard deviation of the estimation of mean value of deflection w	123
11.18	Estimated standard deviation of deflection w (ave, ave \pm ssd).	124
11.19	Sample standard deviation of the estimation of standard deviation of deflection w	124
11.20	Estimated failure probability p_f (ave, ave \pm ssd).	125
11.21	Sample standard deviation of estimation of failure probability p_f	125
11.22	The process of transformation of optimized samples (blue) using the sampling density of Importance sampling method to achieve a sampling points concentrated in closer proximity of the design point. The position of the design point is highlighted.	129
11.23	a) The probability on the failure surface of the reduced 3d model and the position of the design point in the probability space. b) Results of simulations in sampling points of the IS sample from Figure 11.22c. The color scale of points shows the value of $g_2(\mathbf{X})$ to represent the position of each sampling point with respect to the failure surface.	130
11.24	Estimated failure probability p_f using IS in the original 5d space (ave, ave \pm ssd).	131
11.25	Sample standard deviation of estimation of failure probability p_f using IS in the original 5d space.	131
11.26	Estimated failure probability p_f using IS in the reduced 3d space (ave, ave \pm ssd).	132
11.27	Sample standard deviation of estimation of failure probability p_f using IS in the reduced 3d space.	132
11.28	Monte Carlo approximation: comparison of convergence of estimated failure probability, p_f , in the 5d space of the original problem and the benefit of using IS in 5d and the reduced 3d space (ave, ave \pm ssd).	134
11.29	Random LHS approximation: comparison of convergence of estimated failure probability, p_f , in the 5d space of the original problem and the benefit of using IS in 5d and the reduced 3d space (ave, ave \pm ssd).	134
11.30	DYN ϕ_p Periodic samples: comparison of convergence of estimated failure probability, p_f , in the 5d space of the original problem and the benefit of using IS in 5d and the reduced 3d space (ave, ave \pm ssd).	135
11.31	Latinized DYN ϕ_p Periodic samples: comparison of convergence of estimated failure probability, p_f , in the 5d space of the original problem and the benefit of using IS in 5d and the reduced 3d space (ave, ave \pm ssd).	135
11.32	An illustration of experimental setup of three-point bending of a notched specimen.	136

11.33	Histogram of results of 1200 Monte Carlo simulations that have been used to obtain reference solutions of $P_{\max,0.05}$	138
11.34	Estimation of the 5th percentile of the peak loading force, $P_{\max,0.05}$, using the second lowest value of each 40 simulations (ave, ave \pm ssd).	140
11.35	Estimation of the 5th percentile of the peak loading force, $P_{\max,0.05}$, using 5th percentile of fitted normal and Weibull minimum distributions (ave, ave \pm ssd).	140
12.1	Illustration of convergence of normalized volumes of 25 Voronoï cells.	143
12.2	a) Growth of execution time in dependence on number of Voronoï cells and number of evaluated nodes used for approximation. b) Convergence of normalized volumes of Voronoï cells to the exact solution.	144
12.3	Identifying the largest empty area by evaluating the distribution of potential energy.	146

List of Tables

8.1	Generations of Nvidia GPUs and their Compute capability.	49
8.2	Memory types of contemporary Nvidia GPUs.	57
8.3	Comparison of size and latency of memory types of Nvidia Tesla GP100.	69
8.4	Comparison of hardware properties of Pascal GP100 and Kepler GK100 cores.	72
11.1	Marking of compared optimization methods.	107
11.2	Random variables of truss example and their properties.	121
11.3	Three random variables of the reduced truss system.	126
11.4	Results of sensitivity analysis of the truss model.	127
11.5	Random variables of FEM model example and their properties.	136

Chapter 1

Introduction to thesis structure

The overarching topic of the doctoral thesis aims towards developing a new tool for efficient generation of uniformly distributed samples for statistical approximation methods of Monte Carlo type. The content of this work is divided into three major **Parts**, topics and order of which are intended to form a comprehensive, structured content.

Throughout the **Part I**, the reader is introduced to the field of Design of Experiments. Accented is the importance of conscious planning of configurations of numerical or physical experiments. The attention is devoted to fundamentals of using the Monte Carlo method for statistical approximation of solution of inherently random as well as deterministic problems. The advantageous simplicity of Monte Carlo approximation is set into contrast with the large variance of estimated characteristics and it is argued that a feasible practical use calls for optimized samples that contain *uniformly distributed sampling points* to reduce the error of estimation.

Further, general requirements on statistical samples are drawn and the state of the art of statistical optimization is discussed. The efforts of capturing the level of uniformity of a set of points by comparing its deviance from an uniform distribution (i.e. discrepancy measures) as well as evaluating mutual distances between pairs of points (i.e. distance-based criteria) are presented. The contemporary methods of utilizing these criteria for optimization of uniformity of statistical samples, or any point sets for that matter, are studied and objections towards their certain properties are raised.

The closing portion of Part I concerns (i) refinement of the standard ϕ_p optimization criterion in pursuit of achieving uniform designs with self-similar properties (scale-independent patterns), followed by (ii) the derivation of a physically analogical dynamical system of mutually repelling particles that is proposed to be used for the actual sample optimization. The proposed dynamical system considers, in general, an arbitrary number of bodies interacting with each other within a domain of an arbitrary dimension. Therefore, after observing the computational demands of such a simulation when executed by a conventional single-thread implementation, it was evident that for an efficient study of the developed system, exploiting a kind of a parallel platform is inevitable.

Part II follows, attempting to provide a self-contained block of notes on such a complex topic that is the Nvidia Compute Unified Device Architecture (CUDA). Since the beginning of doctoral studies, the author has consciously studied this massively parallel platform for it has been rightfully considered to be one of key components required for a feasible execution of the demanding simulations. The CUDA platform is a complex structure consisting of

mutually linked layers that are (i) the parallel SIMD/SIMT execution model, (ii) the characteristic CUDA memory hierarchy and (iii) the hardware model of the CUDA-capable GPU core. Part II is also intended as an accessible, coherent text to those interested in the CUDA platform. On the other hand, those interested mainly in the topic of statistical sampling may devote less attention to Part II.

Since its introduction in 2007, the CUDA platform, as well as the entire field of general purpose computing on graphics processing units (GPGPU), is a subject of a lasting, rapid development. Each particular generation of Nvidia devices does have its characteristic properties and limitations, both logical and physical. In general, nevertheless, the basics of the programming model remain similar. Therefore, Part II is treated in an intentionally generalized manner. That way, the content should sustain relevant longer. Although an excessive focus on a particular GPU generation is avoided, characteristic differences between particular generations of hardware are accentuated.

Due to the promising performance attained during solution of challenges of this work, the massively parallel computing has become a recognized tool within the research collective. When considering the future research plans, the usage of the CUDA platform certainly enhances the computing capabilities of the research group.

The content of **Part III** first discusses characteristics of the dynamically optimized samples, proposing their possible post-processing in pursuit to additional enhancement of their properties. The major portion of Part III, however, addresses the study of performance of dynamically optimized samples in various use-cases of Monte Carlo type approximation. Throughout the consciously selected numerical examples, the reader is provided with comparison of output samples from the developed sampling method with other, commonly used optimization methods. The suitability of using the dynamically optimized samples for estimation of characteristics of deterministic and inherently random systems is studied. In addition, engineering problems of estimating the failure probability of a structural model and approximation of characteristics of a complex FEM model are studied. Along the discussed numerical results, comments based on observations of related, elsewhere published or unpublished simulations done by the author are amended where appropriate.

As an ending note of the last part, secondary topics that have been encountered during the development of this work are presented. Both (i) fast parallel approximation of Voronoi diagram as well as (ii) parallel algorithm for search of the largest empty area in the design domain are promising topics on their own and are intended to be examined in further research, rather than considered as part of this work. For an efficient solution of these problems, the gained knowledge and experience of the author with the CUDA platform were utilized.

The closing Chapter 13 of the work summarizes the achieved goals, lessons learned during the studies of statistical sampling, parallel computing platforms and implementing and observing dynamical particle systems. As important are also considered notes on ideas that remained fruitless and, conversely, topics that are considered as perspective for further research.

Part I

STATISTICAL SAMPLING

Chapter 2

Introduction to Design of Experiments

During recent decades, researchers across engineering and research fields acquired a truly paradigm-shifting tool that is the power of contemporary computing hardware. The growth of computing performance of modern-era computers has been long sustained dominantly by the strides in microprocessor manufacturing technology. More recently, as the technology approaches its physical limits [1], advances in the field of distributed and parallel computing are the main driving force that keeps pushing boundaries of computational performance, see e.g. [2, 3] and also Part II of this work.

Due to the formerly unseen power of the contemporary *throughput-oriented* hardware, numerical simulations of vast, detailed models are conducted to complement or even replace costly physical experiments. Quite commonly, the output of these models is in fact a result of a complex deterministic function $g(\mathbf{X})$ of a vector of input variables, \mathbf{X} . The actual simulation of such a model, i.e. the evaluation of $g(\mathbf{X})$ for a given \mathbf{X} might be and typically is a computationally intensive task. Still, running a computer simulation of a complex model for hours or days turns out cheaper than conducting an expensive physical experiment. Furthermore, in case of numerical models that require heavy computational effort, one might attempt to approximate the response of these complex models in a similar sense as these approximate the costly physical experiments. Such subsequent simplification efforts of numerical models that are expensive to solve are commonly recognized as *surrogate models*, *metamodels* or *response surface models*, see e.g. [4]. Based on known outputs from a limited number of *cleverly designed* runs of the more detailed model, a response surface model aims to construct a substitute approximative function of the input vector \mathbf{X} . Such a metamodel is constructed to offer an orders of magnitude faster evaluation in comparison to the approximated model while retaining a reasonably accurate output.

It comes out natural that when conducting physical experiments, simulating a numerical model or constructing a metamodel, to capture the behavior of any modelled system, realizations of a physical or numerical experiment shall (i) be as many as possible in compliance with computational or other costs and (ii) shall be designed with as diverse initial configurations as possible to provide a maximal amount of new information about the behavior of the system.

Efforts to optimize configurations of a finite number of realizations of an experiment in

order to obtain unbiased and accurate results while reducing computational costs are often referred to as Design of Experiments (DoE) [5, 6]. To design a configuration of a planned experiment, one has to set the value of each random variable that is considered by the numerical model or is under control in case of a physical experiment. An underlying assumption for computer experimentation is that these simulations are fully deterministic, i.e. that computing of another realization of a numerical simulation with unchanged inputs always leads to identical results on the output (repeatability). In other words, the behavior of the model does depend only and entirely on a deterministic definition of values of input variables. Such a luxury of designing an unambiguous experimental setup is not very common in physical experimentation, where there are other sources of uncertainties that remain beyond control. Further in this work, design of numerical experiments is concerned.

Considering a finite number of N_{var} independent random variables $X_1, X_2, \dots, X_{N_{\text{var}}}$, that form the input random vector \mathbf{X} of a deterministic model, one may introduce a *design domain* of dimension of N_{var} where a vector \mathbf{x}_i that contains N_{var} orthogonal coordinates (i.e. the experimental setup) of the sampling point \mathbf{x}_i that is the i th realization of numerical simulation. Typically, and also in this work, the design domain is considered to be a unit hypercube $\mathcal{U} = [0, 1]^{N_{\text{var}}}$.

The set of sampling points $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{N_{\text{sim}}}$ is commonly referred to as a statistical *sample* or simply a *design*, from here on denoted \mathcal{D} . The actual results from executed simulations of a numerical model are often used for approximation of statistical moments of the function $g(\mathbf{X})$ that is represented by the model. Generally, such a statistical analysis is conducted in the sense of the Monte Carlo method [7].

In its original definition, the Monte Carlo method considers its set of sampling points to be randomly distributed across the design domain without any optimization whatsoever. Due to this assumption, after running a sufficient number of numerical experiments, an unbiased estimation of the first statistical moment (the mean value) of any function is achieved. The price that is paid for an unbiased mean value is a large variance of such an estimation. Since the Monte Carlo method was introduced, efforts to *reduce the variance* of Monte Carlo-type estimates gave birth to the field of statistical sampling. The task of sampling optimization is the goal of DoE; it will be shown further that the variance of an estimation decreases as design points within a sample are distributed *more uniformly* and therefore have good *space-filling* properties.

Before moving onwards to optimization techniques of statistical sampling, let us discuss the properties of the Monte Carlo method that is inherently present in core of each hierarchical approximation method.

Chapter 3

The Monte Carlo method

Introduced in 1949 by John von Neumann and Stanisław Ulam, see [7] and also e.g. [8, 9], the Monte Carlo method (MC) is a numerical method that serves to solve mathematical problems by means of *random sampling* [10].

The Monte Carlo method is a statistical method that approximates a solution of (i) processes affected by random variables by evaluating the respective probabilistic model. Furthermore, MC allows to solve (ii) processes that do not exhibit any randomness at all by constructing a kind of synthesized probabilistic model.

The actual solution algorithm of the Monte Carlo method is rather simple. A single realization of the probabilistic model is performed as a *random trial*, meaning that each of input variables is set randomly within its respective domain. Such a random simulation is repeated N_{sim} times. Another assumption is that each of N_{sim} simulations is independent of all other trials. Therefore, it is possible to compute the solution approximation as a simple average of all trials. The variance of Monte Carlo method estimates decreases to zero as $1/N_{\text{sim}}$ (or $\mathcal{O}(N_{\text{sim}}^{-1})$), the relative error of solution is proportional to $D/\sqrt{N_{\text{sim}}}$, where D is a constant depending on the used variant of MC method [10]. It is noteworthy that the error of estimation is not dependent on the dimension of the problem, N_{var} . On the other hand, to reduce the error, the number of simulations (= computational effort), must rise quadratically. Due to its computational demands, the so-called naïve Monte Carlo method is not quite suitable for problems that require high accuracy. Despite of these drawbacks, the Monte Carlo method remains a benchmark for other methods due to its unbiased estimates. In what follows, let us examine how the Monte Carlo method is used for solving deterministic and inherently random processes.

3.1 Approximation of a definite integral

Perhaps the most illustrative example of utilization of the Monte Carlo method is an approximation of a deterministic process that is a definite integral. The simplicity of MC allows a rather elegant solution even of complex, hyper-dimensional problems. Suppose that one is interested in computation of the two dimensional area J occupied by the region Ω , see Figure 3.1 bottom left.

Let us assume that the domain containing the circle is given as a unitary plane $[0, 1]^{N_{\text{var}}}$. For a known radius of the circle, r , one is immediately given an exact solution. The goal,

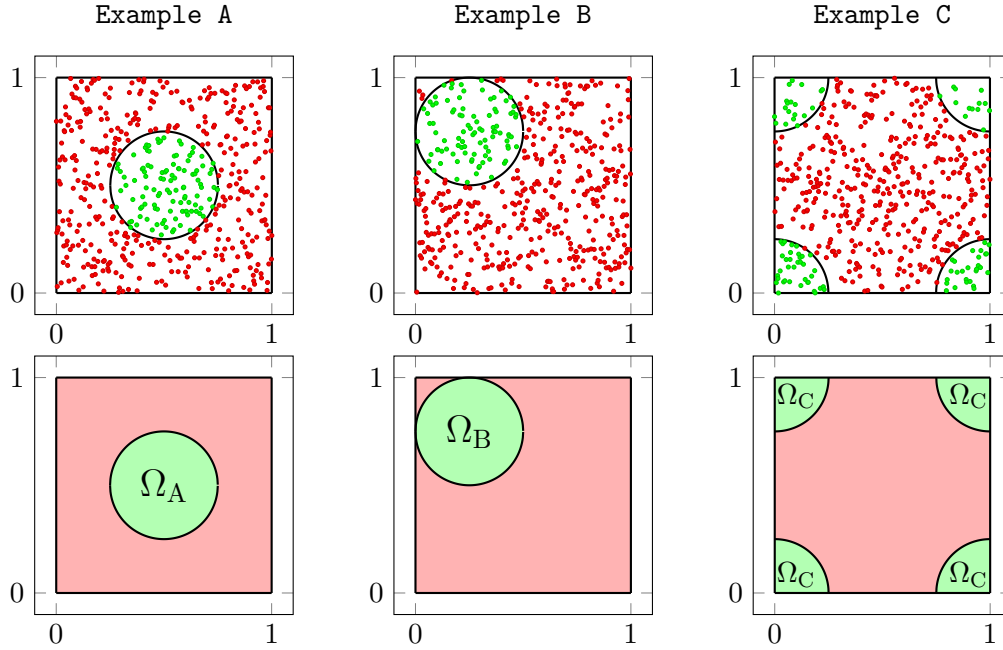


Figure 3.1: Monte Carlo approximation of a deterministic integral.

however, is to approximate this definite integral J numerically by using the Monte Carlo integration. For approximation of this finite area J , MC covers the design domain $[0, 1]^2$ with N_{sim} randomly distributed sampling points. Each of these points is supposed to represent a proportional part of the design domain volume $v = \frac{V}{N_{\text{sim}}} = \frac{1^{N_{\text{var}}}}{N_{\text{sim}}} = \frac{1}{N_{\text{sim}}}$. The approximated volume J_{MC} is then a simple sum of contributions of all trials N' that fell into the region Ω :

$$J_{MC} = \sum_{i=1}^{N'} \frac{1}{N_{\text{sim}}}. \quad (3.1)$$

The constant $\frac{1}{N_{\text{sim}}}$ (volume represented by each sampling point) can be moved in front of the summation and the Equation (3.1) can be interpreted as the ratio of all points that belong into the region Ω and the number of all points, N_{sim} :

$$J \approx J_{MC} = \frac{1}{N_{\text{sim}}} \sum_{\text{sim}=1}^{N_{\text{sim}}} I_{\Omega}(\mathbf{x}_{\text{sim}}), \quad (3.2)$$

where I_{Ω} is an Indicator function, for this purpose defined as:

$$I_{\Omega}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in \Omega \\ 0 & \text{otherwise} \end{cases} \quad (3.3)$$

In a limit case of $N_{\text{sim}} \rightarrow \infty$, the discrete approximation tends to the exact continuous form:

$$\begin{aligned}
 J_{MC} \rightarrow J &= \int_{x_1=0}^1 \int_{x_2=0}^1 I_{\Omega}(\mathbf{x}) \underbrace{dF_{X_1}(x_1) dF_{X_2}(x_2)}_{f_{X_1}(x_1) f_{X_2}(x_2) d\mathbf{x}_1 d\mathbf{x}_2} = \int_0^1 \int_0^1 I_{\Omega}(\mathbf{x}) d\mathbf{x}_1 d\mathbf{x}_2 = \\
 &= \int_0^1 \int_0^1 I_{\Omega}(x_1, x_2) d\mathbf{x}_1 d\mathbf{x}_2 = \int_{\Omega} d\mathbf{x}_1 d\mathbf{x}_2 = J,
 \end{aligned} \tag{3.4}$$

where $F_{X_1} = F_{X_2}$ are distribution functions of uniform variable $\mathcal{U}(0, 1)$ along each marginal. The product of $f_{X_1}(x_1) f_{X_2}(x_2) d\mathbf{x}_1 d\mathbf{x}_2$ is for $N_{\text{sim}} \rightarrow \infty$ equal to an infinitesimal volume.

It is noteworthy that MC assumes that the sampling points are not only randomly distributed, but their distribution is also *statistically uniform*. This means that the probability of appearance of a sampling point is equal for the entire design domain. That way, the result of the discussed approximation shall not statistically depend on the location of the region Ω . A simple *patch test* of statistical uniformity can be conducted by expectation of identical, unbiased results for examples A, B, and C in Figure 3.1.

3.2 Estimation of properties of functions of random vectors

Approximation of properties of functions of random vectors can be computed in a similar fashion as the Monte Carlo method is able to approximate the solution of a deterministic process. The approximated value is typically considered to be a scalar value or a result of a scalar function of an array of outputs of the model.

Let us consider a random vector \mathbf{X} that, instead of its joint probability distribution function $f_{\mathbf{X}}(\mathbf{x})$ is described by its independent continuous random variables $X_1, X_2, \dots, X_{N_{\text{var}}}$ along each marginal. Each of these random variables is described by its own probability density function (PDF):

$$f_{X_i} \equiv f_i(x), \tag{3.5}$$

assuming that $f_i(x)$ is strictly positive ($f_i(x) > 0$, for $x \in \mathbb{R}$) and its integral over its support is equal to one:

$$\int_{-\infty}^{\infty} f_i(x) dx = 1. \tag{3.6}$$

The PDF of each random variable X_i is complemented by its respective cumulative distribution function (CDF):

$$F_{X_i} \equiv F_i(x) = \int_a^x f_i(t) dt = P(X_i < x). \tag{3.7}$$

Commonly evaluated properties of random variables are their respective statistical moments [11]. In a generalized sense, a statistical moment of a random variable of an n th order is defined as follows:

$$\mu_n = \int_{-\infty}^{\infty} (x - c)^n f_i(x) dx, \tag{3.8}$$

where c is defines an offset towards which the moment is calculated. Quite commonly, *central moments* are calculated by substituting c for the mean value of random variable, μ_i :

$$\mu_i = \int_{-\infty}^{\infty} x^i f(x) dx. \quad (3.9)$$

Furthermore, let us introduce a function $g(\mathbf{X}) = g(X_1, X_2, \dots, X_{N_{\text{var}}})$ that transforms the random vector \mathbf{X} into a random variable $Z = g(\mathbf{X})$. The function $g(\mathbf{X})$ might be a kind of analytic function or can be considered to be an unknown operator representing a numerical model, input of which is the random vector \mathbf{X} . The mean value of μ_z can be solved as follows:

$$\mu_z = \mu(g(\mathbf{X})) = \int_{-\infty}^{\infty} z f_Z(z) dz. \quad (3.10)$$

Since the density $f_Z(z)$ is generally not known, the integration is performed over the domain of input vector \mathbf{X} and the transformation is weighed by the joint density function $f_{\mathbf{X}}$:

$$\mu_z = \int_{x_1} \int_{x_2} \dots \int_{x_{N_{\text{var}}}} g(\mathbf{x}) f_{\mathbf{X}}(\mathbf{x}) d\mathbf{x}. \quad (3.11)$$

Due to the independence among variables, one can substitute the product of marginal density functions for the joint density function $f_{\mathbf{X}}(\mathbf{x})$:

$$f_{\mathbf{X}}(\mathbf{x}) = f_{X_1, \dots, X_{N_{\text{var}}}}(x_1, \dots, x_{N_{\text{var}}}) = \prod_{i=1}^{N_{\text{var}}} f_i(x_i). \quad (3.12)$$

Equation (3.10) can be then reformulated to:

$$\mu_z = \int_{-\infty}^{\infty} \dots \int_{-\infty}^{\infty} g(x_1, \dots, x_{N_{\text{var}}}) \prod_{i=1}^{N_{\text{var}}} [f_i(x_i) dx_i], \quad (3.13)$$

and moreover substituting for $f_i(x_i) dx_i = dF_i(x_i)$ results in:

$$\mu_z = \int_{-\infty}^{\infty} \dots \int_{-\infty}^{\infty} g(x_1, \dots, x_{N_{\text{var}}}) \prod_{i=1}^{N_{\text{var}}} dF_i(x_i). \quad (3.14)$$

At this point, it is possible to estimate the above mean value by using the Monte Carlo method. However, this would require sampling of points across the design domain of each marginal x_i , i.e. from negative to positive infinity. Another obstacle is that these sampling points must represent equal weights¹. In this case, each sampling point must represent equal probability $\frac{1}{N_{\text{sim}}} = f_i(x_i) dx_i = dF_i(x_i)$. In other words, the distribution of sampling points along x_i must be uniform with respect to probabilities, not x_i itself. Therefore, yet

¹In the example of a definite integral, these “weights” are the partial volumes represented by each point, $\frac{1}{N_{\text{sim}}}$. That is because the volume is uniformly distributed across the domain, unlike the probability densities $f_i(x_i)$.

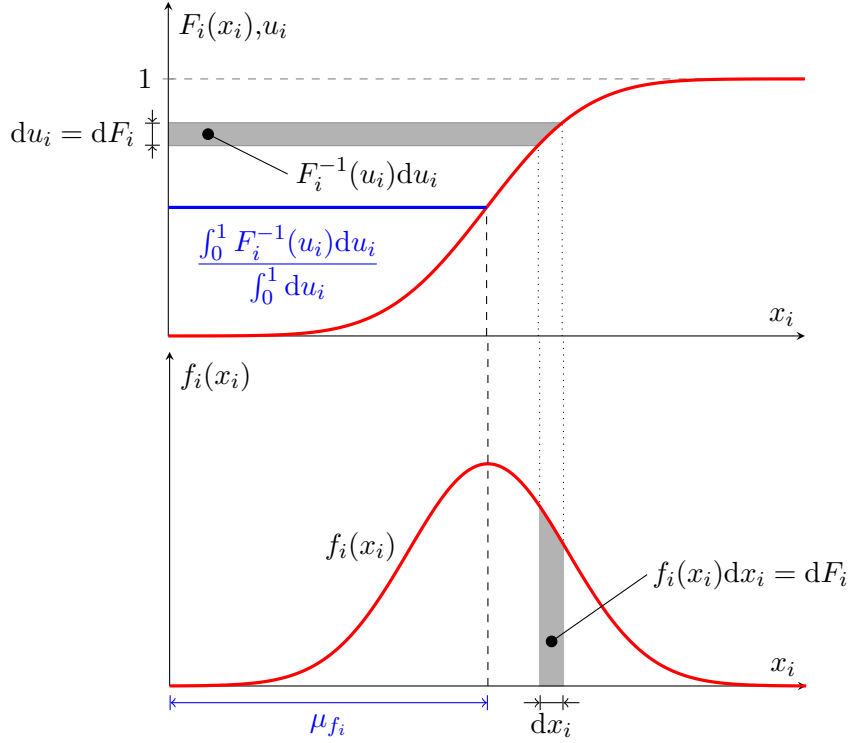


Figure 3.2: Computation of the mean value μ_{f_i} of random variable X_i .

another transformation is proposed by substituting $dF_i(x_i) = du_i$, where u_i is the probability $P(X_i < x)$, explore the Figure 3.2:

$$u_i(x) = \int_{-\infty}^x f_i(t) dt = F_i(x). \quad (3.15)$$

Then, performing a spatially uniform sampling of points in the domain of probabilities $\mathbf{U} = (U_0, U_1, \dots, U_{N_{\text{var}}-1})$ results also in an uniform sampling with respect to probabilities and that is what we have been after. The problem is hence transformed into the domain of *uniformly distributed probabilities* U_i , as oppose to the original space of $X_1, X_2, \dots, X_{N_{\text{sim}}}$:

$$\mu_z = \int_0^1 \dots \int_0^1 g(F_1^{-1}(u_1), \dots, F_{N_{\text{var}}}^{-1}(u_{N_{\text{var}}})) \underbrace{du_1 \dots du_{N_{\text{var}}}}_{d\mathbf{u}^{N_{\text{var}}}}. \quad (3.16)$$

The result of the Monte Carlo approximation using N_{sim} points, the following statistics (average) is obtained to estimate the *expected value* $E[g(\mathbf{X})]$:

$$\mu_z = \mu(g(\mathbf{X})) = E[Z] \approx \frac{1}{N_{\text{sim}}} \sum_{i=1}^{N_{\text{sim}}} g(F_1^{-1}(u_1), \dots, F_{N_{\text{var}}}^{-1}(u_{N_{\text{var}}})) . \quad (3.17)$$

It should be noted that apart from linear functions, one cannot substitute $E[g(\mathbf{X})]$ for $g(E[\mathbf{X}])$.

Approximation of failure probability

Approximation of the failure probability or failure rate using Monte Carlo is similar to the discussed definite integral. The probability of failure, p_f , of the studied system $g(\mathbf{X})$ is essentially equal to the volume of the probability domain $\mathbf{U} = (U_1, \dots, U_{N_{\text{var}}})$ where values of the function $g(F_1^{-1}(u_1), \dots, F_{N_{\text{var}}}^{-1}(u_{N_{\text{var}}}))$ signalize a *failure event*. Similarly to the approximation of a definite integral, a kind of Indicator function:

$$I \left[g \left(F_{\mathbf{X}}^{-1}(\mathbf{u}) \right) \right] = \begin{cases} 1 & \text{if } g \left(F_{\mathbf{X}}^{-1}(\mathbf{u}) \right) \Rightarrow \text{failure event} \\ 0 & \text{otherwise} \end{cases} \quad (3.18)$$

can be integrated over U for an exact solution of p_f :

$$p_f = \int_0^1 \cdots \int_0^1 I \left[g \left(F_{\mathbf{X}}^{-1}(\mathbf{u}) \right) \right] du_1 \dots du_{N_{\text{var}}} = \iint_{D_f} du_1 \dots du_{N_{\text{var}}}, \quad (3.19)$$

where D_f is the failure domain in the unit hypercube. Finally, the approximation in the sense of Monte Carlo can be computed as follows:

$$p_f \approx \frac{1}{N_{\text{sim}}} \sum_{\text{sim}=1}^{N_{\text{sim}}} I \left[g \left(F_{\mathbf{X}}^{-1}(\mathbf{u}_{\text{sim}}) \right) \right]. \quad (3.20)$$

3.3 General requirements on point samples

Even from such a brief introduction to the Monte Carlo method, one can perceive that the distribution of sampling points across the design domain is a crucial factor that influences both convergence and error of Monte Carlo-type approximation. A recognized term that emphasizes the need for *well distributed* sampling points in the design domain is the Koksma-Hlawka inequality [12, 13]:

$$\left| \frac{1}{N_{\text{sim}}} \sum_{i=0}^{N_{\text{sim}}-1} g(u_i) - \int_U g(u) du \right| \leq V(g) D_{N_{\text{sim}}}^*(\mathcal{D}). \quad (3.21)$$

The Koksma-Hlawka inequality sets an upper bound on the error of solution approximation (left hand side) as a product of two independent terms: (i) $V(g)$, which depends on the variation (fluctuation) of the function $g(U)$, see [13], and (ii) $D_{N_{\text{sim}}}^*(\mathcal{D})$ which is the measure of *discrepancy* (i.e. non-uniformity) of the set of sampling points, \mathcal{D} (to be discussed in Chapter 4).

As is assumed, the function $g(U)$ that represents e.g. a numerical model is considered to be unknown or at least beyond control. Therefore, according to Equation (3.21), the only remaining way how to decrease the approximation error (*reduce the variance*) is to ensure that the layout of sampling points is as *uniform* as possible. Immediately though, the problem how to measure uniformity of a point set arises. Various measures of uniformity have been put forth over past years, often with related optimization techniques. Properties of their representatives are discussed in the Chapter 4.

When the positions of sampling points are optimized independently of the simulated model, the single aim of such a sample optimization is to fill the design domain as evenly as possible. Such samples are often referred to as *space-filling* designs, see e.g. [14].

The requirements for sample uniformity should be complemented by pursuing another property that is *noncollapsibility*. A noncollapsible design exhibits N_{sim} unique projections of point coordinates into any subspace of lower dimension than N_{var} . In other words, there should be no pair of identical coordinates along the same axis. That way, it is guaranteed that each sampling point will not provide a redundant information about the studied system. Noncollapsibility is especially important for systems where one or more variables will turn out insignificant. Such a sample is then effectively reduced only to the subspace of remaining significant variables. Therefore it is not desirable for the sample to be spoiled within any subspace due to collapsibility.

The requirement for each sampling point to provide a maximal amount of new information about the system is related to the so-called *D-optimality*, see [15–17]. The D-optimality, or determinant-optimality, considers a *matrix of candidate points* $\boldsymbol{\xi}_{N_{\text{sim}}, N_{\text{var}}}$ that contains coordinates of all sampling points $X_0, X_1, \dots, X_{N_{\text{sim}}-1}$. To be D-optimal, the *design matrix* of the model $\mathbf{X}_{N_{\text{sim}}, N_{\text{var}}}$ must produce such a *dispersion matrix* $(\mathbf{X}'\mathbf{X})^{-1}$ whose determinant is minimized. In other words, the determinant of the *information matrix* $(\mathbf{X}'\mathbf{X})$ is maximized.

Chapter 4

Measures of uniformity and optimization criteria

To be able to operate upon a set of points, let us first establish a suitable distance metric d to measure distances between pairs of points within the design domain. Generally, such a metric shall satisfy following conditions:

- non-negativity: $d(\mathbf{x}_i, \mathbf{x}_j) \geq 0$,
- identity of indiscernibles: $d(\mathbf{x}_i, \mathbf{x}_j) = 0 \Leftrightarrow \mathbf{x}_i = \mathbf{x}_j$,
- symmetry: $d(\mathbf{x}_i, \mathbf{x}_j) = d(\mathbf{x}_j, \mathbf{x}_i)$,
- triangle inequality: $d(\mathbf{x}_i, \mathbf{x}_j) + d(\mathbf{x}_j, \mathbf{x}_k) \geq d(\mathbf{x}_i, \mathbf{x}_k)$.

A commonly used kind of distance metric is the Euclidean distance $L_{ij} = d(\mathbf{x}_i, \mathbf{x}_j)$. It represents the length of a straight line segment that connects the points \mathbf{x}_i and \mathbf{x}_j :

$$L_{ij} = \sqrt{\sum_{v=1}^{N_{\text{var}}} (x_{i,v} - x_{j,v})^2} = \sqrt{\sum_{v=1}^{N_{\text{var}}} (\Delta_{ij,v})^2}, \quad (4.1)$$

where $\Delta_{ij,v}$ is the projection of the connecting line onto the axis v . A generalization of Euclidean distance is the Minkowski distance, see [18, 19]:

$$L_{ij}^m = \sqrt[m]{\sum_{v=1}^{N_{\text{var}}} (x_{i,v} - x_{j,v})^m} = \sqrt[m]{\sum_{v=1}^{N_{\text{var}}} (\Delta_{ij,v})^m}, \quad (4.2)$$

which considers a general positive real value of the exponent m . Such a formulation then encompasses a whole family of distance metrics such as the Manhattan (or taxi cab) distance for $m = 1$, Euclidean distance for $m = 2$ or the Chebyshev distance when $m = \infty$. Any kind of Minkowski distance satisfies the presented requirements on a metric and various metrics have been proposed for evaluation of point sets, see e.g. [20, 21]. However, in the present application, we request the metric to exhibit an additional property that is *isotropy*, i.e. to be independent of the direction of distance measured. Metrics with various values of the exponent m may certainly offer useful properties such as accenting short or long distances. However, the requirement of isotropy narrows the array of suitable metrics solely to the Euclidean metric.

4.1 Capturing sample uniformity

The motivation for efforts to optimize the layout of sampling points within a sample was briefly given throughout previous chapters and particularly in Section 3.3, it was shown that, for an unknown function, the upper bound on numerical integration error can be lowered only by providing the Monte Carlo method with as uniformly distributed sampling points as possible. Without a further explanation, it has been stated that one of the measures for point set uniformity is *discrepancy*, see [22, 23].

4.1.1 Discrepancy measures

The discrepancy criterion aims to express the uniformity of a point sample by evaluating its deviation from an ideally uniform distribution. In other words, discrepancy measures the *irregularity* of a point set. Multiple kinds of discrepancy metrics have been proposed over past years, for instance:

- Star discrepancy and Extreme discrepancy, both [23],
- Modified L_2 discrepancy [24],
- Wrap-Around L_2 discrepancy [25],
- Centered L_2 discrepancy [26].

Each discrepancy measure uses its specific norm for evaluating the deviation of the sample from uniform distribution in all subspaces of the N_{var} -dimensional design domain. A kind of discrepancy metric may be directly used as an objective function for sample optimization efforts. However, the value of discrepancy of a sample does not provide the optimization algorithm with any suggestions how to proceed to decrease the discrepancy.

4.1.2 Distance-based criteria

Another approach how to evaluate the uniformity of a layout of sampling points is to study their mutual distances. An inherent advantage of the distance-based criteria against discrepancy is that these may provide more information about the topology of the sample. For example, one can study the distribution of pair-wise distances or seek for points that violate a particular criterion the most. If identified, such points or pairs of points then might be a subject of a conscious optimization effort.

For example, one can evaluate distances between all pairs of points using a chosen distance metric and search for the pair of points with a minimum or maximum mutual distance:

$$\begin{aligned} L_{\min} &= \min_{\mathbf{x}_i, \mathbf{x}_j \in \mathcal{D}} d(\mathbf{x}_i, \mathbf{x}_j), \\ L_{\max} &= \max_{\mathbf{x}_i, \mathbf{x}_j \in \mathcal{D}} d(\mathbf{x}_i, \mathbf{x}_j). \end{aligned} \tag{4.3}$$

One of natural, visual properties of a “uniformly” distributed set of points is that no pair of points should be situated overly close to each other. Therefore positions of the points that are responsible for the minimal mutual distance L_{\min} should be set farther apart to *maximize* this *minimum distance*. This very idea has been proposed by the Maximin criterion [20].

An inverse notion to the Maximin criterion is the Minimax, also see [20]. It employs the requirement that no pair of points should be overly distant from each other. That way, two points that are responsible for the maximal distance L_{\max} should be brought closer together to *minimize* this *maximum distance*.

It is noteworthy that, under certain conditions, see 4.2. in [20], designs obtained by the Maximin criterion may exhibit D-optimality. Also, the Maximin criterion is known to produce designs appropriate for usage in Kriging, see e.g. [27].

Instead of considering only the extremal mutual distances L_{\min} and L_{\max} , certain distance-based criteria propose to take into account all pair-wise distances. An example of such a family of criteria is the ϕ_p criterion, see [21]. The ϕ_p criterion proposes a scalar-valued criterion function to rank designs of sampling points:

$$\phi_p = \left(\sum_{i=1}^{N_{\text{sim}}-1} \sum_{j=i+1}^{N_{\text{sim}}} \frac{1}{d^p(\mathbf{x}_i, \mathbf{x}_j)} \right)^{\frac{1}{p}}, \quad (4.4)$$

where the exponent p upon the metric $d(\mathbf{x}_i, \mathbf{x}_j)$ is considered as an arbitrary positive integer. The authors propose to use the Manhattan or Euclidean metrics. The ϕ_p criterion proposes to minimize the term (4.4) to reach an optimal design. The value of the criterion can be also further normalized with respect to the number of point pairs in order to represent an “average pair of points”:

$$\phi_p = \left(\frac{1}{\binom{N_{\text{sim}}}{2}} \sum_{i=1}^{N_{\text{sim}}-1} \sum_{j=i+1}^{N_{\text{sim}}} \frac{1}{d^p(\mathbf{x}_i, \mathbf{x}_j)} \right)^{\frac{1}{p}}. \quad (4.5)$$

The advantage of the criteria of ϕ_p family is that the scalar value of the criterion depends on all mutual distances of points while, in a way, retaining the property of the Maximin criterion:

$$\lim_{d^p(\mathbf{x}_i, \mathbf{x}_j) \rightarrow 0} (\phi_p) = \infty. \quad (4.6)$$

Further, with rising the exponent p , the contribution of pairs of closest points will become increasingly dominant. In the limit case of $p = \infty$, the designs that minimize the ϕ_∞ criterion are the designs obtained by the Maximin criterion.

Already in 1977, a predecessor criterion to the ϕ_p was proposed by Audze and Eglājs [28]. The Audze-Eglājs criterion considers the Euclidean distance metric and the value of the exponent $p = 2$. Not only that the Audze-Eglājs criterion is a part of the ϕ_p family, it also proposes a remarkable notion of a physical analogy between a set of sampling points and a set of charged, *mutually repelling particles*, see Figure 4.1 for illustration.

Unlike the ϕ_p criterion function that is only said to be minimized, the Audze-Eglājs criterion understands its value as the amount of *potential energy* stored within a system N_{sim} mutually repelling particles within an N_{var} -dimensional space:

$$E^{AE} = \sum_{i=1}^{N_{\text{sim}}-1} \sum_{j=i+1}^{N_{\text{sim}}} \frac{1}{L_{ij}^2}. \quad (4.7)$$

The abstraction of a hyper-dimensional particle system proposed by the Audze-Eglājs criterion combined with a refinement of the more general ϕ_p criterion will further become one of main topics of this work.

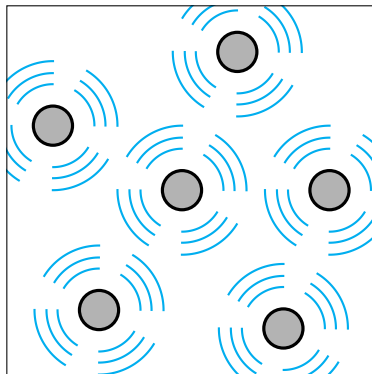


Figure 4.1: A system of mutually repelling particles within a hypercube $[0, 1]^2$.

4.2 Variance reduction techniques

The requirements on non-collapsibility and statistical uniformity of a set of samples were introduced primarily to ensure an unbiased asymptotic convergence of the approximation. However, it has been already pointed out that the Monte Carlo method suffers from large variance of the estimator, order of which is $\mathcal{O}(N_{\text{sim}}^{-1})$. Enhancing the variance of the estimator is a major task that in practice leads to either (i) a decrease in the number of simulations required for reaching sufficiently low estimation error, or (ii) attaining more accurate approximation with the same computational effort.

4.2.1 Lattice samples

A number of *variance reduction techniques* has been proposed since the rise of the Monte Carlo method. The simplest methods are represented by assembling N_{sim} sampling points into various grids: orthogonal, triangular and other patterns or lattice samples. These grids typically require a large number of sampling points to be assembled (slow convergence) and typically exhibit a large degree of collapsibility. Moreover, grids are bound by very strict rules for *sample extension* as it is clearly not possible to extend a grid by an arbitrary number of additional sampling points without violating the pattern. Also, it is not acceptable to perform any translations of points that represent already conducted experiments; these results would be discarded.

4.2.2 Low-discrepancy sequences

A different approach how to achieve a higher accuracy of approximation is to use deterministic *low discrepancy sequences* of integration points, i.e. to perform a *quasi-Monte Carlo* approximation. Generally, low-discrepancy sequences are a family of infinite deterministic sequences of never repeating coordinates of integration points, see e.g. [23, 29]. Coordinates of these points are “judiciously chosen” from the entire design domain to guarantee a small error, i.e. low discrepancy according to Equation (3.21). Examples of these deterministic sequences are, for instance, the Sobol’ [30], Halton [31], Niederreiter [32] and Hammersley [33] sequences.

Indeed, quasi-Monte Carlo methods offer enhancing the probabilistic (average) Monte

Carlo error of $\mathcal{O}(1/\sqrt{N_{\text{sim}}})$ towards $\mathcal{O}(\log(N_{\text{sim}})^{N_{\text{var}}-1}/N_{\text{sim}})$. The latter term is however typically lower than naïve Monte Carlo for high enough N_{sim} . An advantage of low-discrepancy sequences is that usually there are no obstacles in adding new sampling points on top of the already conducted experiments: sampling points can be added one-by-one. Quite conveniently, coordinates of the next points within an infinite sequence are generated, without depreciating former simulations. The main problem with deterministic quasi-Monte Carlo methods and low-discrepancy sequences, respectively, is that these violate the initial assumption of Monte Carlo approximation that is the generation of random samples. Also, low-discrepancy sequences are known to suffer from forming undesired patterns in subspaces for limited number of points, N_{sim} . These patterns tend to disappear only for rather large N_{sim} .

4.2.3 Latin Hypercube sampling

First proposed in [34, 35], the Latin Hypercube sampling method (LHS) aims to redeem the issues of sample collapsibility as well as to reduce the variance of the estimation. The basic idea behind LHS is to divide each dimension (or marginal) of the design domain into N_{sim} subdivisions, each of which can be occupied only by a single sampling point. LHS is therefore a kind of *stratified sampling method*. That way, the entire design domain is divided into $(N_{\text{sim}})^{N_{\text{var}}}$ cells. The actual positions of sampling points within these hyper-cubical LHS sites may be a specific due to the sample purpose. The original suggestion is to place the points within their respective cells randomly, according to a uniform distribution. That way, the assumption of statistical uniformity is met and the estimation remains unbiased. By incorporating any system into cell positions of points, e.g. the mean or median of probability represented by each cell, the mean value of the estimator becomes biased as the statistical uniformity is violated.

By enforcing the LHS rule in each dimension, such a randomized LHS sample provides a reduced variance of $\mathcal{O}(1/N_{\text{sim}})$. Such a variance of estimation holds only if $\text{cov}(g(x_i), g(x_j))$ is negative, which is true whenever $g(X)$ is monotonic along each marginal, see [35, 36].

Possibilities of sample size extension of LHS samples are quite restricted. Without corrupting current sampling points, an LHS sample can be extended only by multiples of current number of points, see e.g. [37] and references therein.

4.2.4 Optimization criteria combined with LHS

The LHS method is often being paired with various optimization criteria in pursuit of a subsequent reduction of its variance and/or improvement of convergence of the estimator, if biased. The simplest approach possible is to use a selected optimization criterion only as an objective function when evaluating randomly generated LHS samples. A typical example of such an “optimization” is the `lhsdesign()` function used in the MATLAB software [38] that generates a set of LHS samples and, according to a user-specified parameter, returns the sample with either (i) the lowest correlation or (ii) the best value of the Maximin criterion. In the specific, although much used, case of Matlab, one should proceed with caution. Matlab merely selects among N_t (user-defined) randomly permuted designs. This approach renders the actual optimization very inefficient, if not entirely unused, see also in [39]

For purposes of an actual optimization, LHS can be suitably coupled with heuristic optimization methods. The changes of positions of points within the sample are typically

done by mutual swapping of coordinates between a pair of points that have been randomly selected or identified by the criterion. For instance, minimization of sample correlation has been proposed [39] by swapping of coordinates of randomly selected points, controlled by the algorithm of simulated annealing [40]. Similar strategy of swapping of LHS coordinates governed by simulated annealing has been suggested in [41] by utilizing the Audze-Eglājs criterion. There, the advantage that the Audze-Eglājs criterion (and the entire family of ϕ_p -criteria, respectively) is able to identify the pair of points that violate the criterion the most has been utilized for choosing the coordinates to be swapped. Also, the notion of using periodically repeated design domain first appeared in [41, 42]. Also coupling genetic algorithms with the Audze-Eglājs criterion [43] has been investigated. An optimization of LHS samples using simulated annealing combined with Maximin or ϕ_p -criterion in [21]. However, it was proposed only to swap randomly selected pairs of coordinates.

The following Chapters 5 and 6 will complement the Part I of this work with the proposed hierarchical refinement of the ϕ_p criterion and the Audze-Eglājs criterion, respectively. Chapter 5 will first present the change towards periodic boundary conditions of the design domain. Further, a study of the influence of the value of the exponent p will be provided along with the derivation of the minimal value of p required for achieving designs of desired qualities.

Chapter 5

Refinement of the ϕ -criterion

It has been mentioned above that the family of ϕ_p optimization criteria does offer the advantageous property of evaluating a sample by investigating mutual distances between all pairs of points. That way, it is able to provide the coupled optimization algorithm not only with the scalar value that evaluates the uniformity of the sample as a whole but also with contribution of each pair of points.

The Audze-Eglājs criterion proceeds even further, as it proposes to understand its value as the amount of potential energy stored within a system of charged, mutually repelling particles:

$$E^{AE} = \sum_{i=1}^{N_{\text{sim}}-1} \sum_{j=i+1}^{N_{\text{sim}}} \frac{1}{L_{ij}^2}. \quad (5.1)$$

To represent potential energy, the Audze-Eglājs criterion omits the exponent of $1/p$ above the whole sum, compare Equations (4.4) and (5.1).

During the recent years, it has been shown that the Audze-Eglājs criterion suffers from existence of boundaries of the design space [41, 42]. A remedy of this behavior was proposed [41, 42], assuming periodically extended design hypercube and thus achieving a design domain without boundaries, see Figure 5.2b. Since then, it has been proved that optimization of point layouts by the introduced Periodic Audze-Eglājs (PAE) criterion combined with LHS switching yields statistically uniform designs (from design to design) and to well distributed set of points in each single point layout, especially in two-dimensional design domain. For higher dimensions, corrupted designs have been observed. Their relatively good uniformity has actually been due to LHS that simply does not allow the malfunctioning criterion to emerge in full effect. Recall this note during Chapter 10, especially when studying Figure 10.2 on page 105.

The following section aims to revisit the periodical extension of the design space. Next, the study of reasons for malfunctioning Audze-Eglājs in dimensions higher than $N_{\text{var}} = 2$ will be conducted, aiming for a remedy that results in obtaining well-distributed, self-similar designs in an arbitrary dimension, N_{var} .

5.1 Periodic extension of design domain

The Audze-Eglājs criterion proposes to use the Euclidean distance metric. This metric is arguably much desired for its property of directional independence, or isotropy. The Euclidean distance between points i and j in N_{var} -dimensional space, L_{ij} , can be expressed as a function of their coordinates:

$$L_{ij} = \sqrt{\sum_{v=1}^{N_{\text{var}}} (x_{i,v} - x_{j,v})^2} = \sqrt{\sum_{v=1}^{N_{\text{var}}} (\Delta_{ij,v})^2}, \quad (5.2)$$

where $\Delta_{ij,v} = |x_{i,v} - x_{j,v}|$ is the difference in their positions projected onto the axis v . Let us assume that the points i and j with their mutual distance L_{ij} are repelled by the force F_{ij} induced by the potential energy E_{ij} :

$$E_{ij}(L_{ij}) = \frac{1}{L_{ij}^p} = \int_{\infty}^{L_{ij}} F_{ij}(x) dx. \quad (5.3)$$

By differentiating the energy potential with respect to the distance, L_{ij} , the *repulsive* force is obtained, also see in Figure 5.1. Neglecting the constant coefficient, the repulsive force is proportional to:

$$F_{ij}(L_{ij}) \propto \frac{1}{L_{ij}^{p+1}}. \quad (5.4)$$

Equation (5.4) can be understood as the constitutive law governing the interaction of particles.

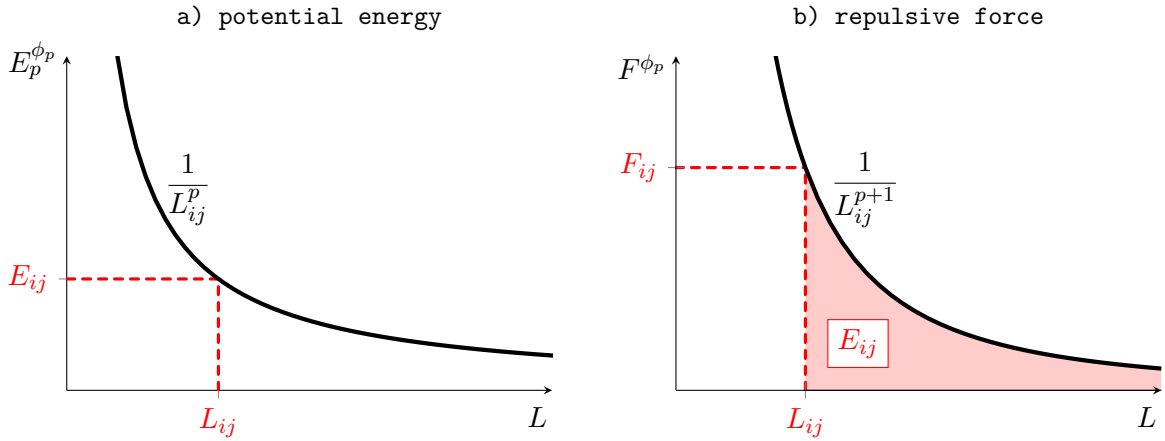


Figure 5.1: a) Potential energy of repulsive interaction between a pair of particles. b) Repulsive force acting on particles.

As the particle system contains N_{sim} interacting particles, the total potential energy of the system is a sum of contributions from all $\binom{N_{\text{sim}}}{2}$ individual pairs:

$$E^{AE} = \sum_{i=1}^{N_{\text{sim}}-1} \sum_{j=i+1}^{N_{\text{sim}}} E_{ij} = \sum_{i=1}^{N_{\text{sim}}-1} \sum_{j=i+1}^{N_{\text{sim}}} \frac{1}{L_{ij}^p}. \quad (5.5)$$

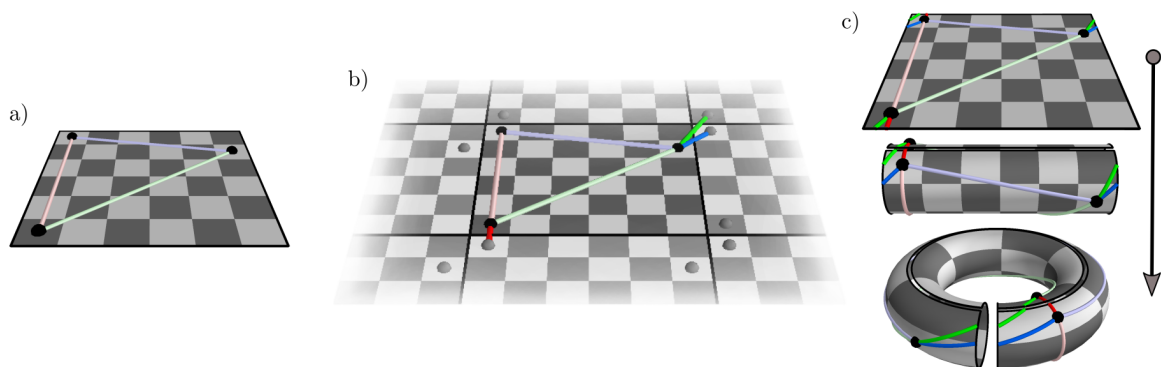


Figure 5.2: Illustration of periodically repeated planar domain. a) the original two-dimensional design domain with pale colored distances L_{ij} (Equation (5.2)). b) periodically repeated design domain with eight additional images of each particle. Periodic distances \bar{L}_{ij} (Equation (5.7)) are rich colored. c) folding the design domain into a torus is another possible illustration of a periodical domain. Note that the computed distances are not defined on the toroidal surface.

The total potential energy in Equation (5.5) represents the value of the Audze-Eglājs criterion to be minimized.

A simple and efficient improvement that considers a periodic extension of the design space has been proposed in [41]. After some simplification, one can derive equations for *periodic* Audze-Eglājs criterion (PAE) by replacing $\Delta_{ij,v}$ in Equation (5.2) with its periodic variant:

$$\bar{\Delta}_{ij,v} = \min(\Delta_{ij,v}, 1 - \Delta_{ij,v}). \quad (5.6)$$

With such a redefined projection, a new metric is obtained and the distance between points i and j , called the periodic length \bar{L}_{ij} , becomes the actual shortest linear path between point i and the nearest image of point j [41], also see Figure 5.2:

$$\bar{L}_{ij} = \sqrt{\sum_{v=1}^{N_{\text{var}}} (\bar{\Delta}_{ij,v})^2}. \quad (5.7)$$

We note that using the nearest image of point j with respect to point i does not cover a true periodic repetition of the design domain. It only satisfies the *minimum image convention*. In a complete periodic repetition, an infinite number of images of point j would interact with point i . The presented approach is a simplification that has been shown in [41] to yield identical results to the fully repeated system in case of sufficient point count, N_{sim} .

If the number of points in the original domain is too low for assembly of the desired self-similar pattern¹, considering additional periodical images of particles is advised. As argued in the following section, this is due to an insufficient resolution between short and long-range forces in the system. Another remedy is also to use stronger differentiation between short and long-range forces, that is to rise the exponent upon the mutual distance L_{ij} in the energy potential.

¹Simplest self-similar space-filling patterns can be assembled from simplest objects which contain volume in the particular dimension N_{var} : line in 1D (2 points), triangle in 2D (3 points), tetrahedron in 3D (4 points), etc.

In this section we consider a generalized model in which a certain number of periodic repetitions of the original design domain is assembled. If the number of points in the original domain is too small to carry enough information about the pattern of a periodically repeated system, making a periodic extension to a sufficient level is desirable. In a true periodic domain, an infinite number of images of point j would interact with point i . When a finite number of copies of the design domain is considered, not only the real particle j , but also all periodically repeated images of the particle j will contribute to the potential:

$$\phi_{(\bar{L}, p, k_{\max})} = \sum_{i=1}^{N_{\text{sim}}-1} \sum_{j=i+1}^{N_{\text{sim}}} \left(\frac{1}{\bar{L}_{ij}^p(\mathbf{x}_i, \mathbf{x}_j)} + \sum_{k=1}^{k_{\max}} \sum_{c=1}^{c_{\max}} \frac{1}{L_{ij}^p(\mathbf{x}_i, \mathbf{x}_j + \mathbf{s}_c)} \right), \quad (5.8)$$

where k_{\max} , introduced as an additional parameter, is the number of added periodical extensions (envelopes) of the design space. In the fully repeated system $k_{\max} = \infty$ and analogically, for a non-extended system $k_{\max} = 0$. Therefore $\phi_{(\bar{L}, p)} = \phi_{(\bar{L}, p, 0)}$.

When a certain number of envelopes k_{\max} is considered, the number of copies of the design domain is denoted as $c_{\max} = 0$. The vector \mathbf{s}_c is the vector needed for shifting the original point j to the particular periodically repeated version indexed by c . Let us denote that the distances to the periodically repeated images of the point j must be measured as the standard intersite Euclidean distances.

A single ‘‘level’’ of periodic extension adds another envelope of periodically repeated images of all other particles around each point, see Figure 5.3 and through Figure 5.6. Such an extension does provide additional information about the point layout within the domain.

The level of the periodic extension is quantified by a positive integer k_{\max} . Within an extended periodic domain of finite value of k_{\max} , the particle i interacts not only with the actual particle j , but with all of $c_{\max} = \left[(2k_{\max} + 1)^{N_{\text{var}}} - 1 \right]$ images of the particle j as well, see Figure 5.3 for $N_{\text{var}} = 1$. The envelopes are considered to be centered around the shortest distance with \bar{L}_{ij} .

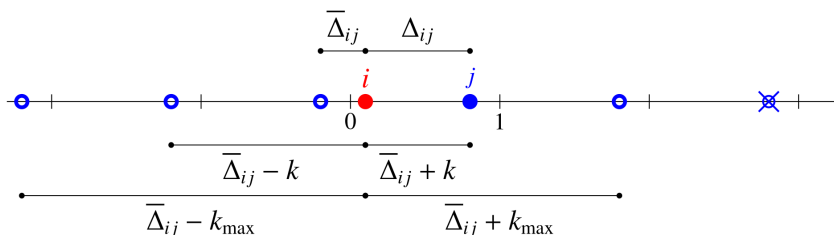


Figure 5.3: 1D example of the periodic extension of level $k_{\max} = 2$.

5.2 Selection of value of the distance exponent p

The authors of [41] argue that already for exponent $p = 2$, the consideration of the shortest distance \bar{L}_{ij} suffices to deliver invariance with respect to random shifts along individual dimension and thus prioritizes designs leading to statistical uniformity of coverage. Moreover, it is argued that the shortest distance is the one associated with the highest contribution

to the criterion and therefore the PAE criterion captures the important features of the full periodic repetition of the design space.

This section focuses on the exponent p in the periodic ϕ criterion. In the original AE criterion and also in the periodic version (PAE), the potential energy between each pair of points is not dependent on the dimension, N_{var} . It has been found [44] that the character of the criterion is different for various N_{var} and also for various numbers of points, N_{sim} . In a 1D situation, the energy tends to infinity linearly with increasing N_{sim} . In 2D, the energy tends to infinity as $\ln(N_{\text{sim}})$. For dimensions $N_{\text{var}} \geq 3$, the energy tends to a constant for increasing N_{sim} , as shown in [45].

This behavior can be explained by the fact that for a given $N_{\text{var}} > 1$, various numbers of points yield different proportions between energy due to the long-range and short-range interactions. The higher is the number of points, the higher the proportion of energy stored in long interactions is. This may not be desirable behavior as the criterion in high dimensions and also for high number of points becomes insensitive to local clusters of points: it becomes dominated by long-range interactions.

The power is suggested to be at least $p \geq N_{\text{var}} + 1$. With this power, the interaction is *dominated by short-range interactions*. With such a sufficient exponent p , the convergence of the potential energy $\phi_{(L,p)}$ or better $\phi_{(\bar{L},p)}$ towards infinity for a uniform distribution of points is a *power law*. Such a convergence signalizes self-similarity of the problem or absence of a length scale. In other words, a zoom into sufficiently dense uniform design with a window greater than a certain size (see below) carries all features of the full design and the energy value can be easily scaled from the value corresponding to the smaller zoom.

This can be shown by studying the behavior of the radial part of the integral of the potential over the volume V of N_{var} -dimensional domain. The potential energy for a uniform design reads:

$$I = \int_{N_{\text{var}}} \frac{1}{L^p} d^{N_{\text{var}}} V, \quad (5.9)$$

where L is used to denote one-dimensional distance between points (the symbol d is not used to avoid confusion with the symbol d for the differential). Transforming this into polar coordinated gives:

$$I = \int_{N_{\text{var}}} \varphi d^{N_{\text{var}}} V |J| \frac{1}{L^p} dL, \quad (5.10)$$

where $|J|$ is the Jacobian. The volume element is thereby given as:

$$d^{N_{\text{var}}} V = L^{N_{\text{var}}-1} dL \cdot d\varphi \prod_{i=1}^{N_{\text{var}}-2} \sin^{N_{\text{var}}-1-i}(\varphi_i). \quad (5.11)$$

Therefore, the integral is performed over the product $L^{N_{\text{var}}-1-i}$. Performing just the radial integration leads to:

$$I_r = \int \frac{L^{N_{\text{var}}-1}}{L^p} dL = \int L^{N_{\text{var}}-1-p} dL. \quad (5.12)$$

For $p = 2$ as used in the AE criterion, we get the behavior described above (i.e. $I \propto 1/L^2$). Using $p = N_{\text{var}}$ leads to:

$$I_r = \int L^{-1} dL = \ln(L), \quad (5.13)$$

which diverges logarithmically and the interaction is still long-ranged. Using $p = N_{\text{var}} + 1$ yields

$$I_r = \int L^{-2} dL = \frac{1}{L}, \quad (5.14)$$

which has the desired asymptotic behavior dominated by short-ranged interactions. Using powers $p > N_{\text{var}} + 1$ only increases the (asymptotically constant) ratio between short-range and long-range interactions.

Figure 5.5 shows the convergence of the normalized potential energy $\phi(\bar{L}, p)$ with rise of the number of particles, N_{sim} . Instead of presenting the results for the point count, N_{sim} , we introduce a variable ℓ_{char} , the *characteristic length* that involves also the dimension of the space and therefore expresses the *saturation of the design domain* with integration points (particles). The characteristic length is defined as:

$$\ell_{\text{char}} = \frac{1}{\sqrt{N_{\text{var}} N_{\text{sim}}}}. \quad (5.15)$$

The geometrical meaning of the characteristic length can be illustrated using a regular orthogonal grid of points in N_{var} -dimensional hypercube $[0, 1]^{N_{\text{var}}}$. The characteristic length represents the smallest distance between points within the grid. If N is the number of points over each dimension, the characteristic length can be derived as follows:

$$\ell_{\text{char}} = \frac{1}{N} = \left| N_{\text{sim}} = N^{N_{\text{var}}} \right| = \frac{1}{\sqrt{N_{\text{var}} N_{\text{sim}}}}. \quad (5.16)$$

At the same time, ℓ_{char} is the characteristic side length of the hypercubical volume belonging to each point in the space of sampling probabilities ($\ell_{\text{char}}^{N_{\text{var}}} = dU = 1/N_{\text{sim}}$), see Figure 5.4.

It can be seen that with the original exponent value $p = 2$ in the dimension $N_{\text{var}} = 2$ ($p = N_{\text{var}}$), the potential energy of the system does not converge to a power law but diverges logarithmically, roughly:

$$\phi(\bar{L}, p) \approx \pi \ln(N_{\text{sim}}) + \underbrace{\frac{1}{\sqrt{N_{\text{sim}}}} - \frac{1}{N_{\text{sim}}}}_{\rightarrow 0}. \quad (5.17)$$

In higher dimensions $N_{\text{var}} \geq 3$, the exponent $p = 2$ further leads to convergence of the potential energy to a constant, see [45].

Using the above proposed exponent $p = N_{\text{var}} + 1$, the potential energy value tends to a power law as $N_{\text{sim}} \rightarrow \infty$. Note the universality here: the quality of the criterion does not depend on sample size, N_{sim} , nor on the dimension, N_{var} :

$$\phi(\bar{L}, N_{\text{var}}+1) \propto \frac{1}{\ell_{\text{char}}}. \quad (5.18)$$

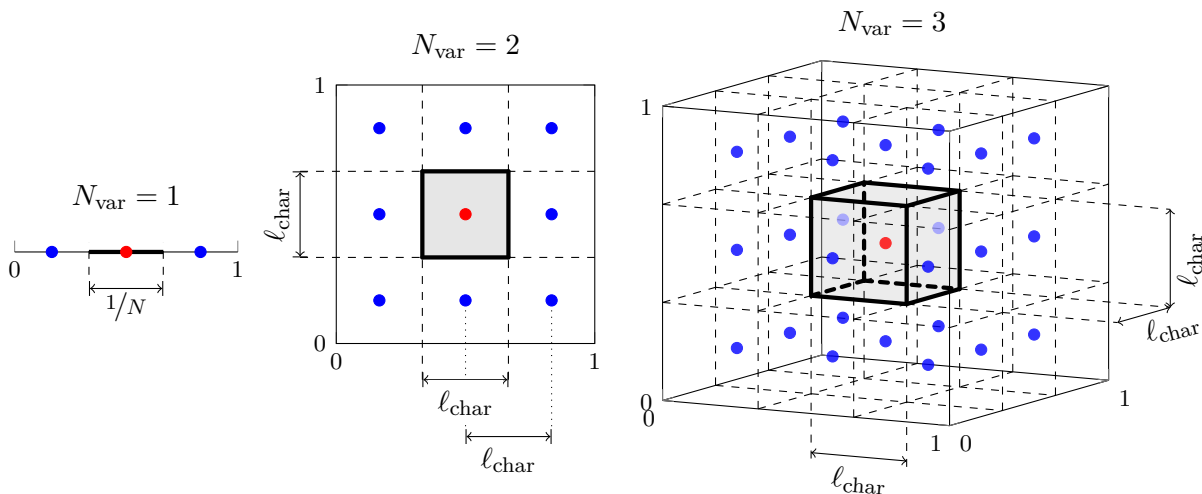


Figure 5.4: Geometrical illustration of the characteristic length, l_{char} .

Such a behavior is desired as the designs for a given dimension, N_{var} , tend to have a universal self-similar pattern and the dependence on sample size disappears (no length scale is present). Thus the character of the criterion is kept independent of N_{sim} and the proportion between short-range interactions and long-range interactions is constant. This stabilization is obtained for a sufficient number of points within the design domain (a kind of N_{var} -dimensional tile). The self-similarity manifested is by the power law dependence (a straight line in Figure 5.5). When the exponent is taken even higher ($p > N_{\text{var}} + 1$), the self-similar regime is achieved for even smaller number of points (greater l_{char}).

Graphs in Figure 5.5 suggest that there must be link between (a) the exponent (responsible for the proportion between long- and short-range interactions) and, (b) the number of “dummy” copies of the design domain that also modify the proportions. This aspect is discussed further.

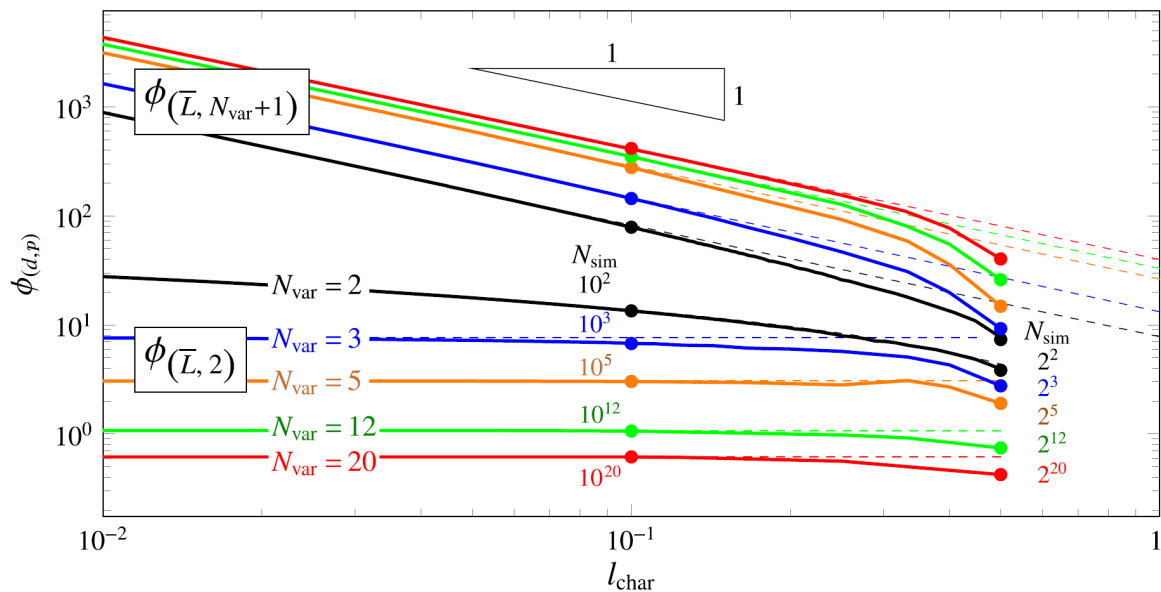


Figure 5.5: Convergence of the normalized potential energy $\phi_{(\bar{L}, p)}$ depending on the exponent p . Coloring that designate N_{var} is identical for the two bundles of curves. Solid circles are accompanied by the sample count corresponding to the ℓ_{char} and N_{var} .

5.3 On the similar effect of rising p and increasing number of periodic envelopes

Let us consider a few-body ($N_{\text{sim}} = 3$) particle system in design space of $N_{\text{var}} = 2$ while using a potential with the exponent $p = 2$. When considering the point layout via the $\phi_{(\bar{L}, p)}$ interaction, for each point, there exist only two mutual distances to other points, i.e. two forces acting upon each particle.

With an exponent of such insufficient magnitude, these forces do not differ significantly enough to represent correctly which particle shall be considered to be close (short-range interaction) and which to be far (long-range interaction). A rise of the exponent above a certain threshold (discussed above) does lead to the necessary qualitative change of the ratio between the acting forces: the closer particles start to act as short-range and the farther particles as long-range. In fact, the higher the exponent, the larger portion of potential energy will be stored in the short-range interactions.

Note that the increase in exponent upon the distance metric is needed for a sufficient differentiation between various pairwise distances in high-dimensional hypercubes. There, the coefficient of variation of measured distances tends to zero, as shown in [45]:

$$\lim_{N_{\text{var}} \rightarrow \infty} (\text{CoV}(L)) \propto \frac{1}{\sqrt{N_{\text{var}}}}. \quad (5.19)$$

The analogy between the effect of increasing the number of envelopes in and rising the exponent is evident. While using the original (low) value of the exponent p , majority of the potential energy is stored in the long-range interactions. However, there is not a sufficient

number of particles for the criterion to distinguish between short and long range. All particles seem to be at similar distance as the design is not filled enough.

The $\phi_{(\bar{L},p,k_{\max})}$ interaction adds one or multiple additional envelopes of neighboring images of actual particles. These images, naturally, will act as long-range. Even longer-range than the real particles previously acting as long-range. This leads to a qualitatively more accurate distribution of forces acting upon the real particles. Hence the identical behavior of the $\phi_{(\bar{L},p,k_{\max})}$ and the $\phi_{(\bar{L},p)}$ interaction with a correct exponent p :

- the $\phi_{(\bar{L},p,k_{\max})}$ interaction does add long-range points for the actual particles to seem closer,
- the $\phi_{(\bar{L},p)}$ interaction with corrected exponent changes the ratio between forces for the close particles to seem closer and the distant particles to seem farther.

It can be therefore shown that while simulating a few-body problem with the $\phi_{(\bar{L},p)}$ interaction, it is advised to rise the exponent p even above the lower bound of $N_{\text{var}} + 1$ to force the desired self-similarity for various N_{var} , see especially in Figures 5.6d and 5.6e. Similar situation is also in higher dimensions, see the same effect in Figure 5.7a and 5.7b.

When using the $\phi_{(\bar{L},p,k_{\max})}$ interaction, especially for few-body problems, greater context of the pattern is carried within the interaction for there is considered $3^{N_{\text{var}}}$ images of each particle. Effectively, a system mimicking $N_{\text{sim}} \cdot 3^{N_{\text{var}}}$ particles is being simulated and the identical pattern should be obtained, see Figures 5.6b and 5.6f.

On a side note, the simulation of a greater (extended) system might be the slower option in comparison with the correction of the exponent in the energy potential. This is due to a steep growth of the number of additional points within the added envelopes.

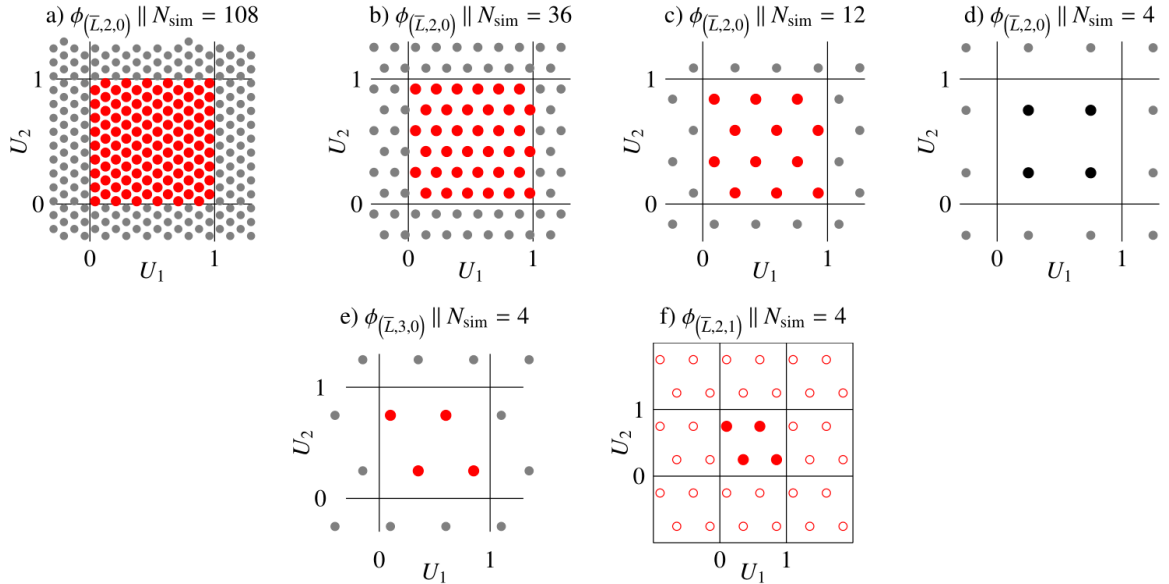


Figure 5.6: Disappearing of the quality of a self-similar design and the effect of remedies proposed.

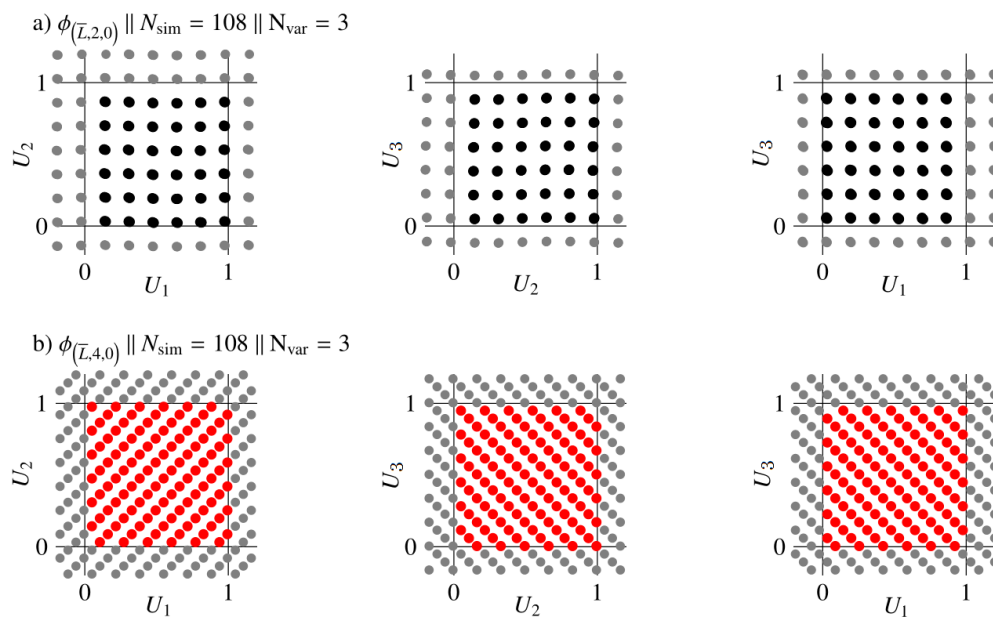


Figure 5.7: 3D designs: a) the original PAE ($p = 2$), b) the corrected potential exponent ($p = N_{\text{var}} + 1$).

Chapter 6

Physical analogy between the ϕ_p criterion and a dynamical N-body system

The Audze-Eglājs criterion, and its generalization, the ϕ_p criterion, propose to understand the layout of design points as a system of interacting (mutually repelling) particles and evaluates the amount of potential energy stored within all interactions, see Figure 5.2 for illustration.

Instead of utilizing the AE/PAE criterion as a norm minimized using combinatorial or heuristic optimization for a fixed set of LHS coordinates [39], this work proposes to solve the physically analogical problem by simulating a discrete dynamical system of mutually repelling particles, recall Figure 4.1. The coordinates of particles of the dynamical system, after reaching the static equilibrium (after a minimization of potential and kinetic energy), may be directly understood as coordinates of design points within the unit hypercube, see Figure 6.1.

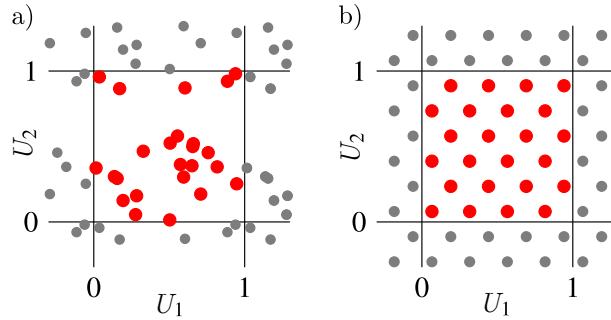


Figure 6.1: Illustration of the optimization process of $N_{\text{sim}}=24$, $N_{\text{var}}=2$. a) initial randomized sample, b) optimized sample.

As argued in Chapter 5, a direct usage of the (P)AE energy potential, see Equation (5.5), is not advised for it has been shown that the exponent upon the mutual distances shall depend on the dimension of the design domain as well as, partially, on the number of particles. The formulation of the potential energy of a system of charged particles will therefore be generalized and from now on, the exponent p will be utilized for derivation of equations

of motion of the dynamical particle system. Through the following content, mind also Figure 6.2 that illustrates the interaction of two particles i and j .

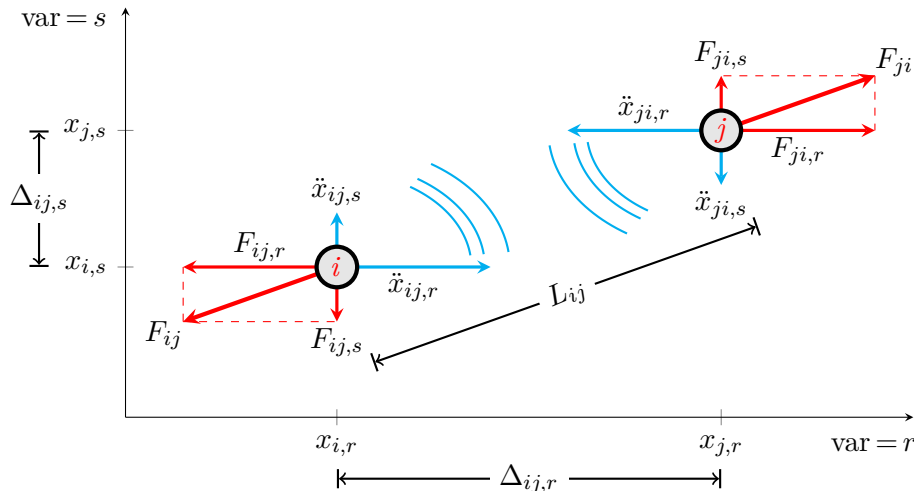


Figure 6.2: Interaction of two charged particles.

The derivation itself may be conducted by using various approaches, all of which lead, of course, to identical results. Essential remarks about derivation using Lagrangian mechanics are provided in what follows. To begin, let us state that the dynamical behavior of a mechanical system with a finite number of degrees of freedom can be described by the Lagrange's function, or shortly Lagrangian, \mathcal{L} . Sometimes also called a *kinetic potential*, the Lagrangian \mathcal{L} is a functional; a sum of formulations of kinetic and potential energy. In case of the ϕ_p -conditioned dynamical system, the Lagrangian can be described as follows:

$$\mathcal{L} = E_k + E_p, \quad (6.1)$$

with the kinetic energy of the particle system E_k being a simple sum of kinetic energies of all particles of equal mass m :

$$E_k = \frac{1}{2} m \sum_{i=1}^{N_{\text{sim}}} \sum_{v=1}^{N_{\text{var}}} \dot{x}_{i,v}^2, \quad (6.2)$$

where $\dot{x}_{i,v} = \frac{d}{dt} x_{i,v}$ is the velocity of i th particle in dimension v .

We now consider a generalized formulation of the potential energy, E_p , employing an arbitrary value of the exponent, p , upon the mutual distances, \bar{L}_{ij} . The potential energy of the model can be written as a sum of energies stored within all mutual inter-particle interactions:

$$E_p = \sum_{i=1}^{N_{\text{sim}}-1} \sum_{j=i+1}^{N_{\text{sim}}} \frac{1}{\bar{L}_{ij}^p}, \quad (6.3)$$

where the exponent is now considered as a general integer, p , (similarly to the ϕ -criterion [21]) and the metric considered is the periodic length, \bar{L}_{ij} , see Equation (5.7).

Further, it is needed to calculate the derivatives of Lagrangian \mathcal{L} with respect to all state variables. In the case at hand, the state variables are the coordinates $x_{i,v}$ and velocities $\dot{x}_{i,v}$ of all particles in each dimension. Obeying the Lagrange's equations of the second kind:

$$\frac{d}{dt} \left(\frac{\partial \mathcal{L}}{\partial \dot{x}_{i,v}} \right) = \frac{\partial \mathcal{L}}{\partial x_{i,v}}, \quad (6.4)$$

one can start off with the assumption that, apart from the derivatives with respect to time, t , the kinetic energy E_k is further differentiable only with respect to velocities $\dot{x}_{i,v}$ and the potential energy E_p (6.3) is differentiable only with respect to coordinates $x_{i,v}$. Therefore, the left-hand side of Equation (6.4) is rather easily obtainable:

$$\frac{d}{dt} \left(\frac{\partial \mathcal{L}}{\partial \dot{x}_{i,v}} \right) = \frac{d}{dt} \left(\frac{\partial E_k}{\partial \dot{x}_{i,v}} \right) = m \ddot{x}_{i,v}, \quad (6.5)$$

with $\ddot{x}_{i,v} = \frac{d}{dt} \dot{x}_{i,v}$ being the acceleration of the i th particle in the dimension v .

The right-hand side of Equation (6.4) becomes:

$$\frac{\partial \mathcal{L}}{\partial x_{i,v}} = \frac{\partial E_p}{\partial x_{i,v}} = \sum_{\substack{j=1 \\ j \neq i}}^{N_{\text{sim}}} \left(\frac{1}{\bar{L}_{ij}^{p+1}} \frac{\bar{\Delta}_{ij,v}}{\bar{L}_{ij}} \right). \quad (6.6)$$

The resulting equation of motion of i th particle in v th dimension as assembled from Equations (6.4), (6.5) and (6.6) finally reads:

$$\ddot{x}_{i,v} = \frac{1}{m} \sum_{\substack{j=1 \\ j \neq i}}^{N_{\text{sim}}} \frac{\bar{\Delta}_{ij,v}}{\bar{L}_{ij}^{p+2}}. \quad (6.7)$$

Note that these are equations of motion of a conservative dynamical system as defined by the energy potential (6.1) which does not cover any form of energy dissipation.

The motion of the dynamical system is therefore described by a system of independent equations. This awareness is of high importance while considering the possibilities for solution method and its computer implementation. It means that accelerations $\ddot{x}_{i,v}$ of each particle $i = 1, \dots, N_{\text{sim}}$, in each dimension $v = 1, \dots, N_{\text{var}}$, can be solved separately, without solving a system of equations.

The damping of motion of particles also depends solely on the velocity of each particle. Furthermore, each distance projection $\bar{\Delta}_{ij,v}$ as well as each absolute distance \bar{L}_{ij} can be computed independently. The above-mentioned properties lead to the possibility of utilizing a *parallel* implementation.

As soon as the new accelerations of each particle in each dimension are obtained, the equations of motion are numerically integrated using the semi-implicit Euler method and the new velocities $\dot{x}_{i,v}$ and coordinates $x_{i,v}$ of each particle in each dimension at the new time $(t + \Delta t)$ are computed:

$$\begin{aligned} \dot{x}_{i,v}(t + \Delta t) &= \dot{x}_{i,v}(t) + \Delta t \cdot \ddot{x}_{i,v}(t), \\ x_{i,v}(t + \Delta t) &= x_{i,v}(t) + \Delta t \cdot \dot{x}_{i,v}(t + \Delta t). \end{aligned} \quad (6.8)$$

For reaching the static equilibrium of the dynamical system, implementing of energy dissipation is desirable. Various types of damping are typically combined. For solving the problem at hand, we add the sum of velocity-dependent damping members into second of Equations (6.8): $\sum_p c_p \dot{x}_{i,v}^p(t)$, where c_p are damping coefficients and p are various powers of the velocity, $\dot{x}_{i,v}$, of i th particle in v th dimension. Note, that the damping part is not derived from the energy potential of the particle system.

6.1 Enhancing numerical stability of solution

As a closing remark on the topic of the dynamical system of mutually repelling particles, notes on the improvements done in order to enhance the numerical stability of solution will be briefly mentioned.

It has been stressed out multiple times that the refinement of the ϕ_p optimization criterion aimed at obtaining a scale-independent power law, recall Figure 5.1. This requirement is argued as desirable for the process of sample optimization. However, if used as a formulation of potential energy of the system of charged particles, the definition of repulsive forces:

$$F_{ij} = \frac{1}{L_{ij}^{p+1}}, \quad (6.9)$$

results in an ill-posed problem in the sense that a small difference in the mutual distance, \bar{L}_{ij} , between a pair of particles might lead to large differences in the resulting repulsive force, F_{ij} . Such a resolution of mutual forces is desirable, however numerically, during the dynamical simulation, there are possible undesired situations:

- since the initial coordinates of particles at the beginning of each simulation are set as random, a value of repulsive force beyond the range of the used 32-bit precision might be obtained if two particles are generated very close to each other¹,
- note the dynamical system considers the particles to be *collisionless bodies*. Therefore, during the simulation using discrete time steps, there exist a possibility of two particles appearing too close to each other, inducing a close-to-infinity repulsive force.

Remedies of such unwanted behavior of a numerical solution of an N-body system can be tracked in the field of astrophysics simulations, see e.g. [46]. The implemented approaches to *regularization* of the energy potential and the repulsive forces, respectively, are introduced in this section. Note that the following modifications of energy potential are done only for the purposes of the dynamical simulation. The ϕ_p optimization criterion, that serves for evaluation of uniformity of statistical samples, remains untouched.

The ill-posed formulation of potential energy is in fact not of any interest during the actual numerical solution of the particle system. Only the formulation of repulsive forces needs to be regularized as it is used for computation of accelerations of particles. The first of

¹Setting the initial coordinates of particles as random is otherwise very beneficial. Random initial coordinates, and resulting high initial repulsive forces, supply the system with enough kinetic energy to overcome most of possible locking in local minima of the potential energy.

explored approaches to regularization of repulsive forces is based on addition of the cross-over distance λ into Equation 6.9 as follows:

$$F_{ij}^\lambda = \frac{1}{\left(L_{ij}^2 + \lambda^2\right)^{\frac{(p+1)}{2}}}. \quad (6.10)$$

That way, the new obtained interaction law is bounded from above by a horizontal asymptote at $L_{ij} = 0$. The force can never exceed:

$$F_{\max}^\lambda = \frac{1}{\lambda^{p+1}}, \quad (6.11)$$

and converges to the second asymptote that is the original power law for $L_{ij} = \infty$, identical to Equation (6.9), see Figure 6.3.

In Equation (6.10), the parameter λ plays a role of cross-over distance, at which the two asymptotes intersect. By setting low enough λ , the issue of the system being governed by an ill-posed, inverse law is remedied at a “small cost”. This approach is commonly used in astrophysics simulations where there is no interest in attaining a static equilibrium of the system. For the N-body system in this work, enforcing a function other than the derived power law is undesired for it distorts the ratio between repulsive forces in the resulting steady-state. Therefore the resulting optimized point patterns do not entirely obey the ϕ_p criterion.

Unlike the astrophysics particle systems, the purpose of the N-body system in this work is to find a configuration that minimizes potential energy of the system. The trajectory that leads to such a steady-state solution is considered to be not of interest. For these static-equilibrium states (point patterns), one can assume that the minimum shortest pairwise distances within such (semi)optimal patterns are about the length of ℓ_{char} . It is therefore crucial to preserve the power law interaction at least to this threshold. Let us require the power law to continue from $L_{ij} = \infty$ to certain distance of $L_{ij} = \lambda$. On a safe side, consider $\lambda = \ell_{\text{char}}/10$. Any changes in particle interaction below this limit will not spoil the desired point patterns, but will prevent the accelerations from approaching infinity.

A convenient modification then is to graft a linear function to the interaction law for distances below λ^{lin} , see Figure 6.4. That way, if the pairwise distance is larger than λ^{lin} , the proposed power law interaction (6.9) is used as is. Once the mutual distance between particles becomes lower than λ^{lin} , the repulsive force is computed as follows:

$$F_{ij}^{\text{lin}} = F_{\lambda^{\text{lin}}} + (\lambda^{\text{lin}} - L_{ij}) \frac{d}{dL} \left(F^{\phi_p}(\lambda^{\text{lin}}) \right). \quad (6.12)$$

The upper bound on the linearized repulsive force F_{\max}^{lin} then reads:

$$F_{\max}^{\text{lin}} = F_{\lambda^{\text{lin}}} + \lambda^{\text{lin}} \frac{d}{dL} \left(F^{\phi_p}(\lambda^{\text{lin}}) \right). \quad (6.13)$$

Using a practical perspective, even smaller value of λ (i.e. even wider range of unharmed original power law), that can produce the same value of F_{\max} , can be obtained by the simplest solution possible. That is by grafting a complete “plasticity” to the interaction law below λ^{cst} , also see in Figure 6.4. That way, if an interparticle distance gets any lower than λ^{cst} , the repulsive force remains constant:

$$F_{ij}^{\text{cst}} = F_{\max}^{\text{cst}} = F^{\phi_p}(\lambda^{\text{cst}}). \quad (6.14)$$

This simple and pragmatic approach eventually remained the used one in the developed dynamical simulation. Empirical observations show that the value of $\lambda^{\text{cst}} = \ell_{\text{char}}/20$ serves the purpose sufficiently.

For stabilizing the dynamical system at yet another stage of simulation, a limit of a maximum particle velocity in each dimension is set. The purpose is mainly to prevent particles from moving too fast to protrude through another particle or a cluster of particles. Naturally, the ultimate solution is to lower the time step, Δt , to an extremely small value. In the case of the inherently chaotic system at hand, such an overly safe time step is not necessary as trajectories of particles are not of interest². To ensure a “smooth” course of simulation, during numerical integration of equations of motion, it is controlled that each particle does not travel farther than $\ell_{\text{char}}/10$ along each dimension during a single integration step:

$$\dot{x}_{ij,v} < \frac{\ell_{\text{char}}/10}{\Delta t}. \quad (6.15)$$

Lastly, to ensure a correct course of simulation in the early stage with the highest kinetic energy of particles, the size of time step, Δt , is adjusted. To capture the relatively fast motion of particles, a rather small value (currently 10^{-8}) is set at the beginning. As the simulation proceeds, the motion of particles becomes less chaotic and increasingly higher value of Δt is used.

As such a *soft-start* of the system, a convenient step function that follows a linear trend limited by a constant is used in this work, see Figure 6.5. The length of steps of constant Δt as well as the increment of Δt can be either user-defined or can depend statistically on the value of total kinetic energy of the system. The combination of the presented methods enhancing the solution stability has brought the desired effect during the conducted simulations.

²If the kinetic energy of the system is high enough at the beginning of simulation, the system can find the desired minimum of potential energy even if the theoretical trajectories of particles are slightly corrupted.

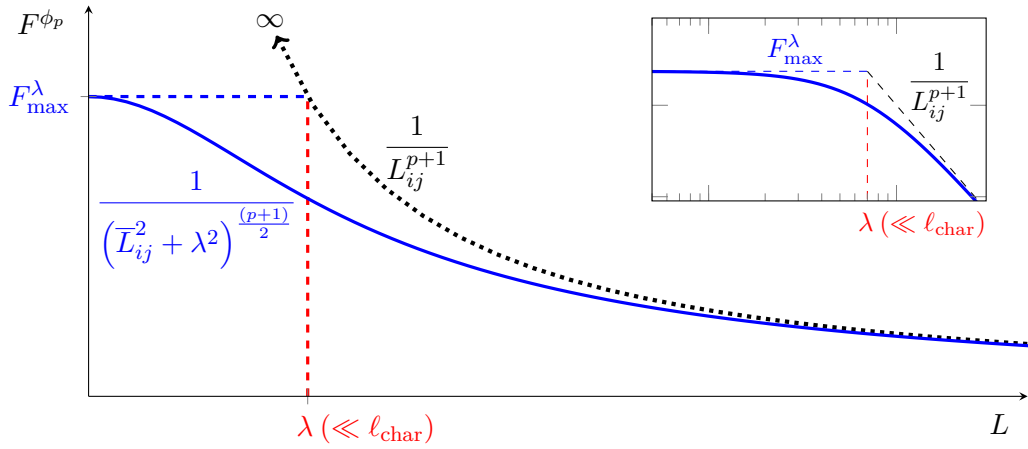


Figure 6.3: A possibility of regularization of repulsive forces by an asymptotical softening of the interaction law.

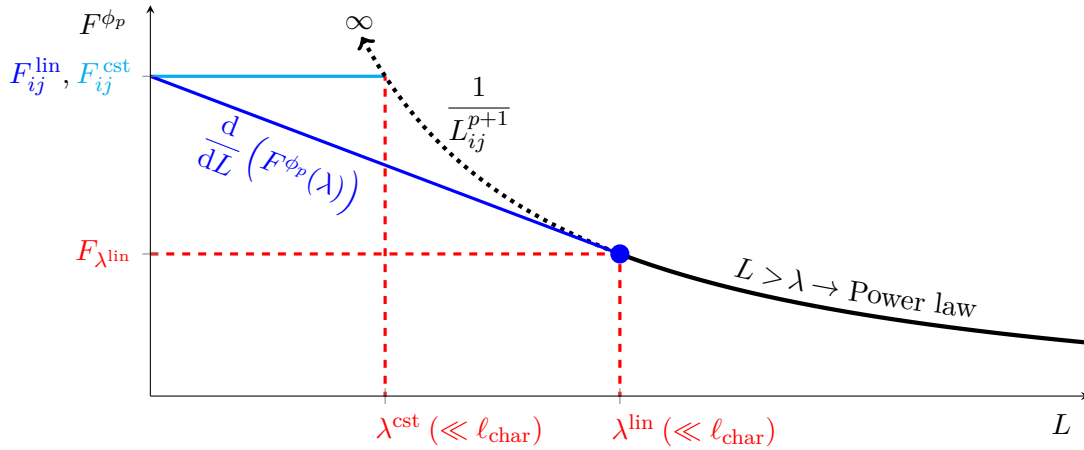


Figure 6.4: A possibility of regularization of repulsive forces by setting a constant bound (red) or by linearization (blue).

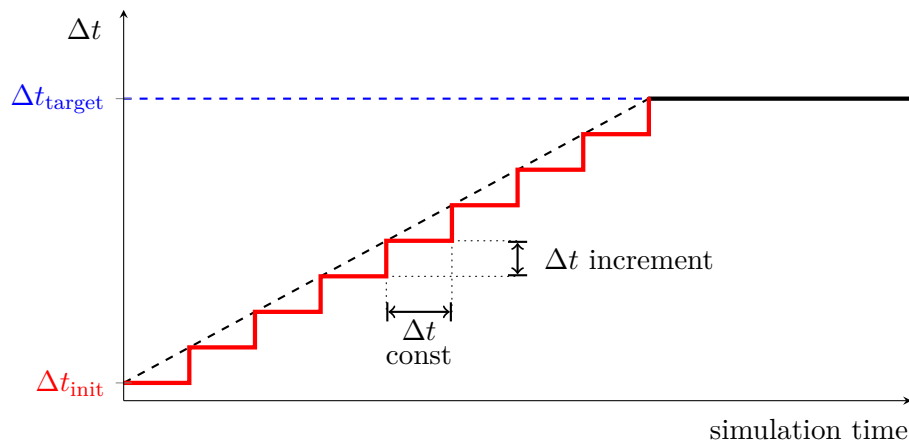


Figure 6.5: Soft start of the numerical simulation.

Part II

NVIDIA COMPUTE UNIFIED DEVICE ARCHITECTURE

Chapter 7

General Purpose Computing on Graphics Processing Units

Since the introduction of the very first operating computing device driven by a germanium integrated circuit (IC) by Texas Instruments' Jack Kilby in 1958 [47] and its silicon-based variant by Fairchild Semiconductor's Robert Noyce in 1959 [48], the computing power of modern-era computing devices was a subject of a rapid growth sustainable for decades.

The strides in the manufacturing technology provided a constant growth of number of transistors in integrated circuits. Such a trend was observed and is well recognized as the Moore's Law [49], named after Gordon Moore, the co-founder of Fairchild Semiconductor and Intel companies. The Moore's law predicted that the number of transistors in integrated circuits will double approximately every two years. Indeed, for many years, the manufacturers were able to maintain this predicted trend and deliver microprocessors with increasingly higher number of transistors. The actual computational performance grew even faster, doubling approximately every 18 to 20 months due to the additional growth of clock rates.

Thanks to these continuous advances in microprocessor manufacturing technology, software developers were provided with a seemingly never-ending supply of additional horsepower for their applications. Moreover, they could take advantage of each faster generation of processors with very little additional adjustments in their current code or, in many cases, without any changes in the code at all.

The microprocessor architecture as used most commonly in CPUs (Central Processing Units) of computers aims at a fast execution of a single sequence of logical instructions upon a single data structure by a single Arithmetic Logic Unit (ALU) operated by a single control unit. According to *Flynn's Taxonomy* [50], such an instruction and data flow is known as Single Instruction Single Data (SISD). Excluding the part for the execution of logic instructions, the remaining majority of the die area of a CPU is typically dedicated to *cache*. The cache is a relatively limited on-chip memory that is used for a temporary storage of necessary data that is being operated. The consequence of using the cache for the required data is that the control unit has to issue significantly less read/write requests from the orders of magnitude slower operational memory. This way, the resulting *memory access latency* during the execution is considerably decreased.

In early 1990's, alongside the CPU architecture optimized for fast sequential arithmetics,

multiple manufacturers began the development of an auxiliary platform for acceleration of generation of raster images for displays, later widely recognized as Graphics Processing Unit (GPU)¹. The rapidly evolving graphics of computer and console games was the main driving force of the frenetic development done by the leading GPU manufacturers of that time: Matrox, S3 Graphics, 3dfx Interactive, ATI Technologies and Nvidia Corporation. The real-time 3D graphics became a standard for computer games of late 1990's and the public demand for more powerful graphic accelerators grew ever since.

The GPU architecture in its essence proposes an entirely opposite philosophy of operation than CPUs. The task of a GPU is to execute the same instruction upon each pixel of the assembled raster image. Therefore, efforts to conduct this kind of operation on sequential CPUs were rather quickly disqualified². The concept that proved itself as the most powerful is the *parallel computing architecture* consisting from multiple ALUs, all of which execute the same instruction on different independent data (a pixel at that time). According to the nomenclature of the Flynn's taxonomy, such an instruction and data flow is classified as Single Instruction Multiple Data (SIMD). Unlike CPUs, the GPU platform dedicates the majority of its die area to ALUs and only a small portion to cache. After all, the GPU's cache was originally meant as a buffer rather than for decreasing latency.

By the end of 1990's, both platforms still followed the prediction of the Moore's law. Each platform, however, benefited from the Moore's law differently. The CPU performance was improved by growing clock rates and using the additional transistors for new faster instruction sets while the GPU horsepower was driven by increasing the number of ALUs rather than their actual clocks and compute capabilities. Around 1998, a crossover of the number of transistors of both platforms was reached as both the Intel Pentium 2 CPU and the Nvidia RIVA TNT GPU provided transistor count around 8 million. Since then, the GPU architecture was always equipped with more transistors than CPUs of each comparable generation³.

At the beginning of the millennium, a notion appeared [1] that the CPUs will no longer be able to provide additional speedup by simply driving the clock rates. While the possibility of increasing the number of transistors did not yet reach the limits of the materials used, the growth of the clock rates as was customary for decades was predicted to soon reach the bearable thermal boundaries. It was universally acknowledged that the future speedup will have to come from *simultaneous execution* of multiple threads operated by multiple cores (ALUs). Both CPU and GPU manufacturers did continue to rise core clocks to some extent but it was clear that the only way how to sustain the growth of computational performance is by the *parallel execution*, effectively meaning by utilizing more ALUs. Indeed, the CPU manufacturers did embrace this doctrine of parallelism and do provide multi-core CPUs with increasingly higher number of physical and virtual cores. Nevertheless, the CPU architecture got into a clear disadvantage as the GPU platform was designed as parallel from scratch.

A temporary obstacle of GPUs was that the performance within reach was so far dedicated only for graphics routines (*shaders*). The purpose of the so-called *shaders* is to compute the level of brightness and color of a pixel due to e.g. lighting, shadows, translucency and the position of the observer of the rendered 3D scene.

Until 2007, any utilization of the GPU performance for computing other than graphics

¹The actual term "Graphics Processing Unit" was officially used years later in 1999 by Nvidia [51].

²An exemplary instance of such a fruitless effort was the Intel MMX (Matrix Math Extension) instruction set [52].

³The most powerful contemporary Nvidia GPU, the Tesla V100, is equipped with 21.1 billion transistors.

rendering was extremely rare and would have to be done using graphics APIs (Application Programming Interfaces) such as OpenGL [53] or Direct3D [54]. The breakthrough happened on June 23, 2007, when Nvidia released its set of development tools named Compute Unified Device Architecture (CUDA) [3]. CUDA allowed general-purpose applications to be executed by Nvidia GPUs starting with the contemporary Tesla architecture (GeForce 8800GTX and Tesla C870).

Currently, Nvidia's CUDA is the leading framework for General-purpose computing on graphics processing units (GPGPU) despite being limited only to Nvidia hardware. Alongside the Nvidia's proprietary CUDA, the standardized OpenCL (Open Computing Language) framework [2] is developed by the Khronos Group. The aim of OpenCL is to provide a standardized platform for parallel computing that is independent of the actual hardware. Effectively, an identical OpenCL code can be executed on desktop or even mobile multi-core CPUs (e.g. Intel, AMD, ARM, IBM, Samsung, Qualcomm etc.) as well as on GPUs (mainly Nvidia and ATI).

CUDA, however, still delivers a more user-friendly environment with its support of programming languages used in scientific computing such as C/C++ [55], Fortran [56], Python [57–59] and more. Moreover, Nvidia continuously maintains and updates a wide array of scientific computing libraries that are supplied with the CUDA Development Toolkit [60]:

- CUDA Math (high performance standard math routines),
- cuBLAS (library for standard basic linear algebra subroutines),
- cuSPARSE (basic linear algebra subroutines for sparse matrices),
- cuRAND (fast random number generation),
- cuSOLVER (Cholesky, LU, SVD and QR decomposition, eigenvalues),
- cuFFT (GPU accelerated Fast Fourier Transformation),
- cuDNN (standard routines for Deep Neural Networks),
- AmgX (nested solvers, smoothers, and preconditioners).

In practical situations, the usage of a GPGPU computing platform may provide a speedup that varies depending mainly on the nature of the problem solved (i.e. how *parallelizable* the solution is) and on the actual programming approach used (i.e. how *parallelized* the solution has become). A more in-depth description of the Nvidia CUDA architecture of the current Pascal generation follows. Its aim is to provide the reader with sufficiently detailed information about this rather complex topic and deliver pointers to further convenient literature sources.

The GPGPU architecture (and CUDA/OpenCL, respectively) still is a relatively new computing platform and its community of programmers and GPU vendors still continue to push the limits of code efficiency even for basic problems. The performance of a GPU solution is also highly dependent on capabilities of the actual used hardware, settings of block and grid sizes, shared memory use, register use and more. As will be explained in what follows, the actual performance of each programming approach better shall be judged by nothing else but the *wall clock time*.

7.1 Architecture of a CPU-GPU system

Before diving into the description of the current CUDA-capable hardware, the following section provides a perquisite information about basic configurations of a CPU-GPU system assembly. In CUDA terms, the part operated by the CPU is called *the host*. The attached GPU component is called *the device*. Every such a hardware configuration has to contain following essential components:

- a motherboard,
- a CPU chip,
- operational memory for CPU,
- a GPU (dedicated card or integrated into motherboard's chipset).

The motherboard serves to connect all the required hardware into a mutually communicating system. Up until ca 2008 (Intel's Core 2 and AMD's K8 CPU architectures), the connection between the CPU and its operational memory was maintained by the so-called *northbridge* chip, a part of motherboard's chipset designed for handling communication between CPU, operational memory and a hub for connection of a dedicated GPU card, see Figure 7.1. Among other, the northbridge was accompanied by the *southbridge* chip that handled the input/output (I/O) functions of the peripherals and other tasks (real-time clock, BIOS, etc.).

The connection of CPU to the northbridge and its memory controller, respectively, was conducted via the Front-Side bus (FSB) interface. For connection of the GPU to the system, the Accelerated Graphics Port (AGP) was used for years, later around 2005 replaced by the Peripheral Component Interconnect Express (PCI-e) bus.

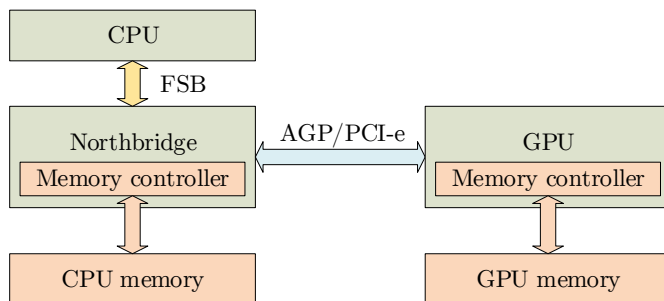


Figure 7.1: A diagram of a CPU-GPU system connected by northbridge.

The approach of using a kind of an intermediate connection layer between CPU and operational memory (here via FSB to northbridge and via the memory bus to operational memory) soon became a significant *performance bottleneck* of such systems. The reason is the following difficulty: both the CPU and northbridge need to run on a clock rate that is an exact multiple of the FSB clock rate. At the same time, the memory bus has to run on the very same frequency as the FSB. This means that the memory runs on an exact fraction of the CPU clock rate. For applications where there is a rather low amount of arithmetic CPU labor to be done upon a large amount of data, the CPU will spend a great portion of execution time idle and waiting for memory response. The clock of AGP/PCI-e buses did not depend on FSB frequency in such a way.

A remedy of this kind of performance bottleneck arrived in 2008 as the new generations of CPUs (Intel's Nehalem and AMD's K10) integrated the memory controller directly into the CPU chip, see throughout Figure 7.2. The CPU thus manages its own connection to operational memory. In case of a multi-CPU system, each CPU operates its respective memory controller and bandwidth.

The motherboard's chipset therefore ceased to contain northbridge and southbridge chips as the northbridge was acquired by the CPU and the southbridge (since then also operating PCI-e buses) is further developed by each CPU vendor as an I/O hub in their proprietary chipsets that is directly connected to the CPU. The Intel's Sandy Bridge CPU architecture proceeds even further as its CPU chip embraces the I/O hub as well.

Instead of FSB, current CPUs are connected to the I/O hub via a proprietary bus (Intel's QuickPath Interconnect (QPI) and AMD's HyperTransport (HT)). Both technologies also serve for mutual connection of CPUs in multi-CPU machines. Such a connection allows for sharing cached data between CPUs which may lead to significant savings of requests to operational memory.

Dedicated GPU cards are equipped with their own operational memory that resides on the actual GPU board. The GPU also operates its own memory controller that is integrated in the GPU chip. Such a basic contemporary assembly of a single CPU-single GPU system is illustrated in Figure 7.2.

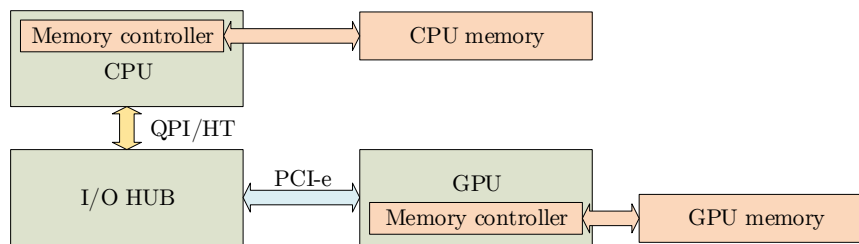


Figure 7.2: Connection of CPU and single GPU via motherboard chipset and PCI-express bus.

Assembly of multi-GPU systems is also made possible as current motherboards are commonly equipped with multiple PCI-e slots. Further, server solutions that may contain many GPUs are offered by GPU vendors. Utilization of the horsepower of such a multi-GPU assembly depends on the way the GPUs are connected to the system. First of the two connection possibilities is to connect GPUs into PCI-e buses as several stand-alone devices, see Figure 7.3. The other multi-GPU solution is to use a physical bridge to connect multiple GPUs. Such connected GPUs then behave as a single (almost) proportionally more capable GPU connected by a single PCI-e slot, see Figure 7.4. Both leading GPU vendors, ATI and Nvidia, offer their own multi-GPU solution of this kind. ATI's CrossFire as well as Nvidia's Scalable Link Interface (SLI) currently allow to connect up to 4 GPUs in the described way⁴.

In case of graphics applications, identical data and very similar instructions are broadcasted to all SLI-connected GPUs. Each GPU then executes its portion of rendering labor,

⁴It is worth noting that both ATI and Nvidia exploited the knowledge of 3dfx Interactive and its Scan-Line Interleave (also SLI) technology. 3dfx Interactive was acquired by Nvidia after initiating bankruptcy proceedings in late 2000. Several 3dfx employees were also dragged over to ATI and began the development of ATI's CrossFire technology.

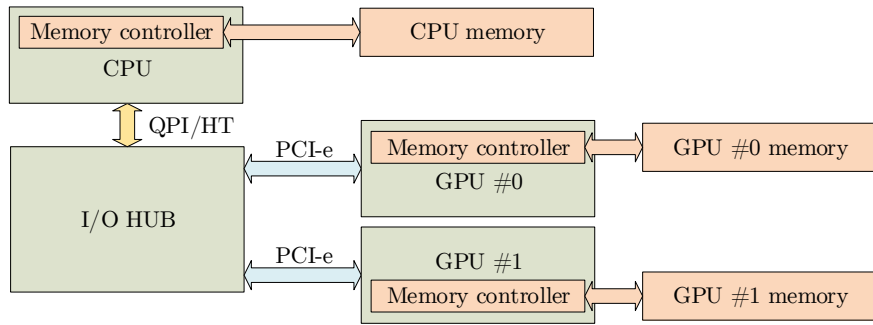


Figure 7.3: Connection of CPU with multiple GPUs via multiple PCI-express buses.

the overall result is assembled from contributions of all GPUs and then typically copied to the graphics buffer. The SLI/CrossFire technologies are meant for convenient distribution of rasterization workloads. For GPGPU computing, utilization of the SLI/CrossFire connection is questionable because of the very fact that all GPUs appear as one large GPU which is not technically true.

Two identical GPUs connected with SLI actually do not operate twice as much continuous resources of memory or fast on-chip cache. All shared data must be copied via the SLI bridge between the GPU cards. Any kind of such interaction between GPUs then relies on the SLI bridge bandwidth. This becomes a significant *memory bandwidth bottleneck* as the bandwidth of the currently most powerful SLI HB bridge is 2 GB per second. Comparison to memory bandwidth of current high-end gaming GPU GeForce GTX1080Ti that is 484GB per second⁵ renders the usage of SLI for CUDA ineligible.

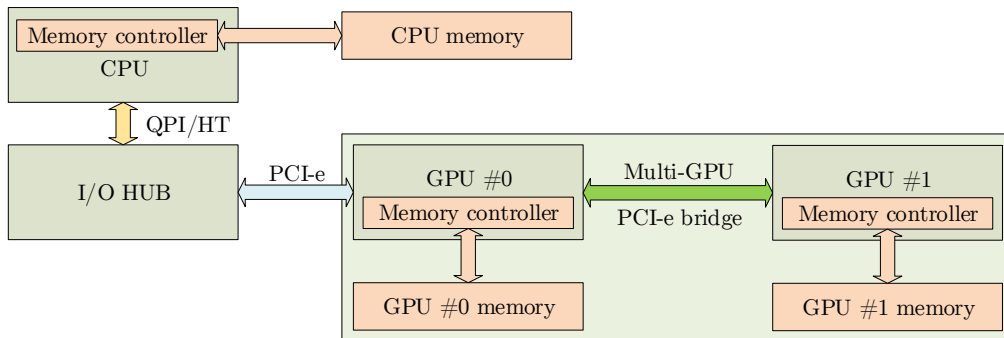


Figure 7.4: Connection of CPU with multiple GPUs mutually connected by a multi-GPU bridge.

Furthermore, such connected GPUs communicate with the CPU and its operational memory only via a single PCI-e slot of GPU#0, see Figure 7.4. The peak bandwidth of the current PCI-e 3.0 bus is 16 GB per second. This already makes it a weak spot as is. Therefore sacrificing the remaining PCI-e buses for SLI makes sense only for applications where there is not much of data transfer between GPUs and GPU memory and also minimal

⁵Not to mention the 720GB/s of memory bandwidth of the currently most powerful Tesla GP100 and 900GB/s expected from the upcoming Tesla V100.

interaction between GPUs. The primary purpose of the SLI interface is to synchronize framebuffers between GPUs. GPGPU usage is definitely not an application of such a kind and generally does not benefit from SLI. A hierarchical interface for GPU-GPU and CPU-GPU interconnection, however, is on sight, see [61].

Further in this work, a single CPU-single GPU system configuration is going to be considered. The GPUs used are the Kepler generation Nvidia Tesla K20c (GK110 core) and the Pascal architecture Nvidia GeForce GTX 1080Ti (GP102 core). The currently most powerful GPU, Tesla GP100 (GP100 core) is also going to be discussed as the state-of-the-art GPU.

Chapter 8

Nvidia GPU architecture

The following chapter aims to discuss the properties of Nvidia’s GPGPU hardware. Each generation of Nvidia devices that supports CUDA is given its unique version of *Compute Capability* (CC). The Compute Capability by two digits describes both physical and logical setup of any Nvidia GPU. The first digit describes the overarching GPU generation (capturing mainly instructional capabilities) while the second digit is related rather to the specific core of each particular device, capturing its own hardware limitations. Table 8.1 lists all Nvidia GPU generations to date.

GPU architecture	Release year	CC
Nvidia Tesla	2007	1.x
Nvidia Fermi	2010	2.x
Nvidia Kepler	2012	3.x
Nvidia Maxwell	2014	5.x
Nvidia Pascal	2016	6.x
Nvidia Volta	2018 (being released)	7.x

Table 8.1: Generations of Nvidia GPUs and their Compute capability.

Since the first generation of Nvidia Tesla¹ GPUs (devices of CC 1.x), the platform is a subject of a continuous and still rapid development. Each particular generation of Nvidia devices does have its characteristic properties and limitations, both logical and physical. In general, nevertheless, the basics of the Nvidia SIMD programming model remain similar. Therefore, it will be beneficial to lead the upcoming chapter in an intentionally generalized manner. That way, this chapter should sustain relevant longer.

There await topics of newly developed hardware capabilities that are related to a particular GPU generation. In that case, the current Pascal architecture (CC 6.x) [62] is going to be set in contrast with its ancestor, the successful Kepler generation (CC 3.x) [63], to illustrate the direction of development of the platform as a whole.

¹Because the Tesla architecture was the first to support CUDA, the entire line of Nvidia’s GPGPU devices was also given the name *Tesla*. This may confuse the reader slightly when seeing a reference e.g. to Tesla Kepler or Tesla Pascal GPUs.

8.1 CUDA execution model

First, let us start with description of the CUDA execution model. The fundamental component of the model is the *kernel* function. In CUDA C/C++ programming language, kernel is a kind of a C function that is executed N times in parallel by N CUDA *threads*, see [64].

During the execution of the kernel function, each thread carries its specific thread index denoted as `threadIdx`. The index of each thread often specifies the actual task it is scheduled to execute. Proceeding to the next layer of the model, threads are then sorted into blocks of threads, called *thread blocks*. Much like threads, thread blocks also possess their very own indexes denoted as `blockIdx`. For convenience, thread blocks can be set as one (x), two (x, y) or three (x, y, z) dimensional objects. Finally, even thread blocks can be sorted into one, two or three dimensional arrays, in the CUDA terminology called *grids*.

Such a hierarchy creates a rather convenient framework for handling vector, matrix or higher-dimension matrix computation. Figure 8.1 illustrates the concept on a 2D grid of 2D thread blocks:

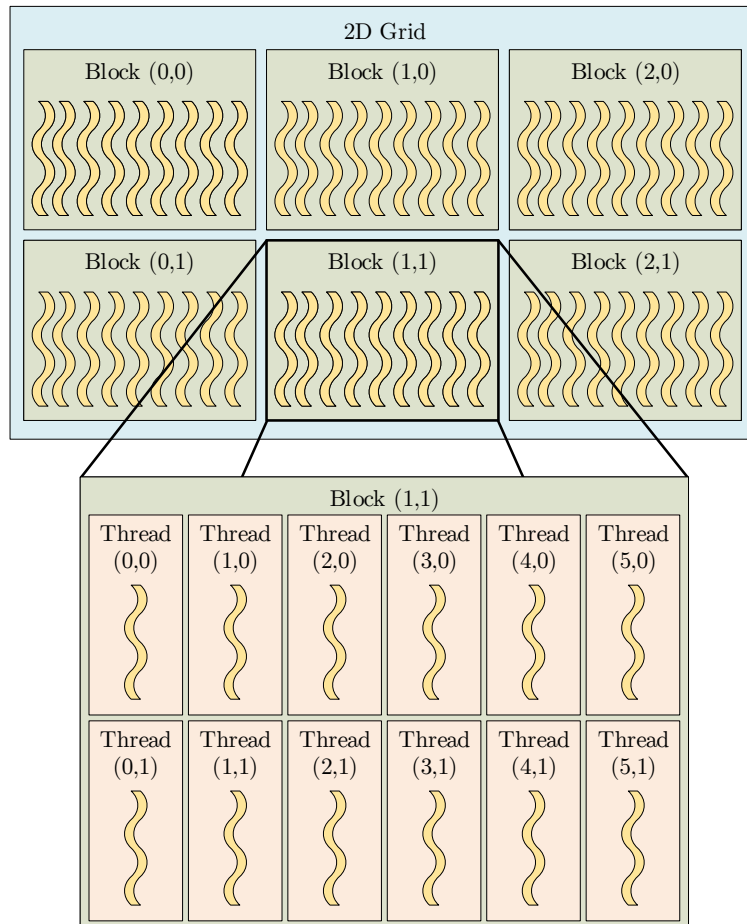


Figure 8.1: Illustration of a two dimensional grid of two dimensional thread blocks.

In practice, each thread computes its own global index based on its local `threadIdx` in its respective block and sizes of blocks (`blockDim`) and grids (`gridDim`) in each their dimension². In the simplest case of 1D grid of 1D blocks, the global thread index is computed as follows:

```
1 int thrdIdxGlb = threadIdx.x + blockDim.x * blockIdx.x;
```

Code 8.1: Computation of global thread index within 1D grid of 1D blocks.

In case of the scenario of 2D grid of 2D thread blocks, as illustrated by Figure 8.1, the computation of thread global index further evolves in the following way:

```
1 int thrdIdxGlb = threadIdx.x +
2               + blockDim.x * threadIdx.y +
3               + blockDim.x * blockDim.y * blockIdx.x +
4               + gridDim.x * blockIdx.y;
```

Code 8.2: Computation of global thread index within 2D grid of 2D blocks.

8.1.1 SIMD/SIMT operation

As was already mentioned in the introduction of chapter 7, the CUDA execution model is in its essence a kind of SIMD (Single Instruction Multiple Data) architecture. Every instruction that is executed in a CUDA kernel is an instruction of SIMD type. This means that it always operates upon a vector region of data. The size of this vector is called the SIMD width. Essentially, it denotes the number of threads that execute the same instruction in parallel. Currently, the SIMD width of all CUDA-capable hardware is 32. In effect, for instance, an add instruction always operates arrays of 32 elements: it adds to each of 32 numbers from register A each of corresponding 32 numbers from register B and stores 32 results in register C. Such an operation is in CUDA natively supported for 32-bit elements, i.e. single precision float values.

This kind of operation may be understood as (i) a vector operation:

$$\mathbf{c} = \mathbf{a} + \mathbf{b}, \tag{8.1}$$

or (ii) as 32 independent scalar additions:

$$\begin{aligned} c[0] &= a[0] + b[0], \\ c[1] &= a[1] + b[1], \\ &\vdots \quad \vdots \quad \vdots \\ c[31] &= a[31] + b[31]. \end{aligned} \tag{8.2}$$

In the actual CUDA execution, truly executed is the vector addition (8.1) by a single *physical thread* that operates with the vector registers A, B and C. The operation is conducted by one of multiple (or many) multithreaded processing units of the GPU. In the Nvidia terminology, one of *Streaming Multiprocessors* (more in Section 8.3). The 32 independent scalar additions (8.2) that are specified in the code by the programmer:

²In CUDA terminology, the size of blocks and grids is actually denoted as *dimension*. However, the term of "dimension" shall be reserved for the actual measure of *dimensionality* of blocks and grids to avoid confusion.

```
1  c [thrdIdxGlb] = a [thrdIdxGlb] + b [thrdIdxGlb];
```

Code 8.3: CUDA C interpretation of a vector addition.

are in fact an abstraction of tasks of 32 *logical threads* that populate the SIMD width that is governed by a *single physical thread*. In other words, workloads of each 32 logical (scalar) threads of the CUDA programming model are executed by a single physical (vector) thread. In CUDA language, the logical threads carry the title *thread* (or CUDA thread). Each described bundle of 32 threads is called a *warp* that is always executed by one *physical thread*.

8.1.2 Handling conditionally divergent code

Assuming that all threads in a warp are scheduled to execute an identical instruction, for example Code 8.3, the architecture behaves truly as a SIMD platform. In the following example Code 8.4, however, certain threads may fulfill the `if()` condition and take the code branch A, other threads are going to execute the code branch B:

```
1  if (condition) {
2      //code branch A
3      a [thrdIdxGlb] *= a [thrdIdxGlb];
4      b [thrdIdxGlb] *= b [thrdIdxGlb];
5      c [thrdIdxGlb] = a [thrdIdxGlb] + b [thrdIdxGlb];
6  }
7  else {
8      //code branch B
9      a [thrdIdxGlb] += b [thrdIdxGlb];
10     a [thrdIdxGlb] *= a [thrdIdxGlb];
11 }
```

Code 8.4: An example of a divergent code.

Because at the time of code compilation it is not possible to determine which thread is going to execute which branch of the code, the GPU has to schedule execution of these branches of *divergent code* on the fly. SIMD hardware that is capable of such a mapping of different branches of code during execution is known as SIMT (Single Instruction Multiple Threads).

If there exist multiple execution paths that threads within the same warp may take, the execution of these different instructions is then scheduled in series. Still, all threads in the warp (i.e. SIMD width) have to conduct the same instruction in the spirit of SIMD with no branching possible. The multithreaded processing unit satisfies this condition by deactivating the respective threads for each branch of code by using the *active bit mask*, as will be explained.

For simplicity, let us now consider a warp of 4 threads and first examine the SIMD execution of Code 8.3. In this case, all threads are meant to execute the very same bit of code. The resulting active mask is then 1111 as all threads are supposed to be active all the time. Figure 8.2 shows the actual execution of Code 8.3 as written in the Assembly language (\approx machine code).

In contrast, the execution of the divergent Code 8.4 has to be more complex than that, as further explained in Figure 8.3. First, the code should declare the beginning of an `if-else`

Instruction	Threads				Warp	
	#0	#1	#2	#3	Order	Mask
I.ADD R3, R1, R2	0	0	0	0	0	1 1 1 1

Figure 8.2: Assembly code of the SIMD execution of Code 8.3.

branching, that is where the threads are expected to diverge. The branching is here declared by the *set-synchronization bit* by the `SSY ENDIF` instruction. Effectively, the *synchronization token* is pushed onto the stack, containing information (active mask) about threads that were active before the branching was encountered. That way, the multithreaded processing unit is able to keep track of all threads that are supposed to execute any branch of the divergent code. As soon as the active mask at the end of divergent code corresponds with the synchronization token, all threads are considered to be successfully synchronized back. Complementarily, each branch of code shall declare its end, that is where the particular code returns (synchronizes) back. The synchronization is declared by the `.S` suffix for the last instruction in each branch.

Before executing the conditionally divergent code, each thread needs to decide which branch, if any, it is supposed to execute by evaluating the `if()` condition. This is done by a predicated branch instruction (the *predication* feature). In Figure 8.3, the boolean result of the condition is held in the register `P0`. The thread selects the `if` branch if `P0==true` or the `else` branch if `P0==false`. According to the evaluation of the predicate register `P0`, active masks for each code branch are assembled, see Figure 8.3.

	Instruction	Threads				Warp	
		#0	#1	#2	#3	Order	Mask
	SSY ENDIF	0	0	0	0	0	1 1 1 1
	@!P0 BRA ELSE	1	1	1	1	1	1 1 1 1
IF:	I.MUL R1, R1, R1	2			2	2	1 0 0 1
	I.MUL R2, R2, R2	3			3	3	1 0 0 1
	I.ADD.S R3, R1, R2	4			4	4	1 0 0 1
ELSE:	I.ADD R1, R1, R2		2	2		5	0 1 1 0
	I.MUL.S R1, R1, R1		3	3		6	0 1 1 0
	ENDIF	5	4	4	5	7	1 1 1 1

Figure 8.3: Assembly code of the SIMT execution of Code 8.4.

Now, let us assume that the result of the `if()` condition is that the threads `#0` and `#3` execute the `if` code branch and the threads `#1` and `#2` execute the `else` code branch. As already discussed, the multithreaded processing unit is going to execute the `if` and `else` branches in series. Suppose that the `if` branch is going to be executed first because the first thread in the warp (also called the *warp leader*) satisfies the `if()` condition. Then, before executing the `if` branch with the active mask `1001`, a divergence token containing the active mask `0110` for the `else` branch is pushed onto the stack. At this moment, the state of the warp is as follows:

```
Execution state: (IF, 1001)
Divergence stack: (ELSE, 0110), (ENDIF, 1111)
```

After that, the `if` branch is executed and the `else` branch is executed as soon as all threads with the previous mask 1001 are synchronized back. Threads with the mask 1001 are deactivated and the state of the warp becomes:

```
Execution state: (ELSE, 0110)
Divergence stack: (ENDIF, 1111)
```

Additionally, the described procedure and Figure 8.3 are accompanied by the flow diagram in Figure 8.4, see also [65]. In similar fashion as the conditional branch divergence are handled other instances of divergent code such as divergence at return and break instructions, see further [66–68].

As should be apparent from the above discussed, any kind of divergent code typically causes a significant performance drop of the SIMD/SIMT execution. That is due to the inability of multithreaded processing units to truly handle each thread individually. One of typical challenges of GPGPU programming therefore is to maximize the execution efficiency by avoiding such an inconsistent code, especially due to conditional branching.

In practice, however, each implemented problem possesses its characteristic inevitable routines. For example, consider the actual implementation of computation of periodic distance between two particles, see Equation 5.6:

```
1  (...)
2  float absoluteDist = 0.0f;
3  float partialDists[nvar];
4  // computation of squared absolute distance between particles i and j
5  for (int v = 0; v < nvar; v++){
6      partialDists[v] = gpu_coordinates[j,v] - gpu_coordinates[i,v];
7      //inevitable conditions due to periodic domain
8      if ( abs(partialDists[v] - 1.0f) < 1.0f ) partialDists[v] -= 1.0f;
9      else if ( abs(partialDists[v] + 1.0f) < 1.0f ) partialDists[v] += 1.0f;
10     //
11     absoluteDist += partialDists[v] * partialDists[v];
12 }
13 (...)
```

Code 8.5: Implementation of periodic boundary conditions.

There is no possibility for the programmer to predict the distance between particles within the system in advance (or for the compiler at compile time or for the driver during execution). Therefore, such a computation inevitably requires each thread to pass through conditional code on lines 8 and 9 in Code 8.5.

Obstacles of a similar flavor also arise when accessing the GPU memory. The CUDA memory model is going to be explained in what follows.

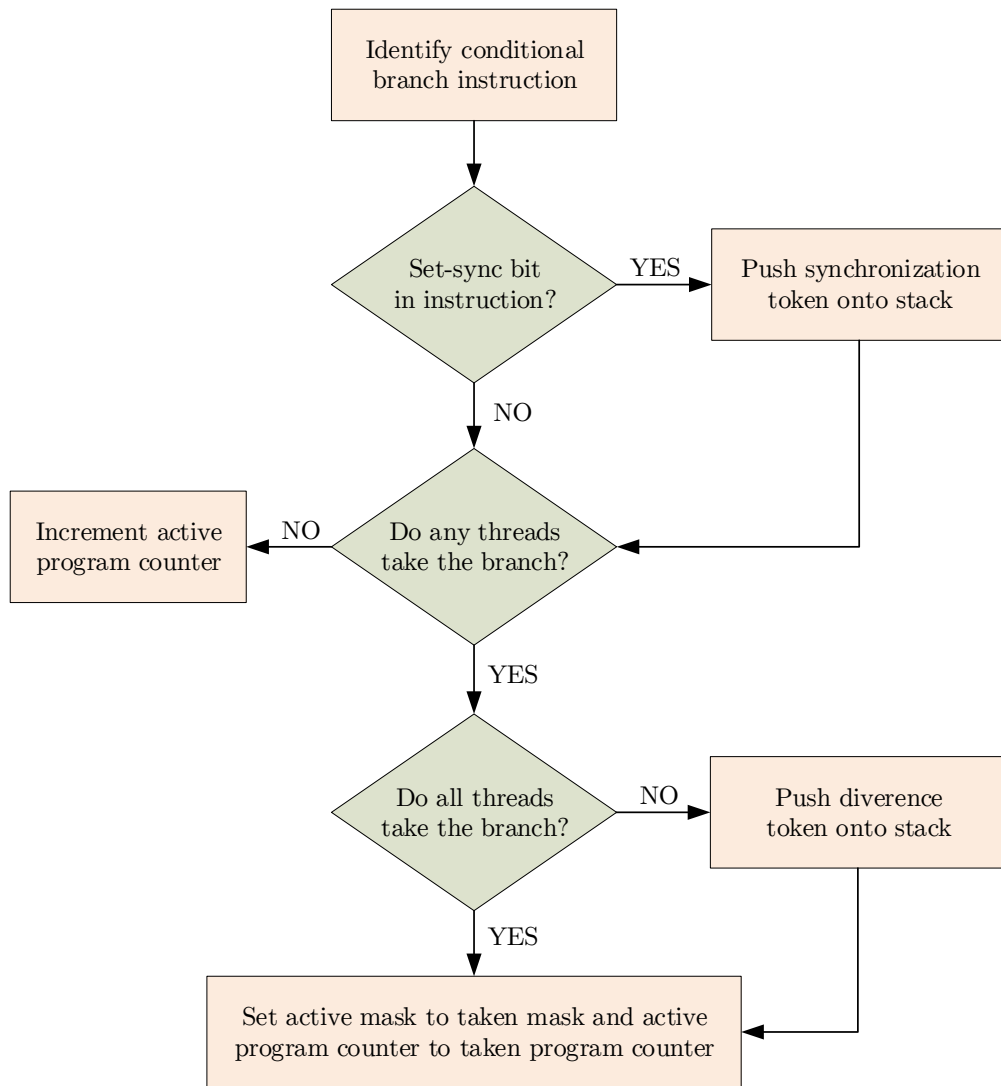


Figure 8.4: A flow diagram of the management of a group of active threads through a conditionally divergent code.

8.2 CUDA memory model

In order to fully exploit the computational (arithmetic) potential of GPU core, a sufficient data throughput from GPU memory to GPU core is essential. First, let us examine the memory hierarchy of a CPU-GPU system. A CPU-GPU system operates with two essential kinds of memory storage:

- the *host memory* that is operated by the CPU, i.e. the operational memory of the computer. Typically of the DRAM (Dynamic Random Access Memory) type, see e.g. [69].
- the *device memory* that resides on the GPU board. The device memory has its own complex physical and logical hierarchy that will be discussed further in this section. Generally, the on-board device memory is of the DRAM type, similar to host. The fast and limited on-chip device memory is of the SRAM type (Static Random Access Memory), see e.g. [70].

From the memory point of view, a typical GPGPU execution consists of the following steps: (i) copy input data from host to device (HtD), (ii) execute the scheduled GPU labor, (iii) copy the output data from device back to host (DtH). As is already apparent from Section 7.1, data transfers between host and device memory are operated by the PCIe bus that has become a weak spot of the entire CPU-GPU assembly. Any HtD/DtH copying other than (i) and (iii) shall be carefully considered as these might consume a major portion of the total execution time. Based on the gained experience of the community of GPGPU programmers, three empirical rules are emphasized, see also [71]:

1. Get the data on the GPU and keep it there.
2. Give the GPU enough work to do.
3. Focus on data reuse within the GPU.

Even after successfully copied on the device, the way the data is accessed crucially influences the resulting performance. Naturally, a memory storage of a high bandwidth, high capacity and minimal access latency is desirable, however not entirely feasible due to e.g. construction or economical reasons. To achieve a compromise solution, the GPU uses its specific memory hierarchy based on the expected programming model that obeys following *assumptions of locality*:

- *temporal locality* is an assumption that if a certain chunk of data is once requested, it is expected to be much more likely referenced again within a short time span rather than after a long period of time,
- *spatial locality* is an assumption that if a certain chunk of data is requested, the data within a close proximity is much more likely to be requested as well.

By expecting such a programming model, the GPU achieves a high memory bandwidth employing multiple types of memory ranging from low-latency/low-capacity to high-latency/high-capacity. Put simply, if the data is currently operated by the GPU, it shall be kept in the low-latency/low-capacity memory. Conversely, when the data is only stored for future use, it should reside in the high-latency/high-capacity memory. Contemporary Nvidia GPUs are equipped two basic kinds of memory: the fast but limited *on-chip* memory and the slow but plentiful *on-board* DRAM memory.

Form the programming point of view, CUDA-capable GPUs are equipped with two basic types of memory: (i) *programmable memory* (software managed) where the programmer does have the control over the data disposition and (ii) *non-programmable memory* (hardware managed) which does not allow the programmer to handle the data placement. The non-programmable memory uses a kind of automatic allocation, often guided by heuristics. In case of Nvidia GPUs, all caches are non-programmable.

Before going into detail, Table 8.2 submits to the reader all the GPU memory and cache types along with their physical and logical properties.

Programmable memory	Location	Access	Available for	Lifetime
Registers	On chip	R / W	Single thread	Thread
Shared memory	On chip	R / W	Threads in block	Block
Global memory	On board	R / W	All threads and host	Host alloc
Local memory	On board	R / W	Single thread	Thread
Texture memory	On board	R	All threads and host	Host alloc
Constant memory	On board	R	All threads and host	Host alloc

On-chip cache	Data cached	Caching	Available for
L1 cache	Global and local memory	Reads	Threads in block
L2 cache	All memory transactions	Reads / Stores	All threads
Texture cache	Texture memory	Reads	Threads in block
Constant cache	Constant memory	Reads	Threads in block

Table 8.2: Memory types of contemporary Nvidia GPUs.

The main *GPU memory* as understood throughout Section 7.1 resides in the GPU on-board DRAM memory. The DRAM storage of current Pascal GPUs reaches up to 24 gigabytes. Therefore, it is both the largest and the slowest memory storage of the GPU. The physical DRAM memory is able to contain several logical types of memory as discussed further.

8.2.1 Global memory

The *global memory* is the most used kind of GPU memory in GPGPU. Certain authors even entitle the entire DRAM memory space as global memory. Global memory is similar to the *heap* in a C program. It can be accessed by any running thread and its lifetime can be up to its CUDA program execution time, hence the title *global*. Global memory is both allocated and freed exclusively by the host, using the `cudaMalloc` and `cudaFree` methods. Typically, global memory is used for storage of the portion of data that is expected to be read as well as overwritten during the program execution.

Global memory can be accessed in 32 or 128-byte linear (1D) memory transactions depending on the caching used (L2 and/or L1 cache, will be discussed later). These transactions must be aligned in the way that the first address must be a multiple of 32 or 128 bytes. A global memory access can be requested by any thread in any thread block. However, the entire warp requests a homogeneous read instruction for a chunk of data, similarly to han-

dling divergent threads. The performance of reading global memory depends on the number of such transactions that are needed to satisfy such a read request.

Because global memory can be accessed by any thread in any thread block, difficulties arise as the execution of thread blocks is not synchronized across the grid. Such read requests of multiple threads (commonly thousands of threads or more) might be extremely scattered across global memory, possibly causing severe performance drops. Conversely, major obstacle of possible unpredictable behavior emerges when multiple threads invoke writing into the very same global memory address. These crucial difficulties that lurk at the programmer arranging data in global memory will be addressed in what follows.

8.2.2 L1 and L2 caches

L1 and L2 caches are non-programmable, hardware controlled caches³. Their purpose is to save time-expensive accesses to DRAM memory based on the *assumptions of locality*. After registers, the L1 cache is in the closest proximity of each multithreaded processing unit (\approx thread block), i.e. the memory storage with the lowest latency. The L2 cache follows as a shared cache for the entire GPU processor (\approx grid).

Both L1 and L2 cache are non-programmable memories. Their content is therefore managed by the hardware, assembled due the operation of the memory bus. It is because any global memory transaction to a thread that requests even a single value does not contain only this value. Global memory transactions are unable to carry data of any desired size. Instead, data is moved in chunks of a given *granularity* that corresponds with the used caching method. These chunks, if the load is cached, are kept in the cache until they are overwritten by another cached load. The management and allocation of cached data is beyond programmer's control.

Contemporary Nvidia GPUs distinguish the granularity of memory transactions for L1 and L2 caching. The L2 cache is always enabled and transmits all memory transactions of the GPU, using granularity of 32 byte chunks of data. L1 caching can be explicitly enabled or disabled by the programmer⁴. Because the L1 cache uses 128-byte memory transactions, its contribution varies depending on the data arrangement in global memory.

In practice, if a thread requests a value from global memory, it first investigates whether the value is currently held in the L1 cache as a part of any of previous memory loads. If yes, the value is read from the L1 cache, resulting in an *L1 cache hit*. If the value is not present in the L1 cache, the request results in an *L1 cache miss* and is forwarded to the L2 cache. Again, the value might be already contained in the L2 cache. If yes, an *L2 hit* is achieved and the value loaded from the L2 cache. If not, an *L2 cache miss* is obtained and the value has to be requested from the slowest storage that is the global memory.

Note that L1 and L2 caches are meant for *spatial locality*, not temporal. Meaning that the probability of a value being available in L1 or L2 cache does not depend on how frequently the value is requested.

³The word "cache" is in this context often replaced by the "\$" sign, primarily for space saving purposes.

⁴Note that this does not make it a programmable cache.

8.2.3 Accessing global memory

It has been already discussed that the global memory is the backbone of GPU memory hierarchy, typically containing a majority of data of every CUDA process. Hence, a maximal efficiency of global memory loads is always the cornerstone of optimization efforts. The efficiency of global memory accesses strongly influences the resulting performance by orders of magnitude. With poor global memory efficiency, any additional speedup using caches or shared memory is rendered negligible.

Similarly to the execution of arithmetics, accessing any kind of GPU memory is not conducted independently for each logical thread but is issued per warp. Memory requests from all 32 threads in warp form a single memory access request that will be serviced by one or multiple memory transactions. The number of necessary transactions depends on the scatter of requested values across the memory as well as on the granularity of the used caching method. L2 caching is always enabled (non-optional), forcing granularity of chunks of data of multiples of 32 bytes. If L1 caching is enabled by the programmer, memory transactions are cached by both L1 and L2 caches, using the granularity of L1 cache that is multiples of 128 bytes (four L2 segments aligned).

For both options (L2 or L1 and L2), cached memory loads are always *aligned* to begin on multiples of 32 or 128 bytes, respectively. If each thread in warp requests a 4-byte word (e.g. one float value), these requests (continuous 128 bytes in total) ideally fill the entire L1 cached load, resulting in a single 128 transaction to L1 and L2 cache (if L1 and L2 cached) or in four 32 byte transactions to L2 cache (if L2 cached).

The values requested by threads in a warp can be scattered or ordered in global memory. If all requested values ideally fit within the scope of a single memory transaction, see Figure 8.5a, the best scenario is achieved as all requested values are delivered at once within this memory transaction. Conversely, all values transmitted within the cached load are requested, meaning that the overall efficiency of memory bandwidth is 100%.

If the values requested are not ordered by the index of thread but still form a continuous chunk of a single memory transaction, the memory request is serviced by a single memory load as well, see Figure 8.5b. The obtained bandwidth efficiency is still 100%. Such effective memory accesses always result from a consistent programming approach and perfect understanding of the implemented problem. This kind of data arrangement and memory accessing is known as *coalesced memory accessing* and is essential for well performing code.

Figure 8.5c complements the list with a *non-coalesced* memory request. As is apparent, such a scenario would result in three 128-byte L1 loads⁵ or six 32-byte L2 loads⁶. The efficiency of these loads would be rather low as only 32 four byte values are truly requested from the 384 or 192 bytes transmitted. The efficiency of such memory transfers is then 33% (L1 and L2 caching) or 67% (L2 caching only). Here it already becomes apparent that the selected caching method may strongly affect the efficiency of memory reads.

In the described instance, the estimated bandwidth efficiency of L1/L2 caching is significantly lower compared to L2 caching alone. The actual performance, however, also depends on instructions that follow after these memory loads. Indeed, only 33% of the L1 loads are actually asked for by the warp and the remaining 67% will end up resting in the L1 cache for no reason. Nevertheless, in case of a kind of coherent implementation approach, the con-

⁵Additional two loads of chunks from 0 to 128 bytes and from 256 to 384 bytes are issued.

⁶Additional two loads of chunks from 96 to 128 bytes and from 256 to 288 bytes are issued.

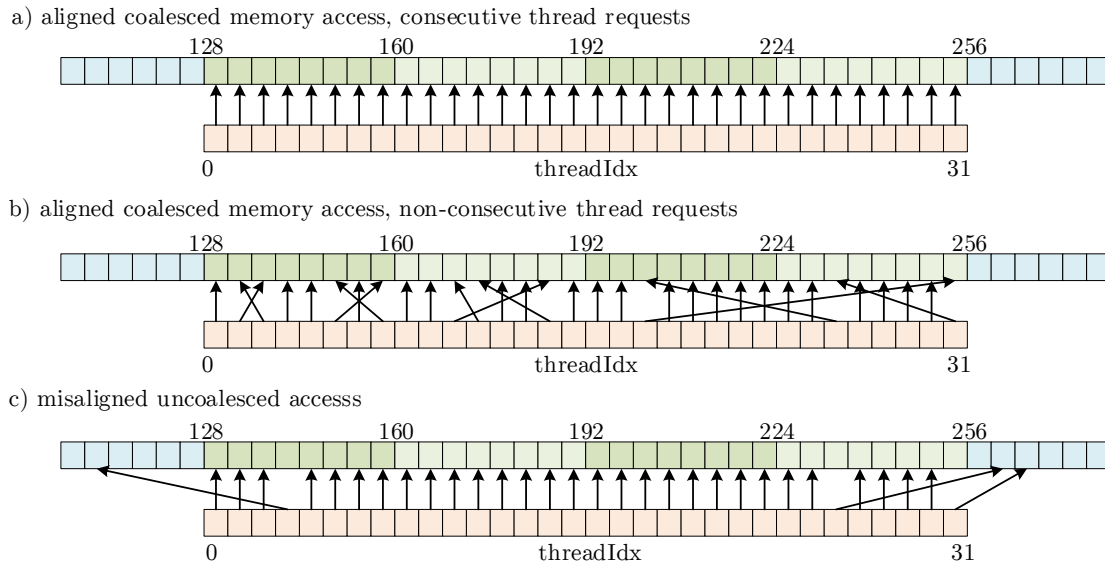


Figure 8.5: Examples of coalesced and non-coalesced memory requests.

tent of cached loads from 0 to 128 bytes and from 256 to 384 bytes can be referenced in following memory requests, resulting in many L1 cache hits with very low latency. The topic of the influence of caching is discussed further.

Bandwidth efficiency of various memory access patterns will be now compared for caching set on L1 and L2 caches or L2 cache only. The best possible solution is an *aligned and coalesced* memory access. Meaning that the whole warp requests a continuous chunk of memory whose beginning is aligned to a multiple of 128 (L1) or 32 bytes (L2). In this case, bandwidth efficiency is 100% for both L1/L2 and L2 caching, compare Figures 8.6a and 8.7a. The 100% bandwidth efficiency is reached also for a randomized access (for devices of compute capability 2.0 and beyond, as long as all values assemble a continuous memory line, see Figures 8.6b and 8.7b).

Another example studies a coalesced access to a continuous 128 byte chunk of data, beginning of which is not aligned to an exact multiple of 128 bytes. When cached by L1 and L2 caches, such a memory request is serviced by two 128 byte transactions with bandwidth efficiency of 50%. However, when cached by L2 cache only, the request is serviced with five 32 byte transactions with bandwidth efficiency of 80%, compare Figures 8.6c and 8.7c.

If all threads in warp happen to request the very same value from global memory, a single caching load of 128 or 32 bytes is needed to service this request. The bandwidth efficiency, however, turns out quite low as only 4 bytes from 128 or 32 byte load are actually asked for. For L1 and L2 caching, the efficiency falls to 3.125% and 12.5% if cached only by L2 cache, see Figures 8.6d and 8.7d.

The last example shows a request for 128 bytes that are scattered randomly across global memory, see Figures 8.6e and 8.7e. In the worst case of each value being in a different cache line, this memory request might end up with efficiency of 3.125% if cached by L1 and L2 caches (loading $32 \times 128 = 4096$ bytes instead of 128 bytes). If cached only by L2 cache, the worst case exhibits efficiency of 12.5% (loading $32 \times 32 = 1024$ bytes instead of 128 bytes).

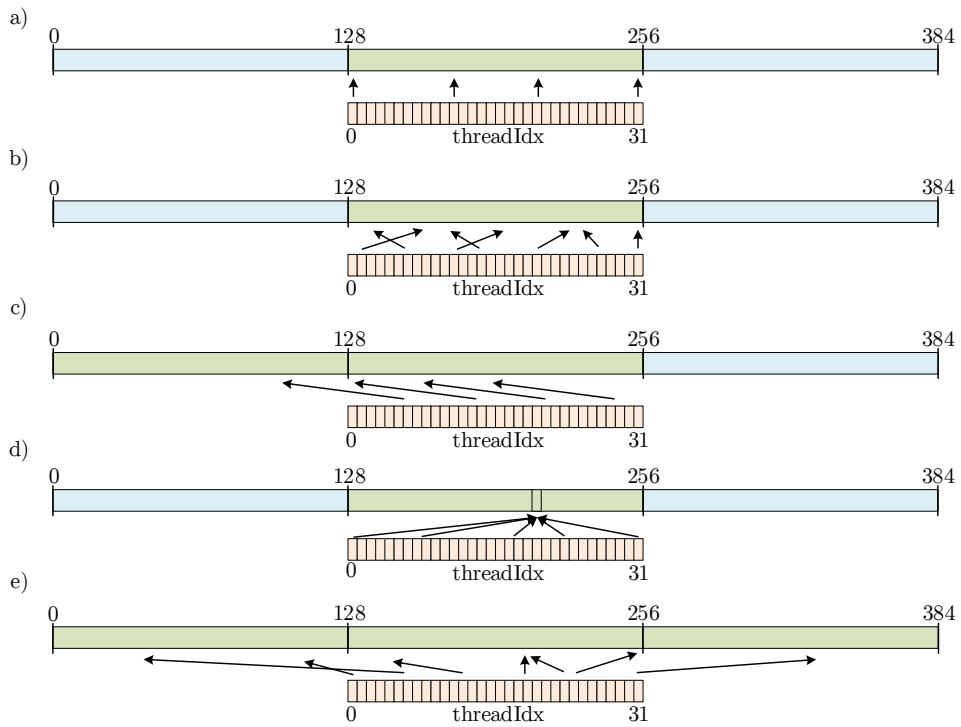


Figure 8.6: Examples of various memory loads cached in L1 and L2 cache.

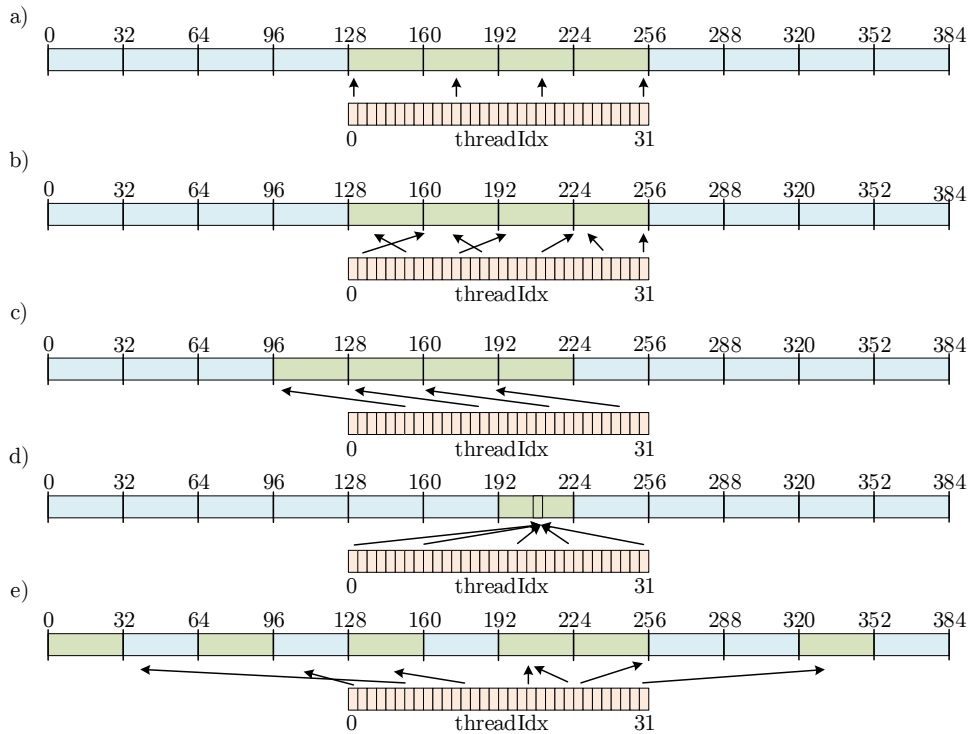


Figure 8.7: Examples of various memory loads cached only in L2 cache.

As is apparent, the arrangement of data in global memory has to be carefully considered at the very beginning of every implementation. Depending on the problem at hand, the programmer should aim for a maximal memory bandwidth utilization. A typical dilemma is choosing between the *array of structures* (AoS) or *structure of arrays* patterns. In parallel programming, the latter is much more suited for coalesced memory accessing, see Figure 8.8.

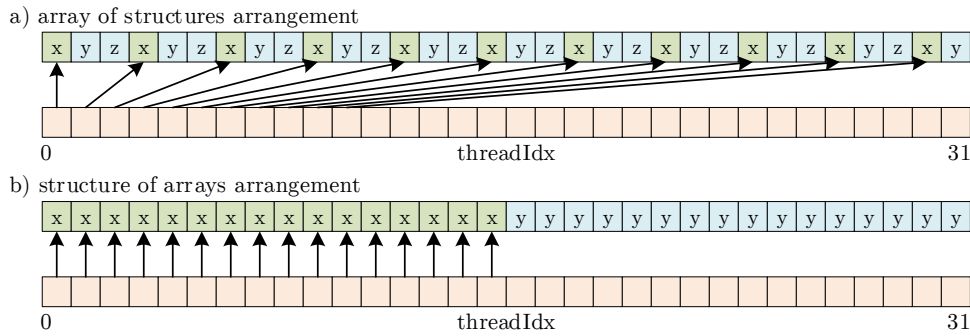


Figure 8.8: Data arrangement into (a) an array of structures or (b) a structure of arrays.

8.2.4 Texture memory and texture cache

The *texture memory* is a distinctive logical type of the on-board DRAM memory of the device. It is sometimes referred to as a kind of global memory. The purpose of texture memory is to accommodate read-only, constant data. In graphics usage, such a kind of constant data are the textures of a rendered model, hence the title *texture*.

The texture memory technically resides in the identical storage as the global memory. There is, however, a difference in the way the texture memory is accessed. Unlike the global memory that is designed for linear accessing, the texture memory is optimized for 2D or 3D spatial locality. The texture memory uses its own dedicated texture cache that caches 2D or 3D 64-byte chunks of memory.

Applications where threads within the same block access the memory in regular 2D or 3D patterns might benefit from such a memory arrangement. Further, texture memory offers a possibility of out-of-bounds index handling (bordered or periodical) and hardware floating-point interpolation of values. Finally, utilization of texture memory for 1D loads is also possible. For kernels of high bandwidth demands, additional global loads through texture memory using the Parallel Thread Execution (PTX) instruction set [72] can be invoked. These loads then pass through the texture pipeline and are cached by the texture cache, relieving the stress on the L1 cache. The bandwidth of global and texture memory then adds up. In the GPGPU usage, texture memory is not used very often as the global memory loads typically exhibit better performance.

8.2.5 Constant memory

The *constant memory* is another logical type of the DRAM memory, or vaguely - global memory. As its name suggests, constant memory shall be populated by constants, i.e. data

that is written only once and read many times. Unlike global and texture memories that are limited only by the capacity of the DRAM memory, constant memory is limited to 64KB.

The purpose of the constant memory and its dedicated constant cache, respectively, is to quickly fetch values that are expected to be frequently requested by a large number of threads, e.g. coefficients and constant arguments of functions. The advantage of constant cache is in its ability to *broadcast* the same data to all threads in a warp on one request. That makes it a 1-to-32 kind of memory, as oppose to L1 or L2 caches and global memory which are 1-to-1, meaning that they return a single value on a single request. The broadcasting action requires only a *single cycle*. That might result in a significant speedup in comparison to the latency of global memory that is hundreds of cycles.

8.2.6 Local memory

Confusingly enough, the *local memory* resides in the on-board DRAM ("global") memory as well, despite its name. The title *local* refers to its logical scope that means *thread-private* rather than to its physical location.

The purpose of local memory is to accommodate a potentially overflowed (or *spilled*) *stack* of each particular thread. Therefore, local memory cannot be allocated by the programmer. It is dynamically allocated by the compiler or the CUDA driver for:

- spilled out data of a thread that runs out of register resources,
- arrays size of which cannot be resolved by the compiler at the compile time.

Generally, the usage of local memory is a sort of an undesired situation as it exhibits the same access latency as the global memory. The programmer shall carefully prevent register spilling (thread stack overflow) because data from registers, the fastest and most precious on-chip memory, is spilled into the orders of magnitude slower on-board DRAM. Similarly to global memory, local memory is cached by the L1 and/or L2 cache.

Register spilling into local memory is also closely related to divergent code, as discussed in Section 8.1. The more conditionally branched the code is, the more registers are allocated for the stack of each thread at the compile time. Because the compiler is unable to decide which thread is going to take which code branch, it allocates registers for all possible scenarios. The number of registers requested then easily exceeds registers available in total or per thread.

8.2.7 Shared memory

The *shared memory* is the cornerstone of an optimized GPU performance, right along with coalesced global memory accessing. Shared memory resides on-chip, in a close proximity to each multithreaded processing unit. It is a fast (low-latency) and quite limited *programmable cache* of the GPU. Shared memory exhibits very similar latency to the L1 cache. In the Kepler generation of Nvidia GPU, shared memory even uses the very same memory space as L1 cache.

Shared memory is allocated statically or dynamically by the programmer for each thread block. It can be accessed by all threads in the same thread block, hence the title *shared*. Its lifetime is equal to the lifetime of this thread block. All threads can access all data in shared memory, even that loaded from global memory by other threads (of the same thread

block). Each multithreaded processing unit operates its own shared memory resources that are divided between all thread blocks scheduled to this processor. Therefore, the amount of shared memory available for each thread block depends on the number of thread blocks that are scheduled to run concurrently on this processor. Conversely, moreover, the number of thread blocks that can run concurrently is limited by the shared memory requirements of each thread block.

Unlike the non-programmable L1 cache that is designed for spatial locality (caching data physically close to the data requested), the programmable shared memory aims for *temporal locality*, i.e. caching of frequently requested data. The programmer does have a full control over the data in shared memory, its loading, arrangement, and lifetime. Shared memory is the GPU memory resource that truly enables cooperative parallel algorithms between threads.

Accessing shared memory

Just like global memory, shared memory requests are serviced per entire warp. There is, however, a difference in the structure of shared memory. To achieve even higher bandwidth, shared memory is divided into separate memory modules, or *memory banks* in CUDA terms. The granularity of shared memory transactions is also different. It is expected that shared memory will be used for fine memory loads of individual values. Shared memory banks therefore operate directly with single 32-bit values called *words*. Devices of the Kepler architecture allow to manually switch to 64-bit words. Its successors, Maxwell and Pascal generations, do not enable such an option anymore. Each bank stores its words ordered one after another in a sort of layers or *drawers*. It is up to the programmer to determine in which order the words are stored. As will be explained, the word ordering in banks strongly affects the shared memory performance.

The number of banks is equal to the number of threads in warp. Each of 32 banks per warp can be accessed independently by one thread at a time. In the best case of different threads requesting words from different banks, all these requests are serviced by a single shared memory transaction (in a kind of SIMD manner). Such a coalesced shared memory accessing is very desirable as it exhibits almost minimal latency similar to registers. Similarly to global memory transactions, a shared memory request is coalesced as long each thread in block references a word in different bank. There are no restraints in the order of banks requested or drawer positions of requested values, granted that no bank is called by multiple threads. Figures 8.9a, 8.9b, and 8.9c all illustrate variants of conflictless shared memory requests that are serviced by a single memory transaction.

In the case of multiple threads requesting different words in the same bank, a *bank conflict* appears. Requests of these threads will be serviced sequentially, one after another. In the worst case, each of 32 threads in warp requests a different word in the same bank. This results in a 32 times slower solution that are 32 serialized memory transactions. Figures 8.10a, 8.10b, and 8.10c show serialized servicing of two-, three-, and four-way shared memory bank conflicts.

The last possible scenario is when multiple (2 to 32) threads in warp request the very same word in the very same bank. That way, a bank conflict of any kind does not arise. The word is obtained by a single memory transaction and the word is *broadcast*⁷ to all threads

⁷The term *multicast* is often used in literature, depending on the author.

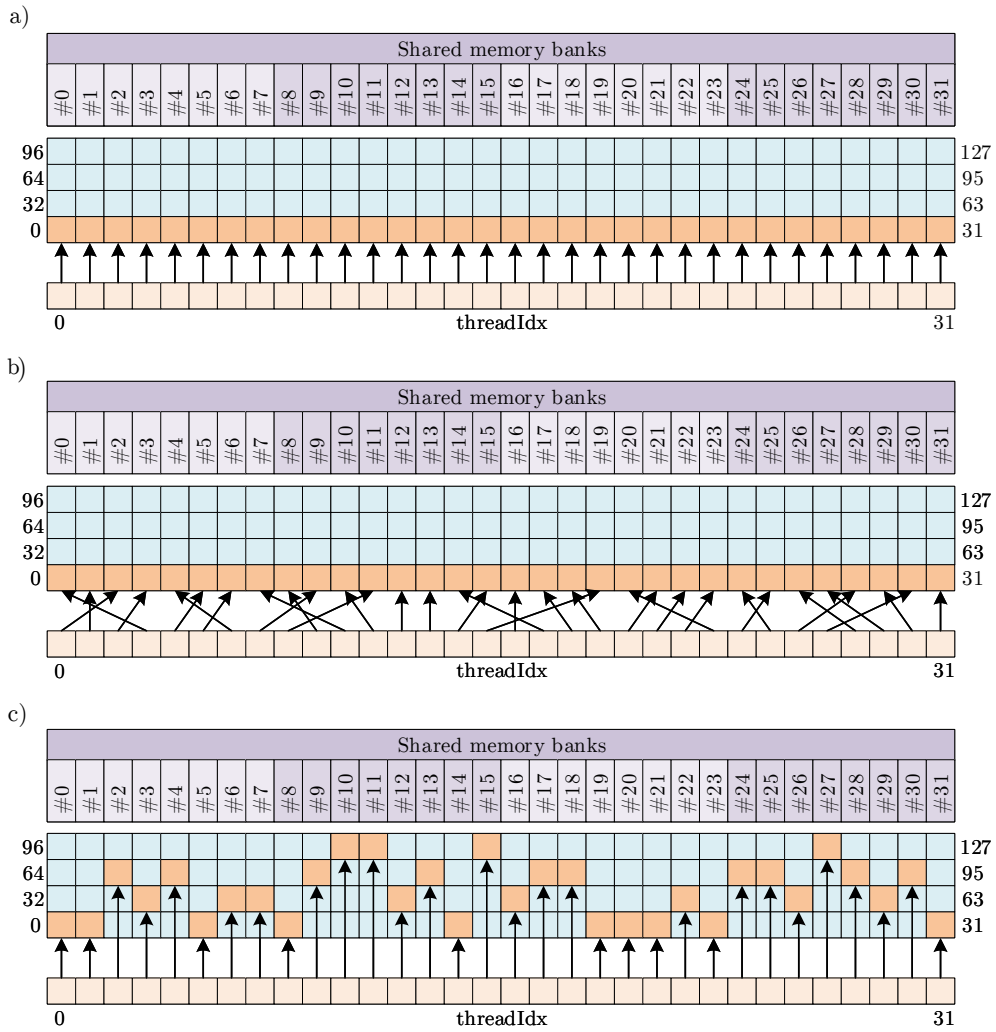


Figure 8.9: Conflictless shared memory requests serviced by a single memory transaction.

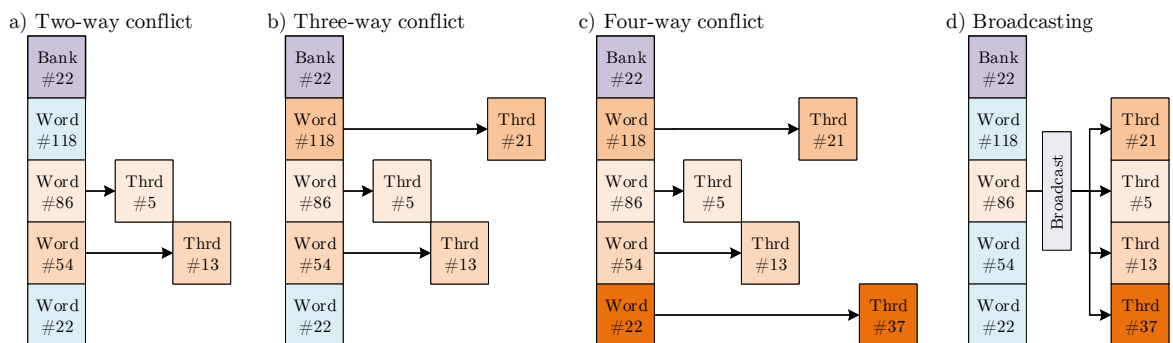
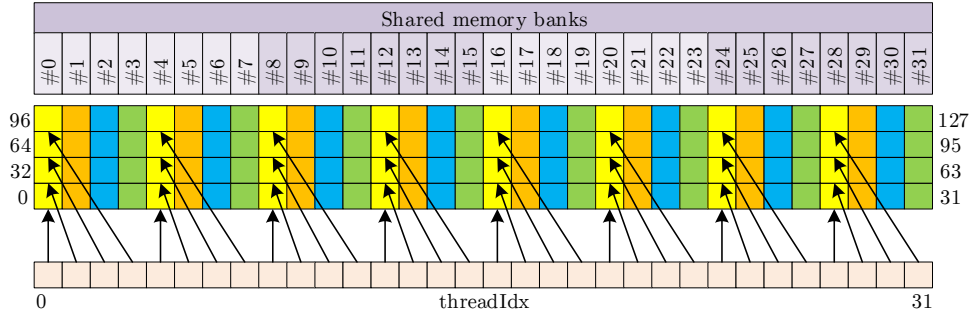


Figure 8.10: Examples of shared memory bank conflicts and broadcasting of a value requested by multiple threads.

it is requested by. The latency of a broadcast access is equal to one memory transaction.

a) Perfectly aligned data in shared memory resulting in four-way bank conflicts.



b) One-word shared memory padding resulting in no bank conflicts.

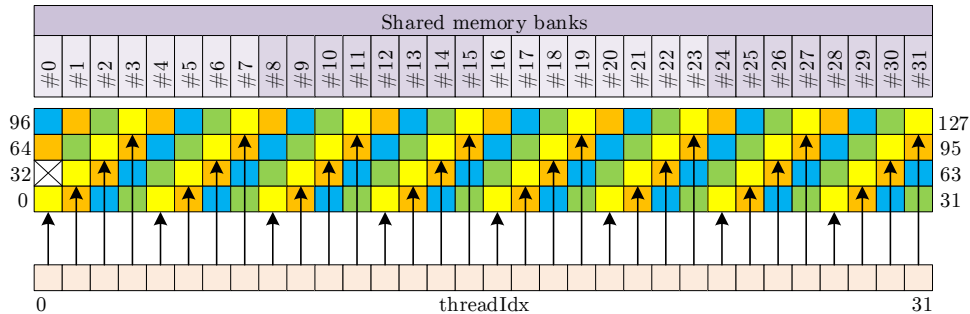


Figure 8.11: Conflictless arrangement of data in shared memory banks by memory padding.

For the programmer, it is vital to avoid bank conflicts during the code execution. Ideal solutions are either to uniformly access different banks with different threads or broadcasting the same word in one bank to all threads at once. The bank where a value is stored can be easily checked [73]:

```

1 bankIdx32 = (byte address / 4 bytes per bank) % 32 banks;
2 bankIdx64 = (byte address / 8 bytes per bank) % 32 banks;

```

Code 8.6: Computation of a bank index.

Certain types of implementations lead to massive bank conflicts and might not allow for such a simple and efficient use shared memory. In such cases, *shared memory padding* comes in handy. Its purpose is to avoid bank conflicts by a clever arrangement of data across shared memory banks. Figure 8.11 illustrates an one-word memory padding to avoid four-way bank conflicts.

8.2.8 Registers

The *registers* are the fastest type of the entire GPU device. These resources are in the closest proximity to the actual execution cores which makes them extremely fast as well as extremely limited memory. Due to their location, registers are the only GPU memory type that has low enough access times (latency) to keep up with the arithmetic horsepower of the GPU.

Registers are thread-private memory with the lifetime of each respective thread. Registers are used by each thread to hold currently operated values, constants and frequently used data. The register allocation takes place at compile time (by the compiler) and also during the code execution (by the driver) to satisfy possible requirements of divergent code. Despite seemingly plentiful register resources of the device (14 336 KB of registers in Tesla GP100 means 3.58 millions of 32-bit registers), the programmer should keep in mind that these are to be divided among thousands or even millions of threads. Due to these and also hardware limitations, there is set a maximum number of registers per thread. Since the Kepler architecture, the maximal number of 32-bit registers per thread is limited to 255.

Similarly to shared memory resources, the number of registers available per thread is limited by the number of threads that are scheduled to run concurrently. And conversely, the number of threads possible to execute concurrently depends on their register consumption. This means that the currently most powerful Tesla GP100 is able to run only around 14 thousands of threads that request 255 registers each. In scales of massively parallel programming, that is not an unusually high number.

Registers can hold single words, vector types and even arrays. Arrays can be stored in registers only if their length is constant and, moreover, can be determined at compile time. If it is not possible to estimate their size at the compile time, the compiler automatically allocates (*spills*) them in local memory. As already discussed, that is not desirable as the latency of local memory (DRAM) is orders of magnitude higher. Similarly as arrays of unknown size are spilled into local memory values that exceed either registers available per thread or the maximum of registers per thread.

On one hand, it is highly desirable to utilize this fastest memory to keep the arithmetic execution units busy. However writing code with minimal register demands should be always one of goals of the programmer. Generally, low register pressure means that more thread blocks can be serviced concurrently per one Streaming Multiprocessor. This further results in more warps being active at the same time, increasing the *occupancy* of the device. Nevertheless, a higher occupancy may or may not necessarily mean higher performance, see e.g. [74]. Therefore there is no universal guide and the path of pursuit of maximal performance may differ for each implementation.

Register shuffle

It has been made clear above that registers are thread-private memory. Therefore, only the thread is permitted and able to read and write its own registers. The consequence is that if a kernel implements any kind cooperative parallel algorithm between threads, any exchange of updated values between threads has to be conducted using shared memory or worse, the global memory.

Nevertheless, since the Kepler generation of Nvidia GPUs, there exists a possibility of migration of words between registers of threads in the same warp by *register shuffle* methods. Although, this does not mean that threads can read each other's registers. Selected values in registers of threads in warp can be only directly swapped between thread registers or copied from one register to another. That way, no shared or global memory is needed for data exchange between threads. This kind of efficient exchange of already loaded data provides an almost instant speedup especially for sorting algorithms, further see[63].

Shuffle instructions allow to swap words in registers within a warp in many possible

permutations, see also [64]. Code 8.7 lists the available shuffle instructions. An illustration

```

1  __shfl();           // direct copy from indexed thread
2  __shfl_up();       // copy from thread with lower idx relative to caller
3  __shfl_down();     // copy from thread with higher idx relative to caller
4  __shfl_xor();      // copy from thread based on bitwise XOR of own lane ID
                        // (butterfly exchange)
    
```

Code 8.7: Thread shuffle instructions.

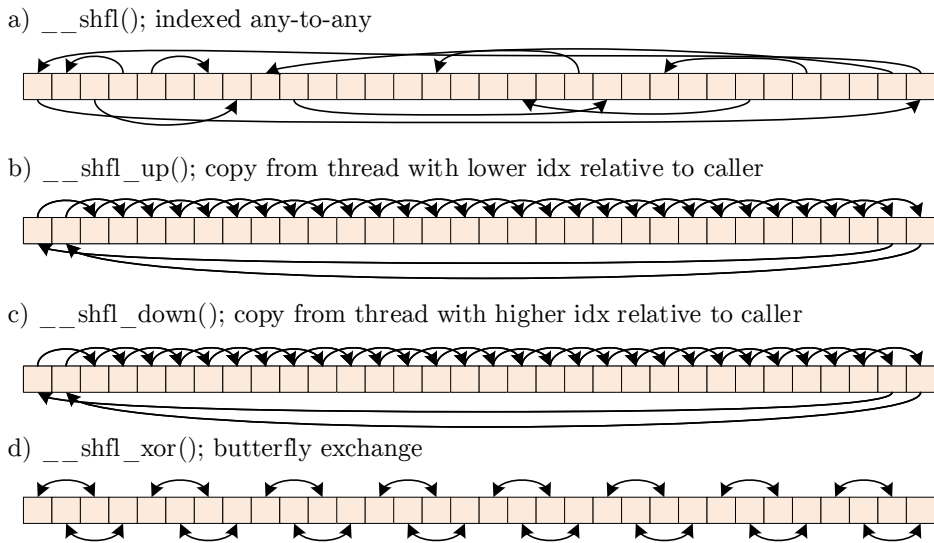


Figure 8.12: Register shuffle methods.

Notes on GPU memory efficiency

The overview of GPU memory usage and utilization as has been presented throughout past pages focuses only on fundamentals of GPU memory hierarchy. A lot more ink has been spilled for an in-depth explanation and pursuit of maximal GPU memory performance, further see [73–78].

As far as the peak bandwidth of each memory type is concerned, only the peak bandwidth of global memory is usually provided by the manufacturer (720 GB/s for Tesla GP100). When operating the data in registers or other on-chip memories, the theoretical peak bandwidth rises steeply as the latency of each memory decreases, see Table 8.3. The effective, truly obtained, bandwidth is however what affects the solution performance:

$$\text{effective bandwidth [GB/s]} = \frac{(\text{bytes read} + \text{bytes written})/10^9}{\text{execution time}} . \tag{8.3}$$

The pursuit of a maximal memory bandwidth is of high interest especially for *bandwidth-limited kernels*. These are usually kernels that do not execute enough arithmetic labor and remain waiting for data from memory transactions. Typically that is the case of various sorting algorithms, matrix transposition kernels, large reductions and similar tasks.

For a closing remark, it should be apparent by now that only a high bandwidth by itself does not necessarily mean high execution performance. Among the bytes loaded in Equation (8.3), there might be a large portion of useless unrequested data loaded due to memory transaction granularity as explained in Section 8.2.3.

Universally desired is therefore a maximal bandwidth paired with a maximal bandwidth efficiency. Any kind of memory performance measurement therefore shall serve only as an auxiliary tool for observing effects of fundamental optimization steps such as coalesced memory accessing and not as an execution performance benchmark. In the case of such a complex platform as CUDA, the judge is always the wall clock time. Well performing kernels with seemingly poor memory bandwidth are not uncommon. Their redeeming property is their ability to hide the memory latency by executing enough arithmetic labor upon the on-chip data.

As soon as the DRAM memory is accessed efficiently enough, the scope of optimization should move to efficient data reuse by maximizing L1/L2 cache hit rate, shared memory allocation and possible register exchange. These optimization steps often require significant code revisions or even changes in the solution algorithm itself.

Table 8.3 complements the GPU memory topic by comparing the capacity and latency of all available memory types. The values are proprietary for the Nvidia Tesla GP100 GPU, the currently most powerful Nvidia GPU.

Memory type	Capacity	Latency [cycles]
Registers	14 336 KB †	1
Shared memory	64 KB per SM	1 to 32
Global memory	16 GB	L1\$ / Tex\$ hit: up to 20
Texture memory		L1\$ miss, L2\$ hit: up to 100
Local memory		L1\$ miss, L2\$ miss: up to 1500
Constant memory	64 KB	Const\$ hit: 1 †† Const\$ miss: up to 1500

† Maximum of 255 32-bit registers per thread.

†† If all threads in the warp read the same address.

Table 8.3: Comparison of size and latency of memory types of Nvidia Tesla GP100.

8.3 CUDA hardware model

Previous sections took apart the abstraction of the CUDA programming model. CUDA SIMT execution model and the CUDA memory model were discussed dominantly from the perspective of a practicing programmer. Only essential remarks were provided about the connection between the programming model and the actual hardware. The content of the upcoming section aims to complete the puzzle by describing the hardware model of CUDA-capable Nvidia GPUs. Much like in previous sections, an excessive focus on a particular GPU generation will be avoided. It was already shown that compute capabilities of each architecture may differ. Similar is the situation in the scope of hardware assembly.

Section 8.1 discussed that the CUDA SIMT execution model merges each 32 logical (scalar) threads into warps (effectively the SIMD width). Warps are then assembled into thread blocks that are to be serviced by multiple Streaming Multiprocessors (SMs) residing on the GPU chip. There is a fundamental analogy between the programming and hardware models that this section aims to convey, see Figure 8.18. Therefore the structure of the following content will be held in consistence with Section 8.1, where appropriate.

8.3.1 Streaming Multiprocessors (SMs)

It has been discussed in the introduction of Chapter 7 that a contemporary GPU core is essentially a large array of thousands of cores that are capable of rather simple arithmetic and logic operations. Such a description is true only from a sufficiently distant perspective. Individuals aiming to exploit the complex architecture that is a GPU device can hardly do so without a deeper understanding of the hardware at a lower level.

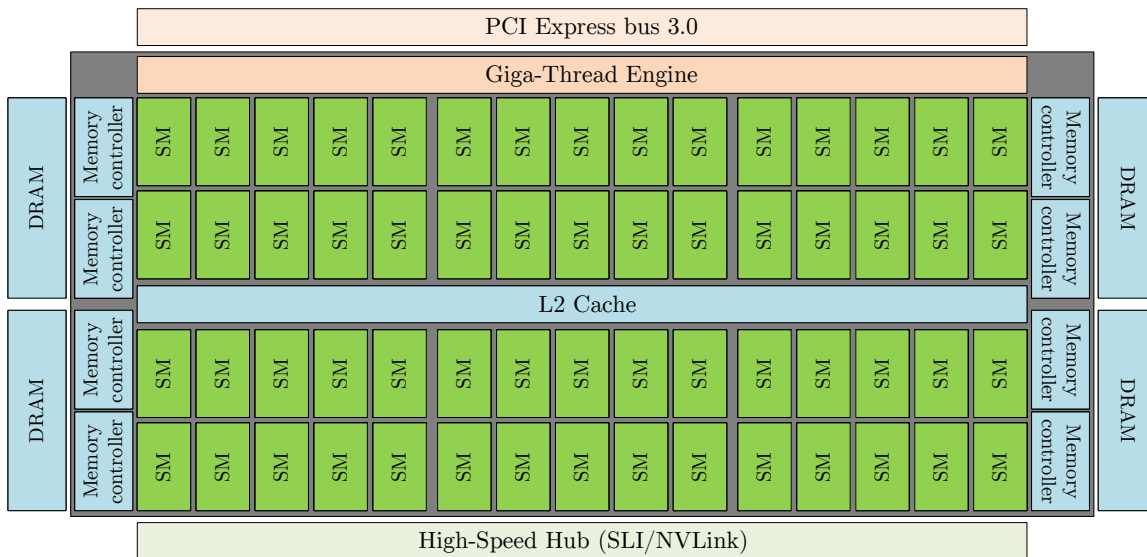


Figure 8.13: Scheme of the Pascal GP100 core.

The fundamental unit of the assembly of each CUDA-capable GPU core is the Streaming Multiprocessor (SM). Commonly, the GPU core contains multiple SMs, see Figure 8.13. By cramming more SMs into the GPU core, the overall performance grows linearly with

the number of SMs. That way, vaguely said, the manufacturer develops an SM of (\approx Compute Capability) and scales the performance of GPU cores for each market segment by incorporating more or fewer SMs into the core.

Figure 8.13 illustrates the assembly of the latest Pascal GP100 core. SMs share the resources of L2 cache and the access to DRAM memory via memory controllers. Each Streaming Multiprocessor is an independent hardware tile that operates its own array of computational resources:

- computational cores,
- memory transaction units,
- registers,
- instructions,
- shared memory,
- L1 cache.

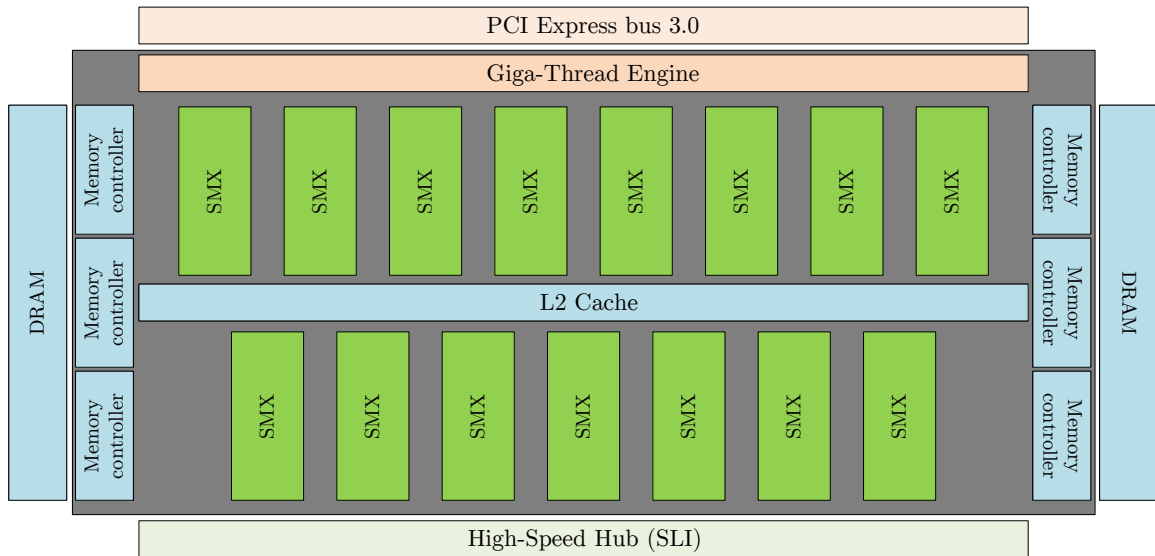


Figure 8.14: Scheme of the Kepler GK110 core.

The hardware content of a Streaming Multiprocessor differs for each GPU architecture and Compute Capability. The direction of development will be demonstrated by comparison of Streaming Multiprocessors in cores GK110 (Kepler) [63] and the latest GP100 (Pascal) [62]. Figure 8.14 provides a scheme of the GK110 core. By comparing of the older GK110 to the state-of-the-art GP100, one can identify the direction of GPU evolution. The GP100 contains more SMs in comparison to GK110. On the other hand, one Pascal SM is also smaller and simpler compared to the Kepler SMX⁸. An illustration of Pascal SM and Kepler SMX is shown in Figures 8.15 and 8.16. Generally, it can be concluded that the development of GPGPU architecture aims for more simple independent Streaming Multiprocessors. That

⁸The Streaming Multiprocessor of the Kepler generation was actually named SMX (Next Generation Streaming Multiprocessor) by the manufacturer. That was due to large differences in comparison to the previous Fermi architecture.

way, the GPU is able to manage its scheduled workflow more precisely and it is easier for it to achieve higher occupancy of computing cores.

Both Pascal and Kepler Streaming Multiprocessors contain similar hardware units. Main hardware differences are in finer scale of SMs compared to SMXs rather than in the actual content of Streaming Multiprocessors⁹. Most significant descriptors of GPU cores are also compared in Table 8.4. The hardware units contained in a typical contemporary Streaming Multiprocessor will be discussed in what follows.

HW Characteristic	Pascal GP100	Kepler GK110
Streaming Multiprocessors	60	15
CUDA Cores total	3840	2880
CUDA Cores per SM	64	192
Shared memory per SM	64 kB	16/32/48 kB †
Registers per SM	65 536 × 32 bit	65 536 × 32 bit

† The Kepler architecture (CC 3.x) allows the programmer to split 96 kB between L1 cache and shared memory, see [63].

Table 8.4: Comparison of hardware properties of Pascal GP100 and Kepler GK100 cores.

⁹The upcoming Volta architecture, however, introduces entirely unique *Tensor Cores* that are capable of fully parallelized matrix operations, further see [79].

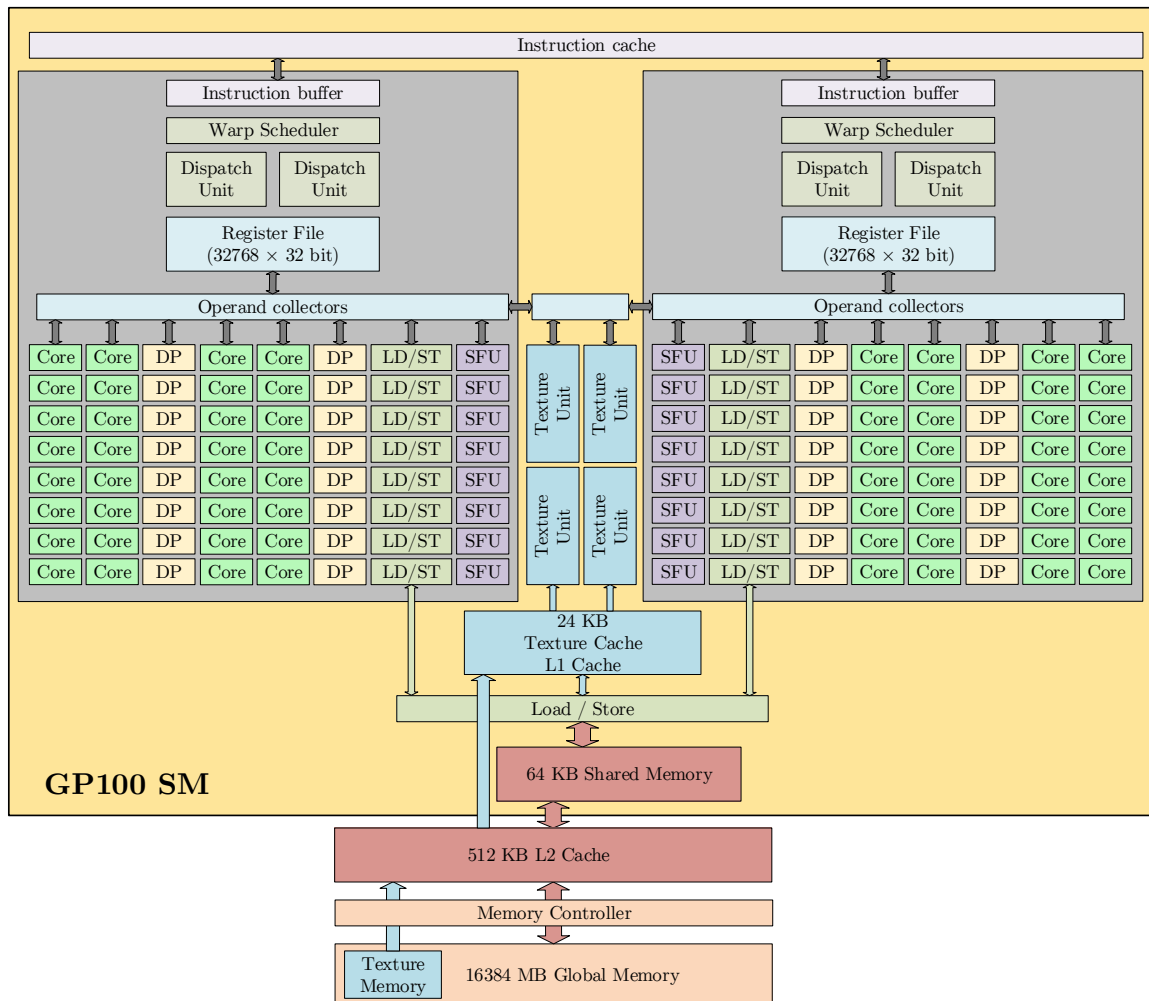


Figure 8.15: A single Streaming Multiprocessor (SM) of the Pascal GP100 core.

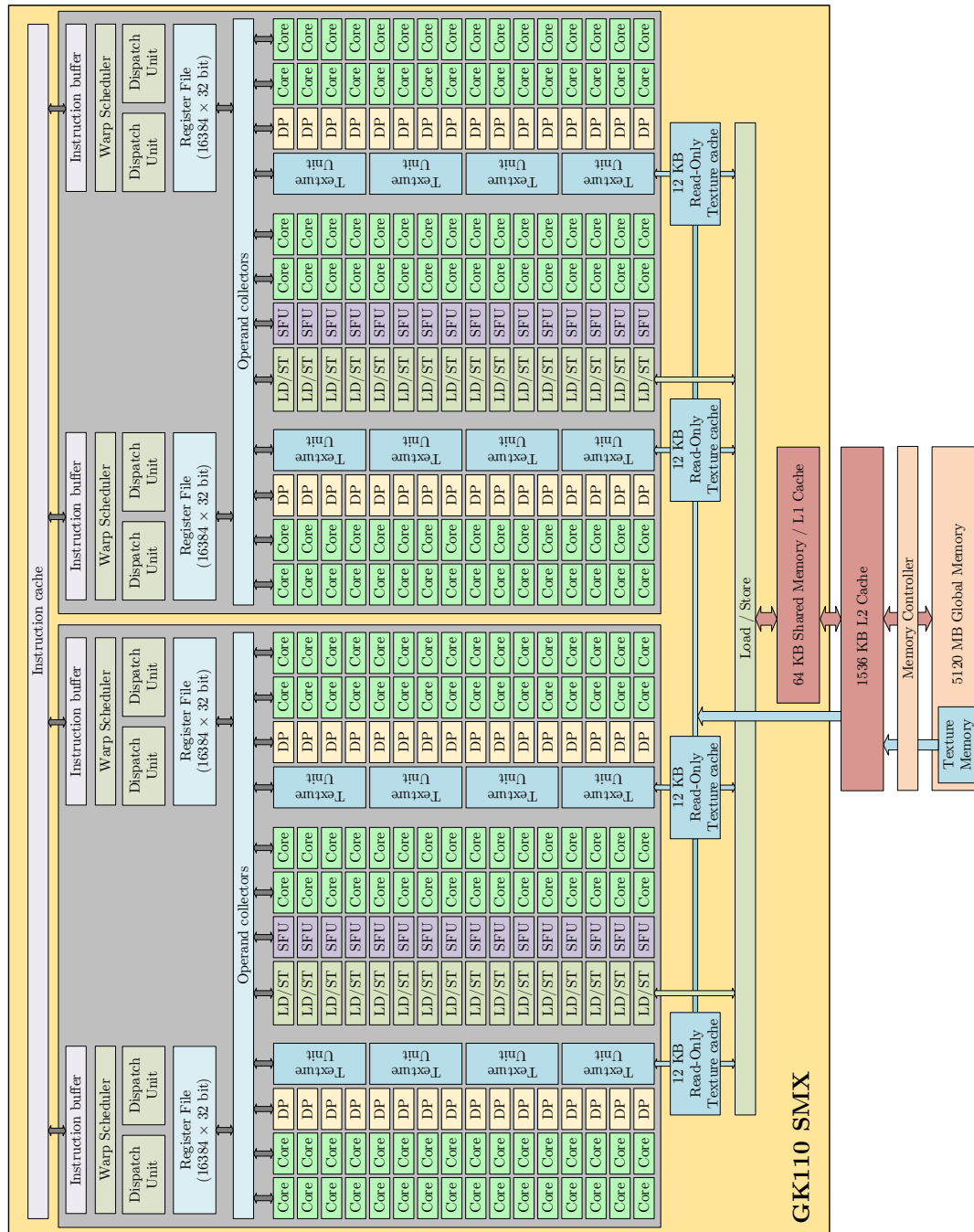


Figure 8.16: A single Streaming Multiprocessor (SMX) of the Kepler GK110 chip.

8.3.2 CUDA Cores

Hundreds (or thousands even) of CUDA Cores, or shortly, *Cores*, are the main workers that execute most of GPGPU labor. The Cores are similar to Arithmetic Logic Units (ALUs) and Floating Point Units (FPUs) of a CPU. A contemporary CUDA Core operates its own Floating Point Unit and Integer Unit (IU), see Figure 8.17. Therefore, it is able to operate upon scalar values of 32-bit floating point or 16/32-bit integer type.

Each CUDA Core contains its own dispatch port through which it receives its respective scheduled instruction from the Dispatch Unit of its Streaming Multiprocessor. The Core also possesses its own operand collector port that serves for receiving operands from the register file. After the FPU or IU of executes an instruction upon the operands, the result is temporarily stored in the result queue of the core, waiting to be sent back to register file.

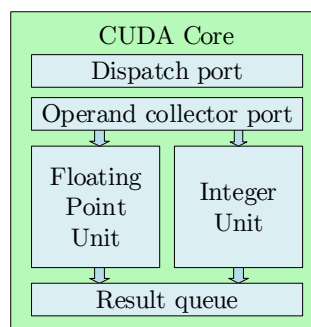


Figure 8.17: Scheme of a single CUDA core.

In practice, a single CUDA Core is a physical counterpart to a single thread of the CUDA programming model. Instructions that are assigned to 32 threads within a warp are then scheduled to be executed simultaneously by 32 Cores. For example, in the instance of Code 8.3, the request for a vector addition is directly serviced by 32 scalar CUDA Cores that simultaneously perform the computation as in Equation (8.2).

8.3.3 Double Precision Units

As has been mentioned, CUDA Cores are capable of conducting arithmetic instructions upon floating point values of single (32 bit) precision. The GPU platform was not designed from scratch for native higher precision arithmetics or even usage in scientific computing. Although, for purposes of this work, computing in single precision has been used almost exclusively due to the available hardware. Graphics applications still require mostly integer and single-precision floating point *algebraic* operations.

Since compute capability 1.3, Nvidia GPUs are capable of executing double-precision (64-bit floating point) arithmetics. The performance of double-precision instructions differs depending on the market segment of each particular GPU. 64-bit FP arithmetics is then conducted either by single-precision FPUs of CUDA Cores or by dedicated double-precision arithmetic units (DPUs).

Gaming GPUs are often not equipped with DPUs (or more precisely - these are disabled) and are forced to use their FPUs for double-precision arithmetics. That way, for example, it takes the FPU 24 (Kepler) or 32 (Maxwell and Pascal) times longer to multiply two 64-bit

FP values in comparison to a multiplication of two 32-bit FP values. This results in single-to-double performance ratio of 1/24 or 1/32. However, such a low performance in double-precision is not a drawback for GPUs designed for graphics. It moreover allows the vendor to offer differently capable GPUs for various market segments with using an identical GPU core. For instance, the same Pascal GP102 core is used in the gaming GTX 1080Ti (retail price of \$700) as well as in professional GPGPUs, the Quadro P6000 (\$5000) and Tesla P40 (\$6500). Apart from several other differences such as the DRAM capacity, and the number of unlocked Streaming Multiprocessors, the major portion of additional costs is paid for the unlocked DPUs per SM.

A GPU that is equipped with unlocked DPUs, exhibits a full double-precision performance proportional to the number of DPUs in the sense that DPUs are optimized to, e.g., multiply two 64-bit floating point values. The total GPU performance in DP arithmetics is dependent on the number of DPUs that are crammed into a GPU core. A fundamental obstacle here is actually the physical size of a DPU. The spatial requirements of a set of transistors for multiplication of two floating point numbers grow with second power of the bit length of each data type. Therefore, designing an arithmetic unit for multiplication of 64-bit values requires roughly four times more transistors as it would for 32-bit values¹⁰. Therefore, the number of DPUs is typically a fraction of the number of CUDA Cores. The most powerful Tesla and Quadro devices come with all their DPUs enabled, that is exactly half as many DPUs as CUDA Cores. Hence, the single-to-double performance ratio of contemporary GPGPUs is 1/2 as these are able to conduct half as many double precision floating point operations per second (DFLOPS).

8.3.4 Special Function Units

Unlike the need for double precision arithmetics that varies for different market segments, all GPUs encounter the computation of approximation of various *transcendental* functions. Transcendental functions are a subset of analytic functions that, unlike algebraic functions, cannot be expressed by a finite sequence of algebraic operations. In graphics and engineering applications, most common examples are approximations of trigonometric functions, hyperbolic functions, logarithms, exponential functions or square root computation.

In practice, an approximation of these functions is computed using first several terms of Taylor series of each function, see [80]. For instance, the sine of an argument can be approximated as follows:

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \quad (8.4)$$

Such a computation surely can be conducted using FPUs and IUs of CUDA Cores. However, this would result in very poor performance because (i) enumerating such a formula in all CUDA Cores of a fully utilized GPU would create a large register pressure for storing intermediate results of summations and divisions and (ii) each CUDA Core would have to perform at least two or three floating point divisions that can take around 300 cycles each, depending on the division algorithm, see [81]. To avoid such performance drops in computation of these frequently used functions, Nvidia GPUs are equipped with Special Function

¹⁰More precisely, it is because the 23-bit mantissa of a 32-bit float and 52-bit mantissa of a 64-bit double.

Units (SFUs). SFUs contain instruction sets designed specially for fast approximation of common transcendental functions in single precision.

8.3.5 Load/Store Units

Load/Store (LD/ST) units, or Load/Store queues, are units that serve the Cores for data transport from and to GPU's memory. If a Core requests a memory load or store transaction, the request is sent into the queue of a LD/ST unit. Until the request is serviced, the Core (and its thread, respectively) is set on hold.

Because the LD/ST units are *pipelined*, one of these can service load/store requests from multiple Cores at a time. In an ideal situation of a coalesced memory reading, i.e., requests from a warp fit into the span of a single memory transaction (see Section 8.2.3), requests of all threads in warp can be serviced by a single LD/ST unit. If accesses in warp are not coalesced, additional memory transactions are invoked by other idle LD/ST units or put in pipelines if these are busy.

Threads typically cannot proceed further in code when waiting for their respective LD/ST requests and are truly stalled. That way, each warp would encounter a significant idle time waiting for memory transactions. Luckily, warp schedulers are able to identify warps that are currently idle and can schedule other warps that are eligible to utilize the Cores of each Streaming Multiprocessor. Understanding the pipelined and independent operation of LD/ST units is a key for understanding how the GPU is able to efficiently service a number of threads that is larger than the number of physical Cores.

8.3.6 Texture Units

Texture units (TUs), or also Texture Mapping units (TMUs), are specialized graphics units contained in each Streaming Multiprocessor. Texture memory itself does offer a couple of features applicable in non-graphics GPGPU computing (e.g. spatial caching and using texture memory bandwidth for additional global loads) as mentioned in Section 8.2.4. The usage of Texture units for GPGPU computing, however, is rather limited. Graphics routines are beyond the scope of this work. Therefore, abilities of Texture units will be addressed in short.

The purpose of Texture units is to efficiently perform *texturing* or *texture mapping*. This is a graphics routine of transforming a 2D raster image (texture) to cover an arbitrary 2D planar or 3D spatial model. Such an operation is conducted by operations such as image resize, rotation, various transformations, interpolation and other. For non-graphics usage, the utilizable ability is an efficient spatial interpolation across data.

8.3.7 Register File

Each Streaming Multiprocessor is also equipped with its own portion of registers. Recall that the properties of registers were thoroughly discussed in Section 8.2.8. It is worth noting that both Kepler GK110 and Pascal GP100 dispose with equal register resources of $65\,536 \times 32$ -bit resources¹¹, compare Figures 8.16 and 8.15. The significant difference of four-years of development is that in the GK110 SMX, the Register File is shared among 192 Cores whereas in the GP100 SM, the same amount of registers is available for 64 Cores.

¹¹One SMX in GK110 operates $4 \times 16\,384 \times 32$ -bit registers while one SM in GP100 is provided with $2 \times 32\,768 \times 32$ -bit registers.

8.3.8 Giga-Thread Engine

The Giga-Thread Engine (GTE), or Giga-Thread Scheduler, is the hardware controller that manages (schedules) the GPU workload across all Streaming Multiprocessors of the GPU. In accordance with everything said so far about CUDA programming and memory models, it is clear now that when a kernel launches a grid of thread blocks to be executed, individual thread blocks need to be assigned to individual Streaming Multiprocessors. This task is up to the Giga-Thread Engine along with providing each thread block with its own `blockIdx` and `blockDim` values based on the information from compiler. The GTE also transmits instructions of each scheduled thread block to Instruction cache of its respective SM.

The number of blocks each Streaming Multiprocessor is able to service depends on how much shared memory and registers each block requests. The most efficient way is, of course, to spread the workload (blocks) evenly across all Streaming Multiprocessors as these operate independently in parallel. For instance, the Pascal GP100 core contains 60 Streaming Multiprocessors, each of which is able to service a block of maximum size of 1024 threads (i.e. 32 warps). This, however, is possible only if there is enough on-chip resources available in each particular SM.

As soon as all SMs are occupied but there still remain unassigned thread blocks, the GTE scheduling waits for a SM that completed its scheduled labor. Therefore, when all SMs available are occupied, the rest of the workload begins to be serialized after each 60 thread blocks. If multiple kernels (grids) are simultaneously executed on the GPU, the GTE ensures an “optimal” distribution of workload of each grid across Streaming Multiprocessors. However, the actual logic behind the operation of Giga-Thread Engine is not published by the manufacturer.

8.3.9 Warp Schedulers

Right after the Giga-Thread Engine schedules thread blocks from the grid to be executed by individual Streaming Multiprocessors, each thread block is from then on maintained by Warp Schedulers of its respective SM. Warp schedulers maintain the division of thread blocks into individual warps, i.e. they assign the value of `threadIdx` to each thread. The enumeration of global thread indexes is then conducted as shown in Code 8.1 and 8.2.

At this point, the warps are *scheduled*, i.e. the logical abstraction of grid and thread blocks is converted to its physical counterpart that is the array of Streaming Multiprocessors. However, the workload of the kernel cannot be executed yet as the resources requested by warp are not assigned.

8.3.10 Dispatch Units

As soon as the Streaming Multiprocessor disposes with available resources requested by a warp (e.g. at least 32 idle Cores and sufficient registers), Dispatch Units of the SM assign `threadIdx` values and instructions of each logical thread to its respective physical counterpart that is a particular CUDA Core, a DPU, an SFU or an LD/ST unit.

Each Streaming Multiprocessor is able to execute only a little number of warp at once, depending on the number of Cores. If there is more warps scheduled than the SM is able to service in parallel, execution of these is then serialized, similarly to a kind of serial execution of thread blocks beyond the number of SMs. However, Dispatch Units of each SM observe

the execution process of each warp and as soon as a warp becomes waiting for a memory load/store, the Dispatch Unit swaps such a stalled warp for another warp that is ready to compute. This way, by such an *asynchronous execution* of warps, the SM ensures that its compute units are as busy as possible. Similarly to the operation of the Giga-Thread Engine, the logic behind this feature is also not published by the manufacturer. If there is enough warps for the SM to choose from, Dispatch Units are able to *hide the latency of memory loads* by executing warps that are ready to run. The issue of *latency hiding* will be addressed in Section 8.4.4.

8.3.11 Instruction cache and Instruction buffer

Each Streaming Multiprocessor operates its own Instruction cache and Instruction buffer. These serve the SM to hold a sufficient amount of instructions for all warps it is scheduled to operate. In practice, the Instruction cache serves as a kind of “L2 Instruction cache” and the Instruction buffer as a kind of “L1 Instruction cache”. The details of behavior of Instruction cache and Instruction buffer are not published by the manufacturer. Their operation and performance can be however estimated by *microbenchmarks*, further see [78].

The aim of the section above was to connect the CUDA programming model with the actual hardware of the device. The analogy between these imaginary and physical layers is once more summarized in Figure 8.18.

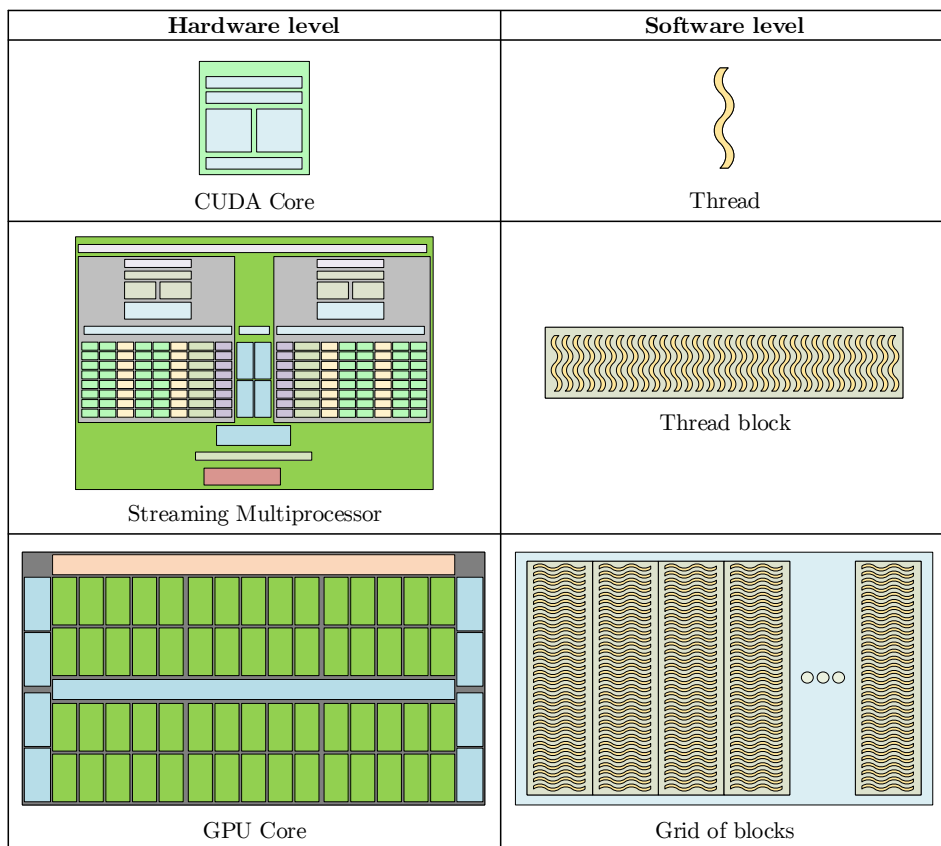


Figure 8.18: Analogy between CUDA programming and hardware models.

8.4 Notes on GPGPU performance

The previous chapter introduced the reader to the complex, multi-layer system that is the architecture of CUDA-capable GPUs. On many occasions, it was discussed that the overall performance of a kernel execution depends on programmer's approach to:

- the CUDA programming model, i.e. the level of achieved parallelism of the solution, to what extent a divergent code is avoided across the SIMD width (warp), etc.,
- the CUDA memory model, i.e. how efficiently the DRAM memory is accessed, what is the ratio of the L1/L2/Tex cache hits and misses, if and how efficiently is the shared memory accessed, how high is the register pressure of the kernel, etc.,
- the CUDA hardware model, i.e. how the imaginary grid of thread blocks distributes the workload across Streaming Multiprocessors of the GPU.

Although there have been advised universal goals for the programmer in pursuit of good performance such as coalesced memory accessing, the overall effect of different programming approaches might be hard to predict beforehand. Moreover, different GPU architectures may execute an identical code with drastically different performance depending on their hardware equipment. That may be with or without a programmer's intervention¹². As should be apparent by now, optimization possibilities depend up to a large extent on the particular device used. Nevertheless, the following content aims to provide the reader with an overall knowledge of how the Nvidia GPGPU architecture executes its workload and how its performance shall be understood.

8.4.1 Terminology of execution performance

First off, let us define the essential terminology of performance analysis. It has been mentioned few times already that the only judge of the execution performance is the *wall clock time*, i.e. the plain difference between time stamps at execution start and end.

The remaining available *performance headroom* of the device as well as the effect of subsequent code changes towards better performance can be determined and observed by evaluating the utilization of possibilities of the particular device. The utilization of the processing horsepower itself can be studied from various perspectives.

Occupancy

One of meaningful descriptors is *occupancy*. Occupancy is related to the number of warps that are concurrently executed by the GPU at any given time. Streaming Multiprocessors of each generation do have a maximal number of warps they can maintain. For example, one Kepler SMX is able to service up to 64 warps. Seemingly no progress was made as 64 warps per SM is the maximum for Pascals¹³. With the knowledge of their respective device, the programmer is able to estimate the occupancy. Occupancy is in literature defined in two similar meanings:

¹²For instance, there was a major solution speedup with the release of the Fermi (CC 2.x) generation. This was due to the introduction of the L2 cache shared for all SMs. Being a non-programmable cache, the L2 cache would not require any programmer's attention to lower the latency of DRAM accesses, where possible.

¹³Note that there are 15 SMXs in the most powerful Kepler GK110 and 60 SMs in the most powerful Pascal GP100.

- occupancy is often understood as the number of warps that are executed at the same time by all SMs:

$$\text{occupancy [warps]} = \text{warps executed in parallel}, \quad (8.5)$$

- possibly more common is the definition as the number of concurrently executed warps in relation to the theoretical maximum of the GPU:

$$\text{occupancy [\%]} = \frac{\text{warps executed in parallel}}{\text{maximum number of warps executed in parallel}} \times 100 \quad (8.6)$$

The programmer can influence the occupancy of their GPU core by adjusting the size (`blockDim`) of thread blocks (i.e. the number of warps in block). Unlike warps within a thread block, execution of thread blocks cannot be interrupted until all warps within the thread blocks are done. In other words, the SM can search for warps that can compute only within currently executed thread block, not across other thread blocks. Therefore, it is desirable to ensure that there is enough (as much as possible) warps to choose from in each thread block. That way, however, shared memory and register requirements for each thread block (i.e. each SM) grow and may result in spilling into local memory on the other side. To predict the occupancy based on the particular Compute Capability and register and shared memory requirements of a kernel, Nvidia provides the programmer with a convenient utility, see [82].

Latency, throughput and concurrency

The occupancy itself, however, does not suffice as a definite description of utilization of the device. For benchmarking purposes, it is desirable to be able to estimate the *processing power* of GPU. *Latency* is one of metrics used for such a performance analysis. Latency is defined as the number of cycles required for execution of a task. This task may be an arithmetic instruction, memory transaction instruction or an execution of a portion of workload. Often, *mean latency* is desirable, i.e. an average latency of multiple tasks.

Further, one can measure another metric known as *throughput*, or also *processing rate*. Throughput is defined as the number of tasks completed within a given time interval. In practice, this interval is often set to a single cycle. The observed task, again, can be a particular instruction or a chunk of workload. In case of a serial execution, throughput directly describes, e.g. the amount of work completed within a given amount of cycles or, more commonly, an average amount of work done per cycle.

The GPU operation differs from a single thread CPU. Multiple independent SMs of a GPU simultaneously execute their respective independent workloads with their own throughputs. To describe the load of an entire GPU core, this has to be taken into account. The metric which describes the amount of tasks done at the same time is *concurrency*, or *parallelism*. In each particular time, the value of concurrency typically differs. Therefore, more convenient information is provided by calculating of the *mean concurrency*. According to *Little's law*, see [83] and [84], mean concurrency can be estimated as follows:

$$\text{mean concurrency [tasks]} = \text{mean latency} \times \text{total throughput}. \quad (8.7)$$

The formula (8.7) assumes that the process studied exhibits a sustained behavior over the time span studied. In other words, that the execution time is infinite. In practical use,

the real scenario is often not too far from these assumptions and Equation (8.7) can be used for a good approximation.

8.4.2 Instruction throughput

It was mentioned that in pursuit of minimization of execution time, the programmer cannot rely strictly on maximizing occupancy, see [85]. Rather than to occupancy, the execution time is closer related to the amount of work (i.e. instructions) being done per time. That means one should rather focus on maximizing the *instruction throughput*, also see [74]:

$$\text{instruction throughput} \left[\frac{\text{instructions}}{\text{cycle}} \right] = \frac{\text{instruction concurrency}}{\text{instruction latency}}. \quad (8.8)$$

Now, let us assume a Kepler GK110 SMX that contains 192 CUDA cores. Therefore, it can execute up to 6 warps concurrently, i.e. at most 6 SIMD instructions per cycle. This value is called *peak throughput*. The Pascal GP100 SM contains 64 cores, therefore its peak throughput is 2 instructions per cycle. To reach peak throughput, the SM needs to be given enough instructions (warps) to execute. Consider a single precision MUL (multiply) instruction that can be completed with latency of 6 cycles, also see [81]. In such a case, the GK110 SMX requires 36 warps to reach its peak instruction throughput. On the other hand, one GP100 SM is fully utilized with only 12 warps. This simple example demonstrates the desirable benefit of more independent and smaller SMs. Moreover, note that both discussed Streaming Multiprocessors dispose with equal register resources. Therefore the GP100 SM is able to accommodate warps with higher register demands and still reach solid utilization of its peak instruction throughput. For GK110 SMX, there is a higher possibility that its resources will not suffice to maintain the number of warps required to reach its peak instruction throughput.

The described approach can be used for approximating the number of warps per block to reach full throughput based on knowledge of latency of given arithmetic instruction. In case of memory transactions, their latency is not a known constant for it depends on shared memory bank conflicts, cache hits and misses and memory coalescing. A rough approximation, however, can be estimated. Let us assume a fully coalesced global memory access that misses both caches. Such a memory transaction exhibits latency from around 350 to 400 cycles (such a value has to be estimated via microbenchmarking, see e.g. [78]). Next, one needs to set the maximum global memory instruction throughput (L1 and L2 cached). This can be estimated by using global memory bandwidth (208 GB/s for Kepler, 720 GB/s for Pascal), clock rate (705 MHz for GK110, 1 328 Mhz for GP100), and number of SMs (15 in GK100, 60 in GP100). Now, the peak throughput of memory transactions can be estimated as follows:

$$\text{mem. throughput} \left[\frac{\text{mem. trans.}}{\text{cycle}} \right] = \frac{\text{peak bandwidth [GB/s]}}{\text{clock rate [GHz]} \times \text{SM count} \times \text{bytes in trans.}}. \quad (8.9)$$

This gives peak throughput of 0.154 memory transactions per cycle for Kepler GK110 and 0.071 memory transactions per cycle for Pascal GP100. To reach this peak throughput, each SMX in GK110 needs at least 51 such instructions¹⁴ (warps) and each SM in GP100 requires

¹⁴Note that the maximum number of warps per SM/SMX is 32 warps (1 024 threads).

at least 24 instructions (warps) per block. It is noteworthy that for estimation of the peak memory throughput, Little’s law does not hold because the peak memory bandwidth is not a sustainable value. At best, a kernel can sustain 20% of memory bandwidth, typically around 5% as argued in [74].

8.4.3 Instruction level parallelism

So far, it might seem that the value of attained instruction throughput is related to the number of concurrently executed warps (i.e. occupancy) so closely that one might rely on occupancy as the only performance metric. Such a statement holds only in case when all warps issue only a single instruction at a time. There is, however, a possibility for a warp to issue multiple instructions in parallel, i.e. to exploit *instruction level parallelism*, or ILP.

The SM allows the warp to issue multiple instructions at the same time under the condition that none of their output values is meant as input for any of the rest of issued instructions. An example of such ILP-capable code might be as follows:

```

1 // example - independent vector additions
2 a[thrdIdx] += b[thrdIdx]; //I#0
3 c[thrdIdx] += d[thrdIdx]; //I#1
4 e[thrdIdx] += f[thrdIdx]; //I#2
5 g[thrdIdx] += h[thrdIdx]; //I#3

```

Code 8.8: A possibility for instruction level parallelism.

That way, if the warp already disposes with all data needed, it puts these independent instructions in *pipeline* of the SM. Depending on the hardware (pipelining is not exclusive for GPUs), the pipeline is equipped with a certain number of *stages* that can service different instructions. Consider a simple arithmetic instruction that can be divided into four stages:

1. decoding stage, i.e. choosing instruction,
2. read input from registers,
3. executing instruction,
4. storing output to registers.

Suppose the hardware (CUDA Core) is provided with a 4-stage pipeline. Each pipeline stage is able of executing only a single instruction stage. The trick behind pipelined execution is that the execution stages operate independently. Therefore, for example, after the decode stage is done servicing the first instruction (I#0), the I#0 continues to the next stage and the decode stage, instead of remaining idle, starts decoding I#1 and so on, explore Figure 8.19. That way, an n -stage pipeline is able to service n instructions in parallel (if sustained) with a single-stage overlap.

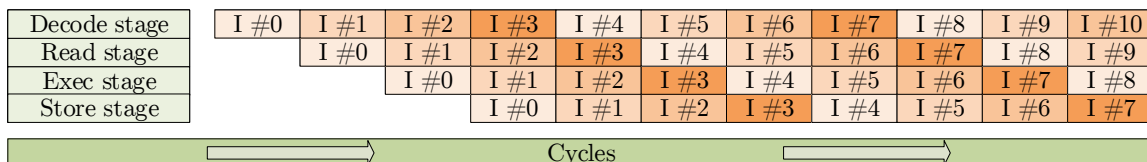


Figure 8.19: Pipelined ILP execution.

Considering such a sustained level of ILP in the previous example of arithmetic instruction throughput, each warp is now able to execute four arithmetic instructions at a time. Thanks to such a pipelined execution, the possible instruction throughput grows four times. In other words, attaining the original throughput would require fewer warps, further see [85]. The possibility of exploiting ILP, however, depends (i) on the nature of each particular problem, (ii) on the programmer's knowledge about such capabilities of their particular device, and (iii) on the programmer's ability to assemble such a code. Finally, in the code, the programmer also needs to ensure that all required inputs are available in registers to allow the hardware to put instructions in pipeline.

Another issue with ILP is that to put an instruction into the pipeline, the device needs to be provided with a fully defined instruction, i.e. the arithmetic instruction must be known as well as its operands must be exactly defined. For example, consider the best possible implementation of computation of mutual distance between particles i and j in bordered three-dimensional space (to omit the conditional code):

```

1  float absoluteDist = 0.0f;
2  float partialDists [3];
3  // computation of squared absolute distance between particles i and j, ILP
4  partialDists [0] = gpu_coordinates [j,0] - gpu_coordinates [i,0];
5  partialDists [1] = gpu_coordinates [j,1] - gpu_coordinates [i,1];
6  partialDists [2] = gpu_coordinates [j,2] - gpu_coordinates [i,2];
7  // squares, ILP
8  partialDists [0] *= partialDists [0];
9  partialDists [1] *= partialDists [1];
10 partialDists [2] *= partialDists [2];
11 // summing squared projections, NO ILP
12 absoluteDist += partialDists [0];
13 absoluteDist += partialDists [1];
14 absoluteDist += partialDists [2];

```

Code 8.9: Computation of mutual distance with using ILP.

In this case, the three instructions for computing three mutual distances (projections) in three dimensions are fetched to the compiler right at the compile time. Therefore the device will put these instructions into pipeline, if possible. After computing these projections of mutual distance, it is required to compute their squares. Again, these instructions do not depend on each other so they will be put in pipeline together. The final summation of squared projections contains three mutually dependent additions that need to be conducted sequentially and cannot use ILP. It is worth noting that the compiler is able to *unroll* loops of known size at the compile time. That way, the compiler generates the exactly same result as in Code 8.9 if given the code rewritten as follows:

```

1  float absoluteDist = 0.0f;
2  float partialDists [3];
3  // computation of squared absolute distance between particles i and j, ILP
4  for (int v = 0; v < 3; v++){
5      partialDists [v] = gpu_coordinates [j,v] - gpu_coordinates [i,v];
6  }
7  // squares, ILP
8  for (int v = 0; v < 3; v++){
9      partialDists [0] *= partialDists [0];
10 }
11 // summing squared projections, no ILP

```

```

12 for (int v = 0; v < 3; v++){
13     absoluteDist += partialDists[v];
14 }

```

Code 8.10: Computation of mutual distance with using ILP and loop unrolling.

As elegant and simple reaching ILP might seem, let us close the subject with an example of such an implementation for an arbitrary dimension as was required in this work:

```

1  (...)
2  float absoluteDist = 0.0f;
3  float partialDists[nvar];
4  // computation of squared absolute distance between particles i and j
5  for (int v = 0; v < nvar; v++){
6      partialDists[v] = gpu_coordinates[j,v] - gpu_coordinates[i,v];
7      //omitting periodic boundary conditions
8      //if      ( abs(partialDists[v] - 1.0f) < 1.0f ) partialDists[v] -= 1.0f;
9      //else if ( abs(partialDists[v] + 1.0f) < 1.0f ) partialDists[v] += 1.0f;
10     //summing squares of projections
11     absoluteDist += partialDists[v] * partialDists[v];
12 }
13 (...)

```

Code 8.11: Computation of mutual distance in an arbitrary dimension.

In Code 8.11, the dimension of the design space is not known until each specific particle system is defined by the user. That way, the compiler cannot unroll the loop of an unknown size N_{var} . Even during the execution, the driver is unable to predict the number and mutual dependencies of the instructions within the kernel as the indices of elements within arrays are not evaluated. Hence, the device cannot utilize ILP. Therefore, reaching a high or even maximal instruction throughput requires increasingly more warps as the dimension of the system grows, see also in the following section. This is an example of a rather pitiful scenario where the achieved instruction throughput tends to decrease as the actual amount of arithmetic instructions increases. A remedy of such an obstacle is to write a specific dedicated kernel such as Codes 8.9 or 8.10 for any possible (ideally integer) dimension of design space, N_{var} .

8.4.4 Bounds on execution performance

Throughout the previous content, the concepts of instruction latency, throughput and concurrency were introduced along with their mutual dependencies. The overall GPU performance, however does not depend quite directly on throughput of individual instructions. More accurate is to observe the throughput of the entire workload that is scheduled. Considering that the workload is divided across a grid of thread blocks of warps, one can pick the *warp throughput* as such a metric.

Despite that all scheduled warps contain an identical workload to be done (keeping aside divergent code), their individual latencies typically differ slightly due to fluctuating latency of individual memory accesses¹⁵. The occupancy therefore also varies over time (although almost negligibly). Each SM can only switch between stalled and idle warps within the

¹⁵The arithmetic performance of individual SMs is considered to be equal.

current thread block. The next scheduled thread block is fetched to the SM after all warps in the previous block are finished.

Building on the already said, the relationship of average warp characteristics are as follows:

$$\text{mean occupancy [warps]} = \text{mean warp latency} \times \text{warp throughput}. \quad (8.10)$$

The expression (8.10) suggests that the warp throughput should grow linearly with rising number of warps scheduled, i.e. with the size of workload. To certain extent, such behavior is close to reality. As long as there are still GPU resources available (registers, shared memory, SMs), the GPU exhibits growing warp throughput (\approx execution performance) close to the following relation:

$$\text{warp throughput} = \frac{\text{mean occupancy}}{\text{mean warp latency}}, \quad (8.11)$$

where mean warp latency is expected not to depend on occupancy. This corresponds with reality only for relatively low number of warps. At that point, the device is ready to execute a warp immediately as it is ready to compute. Such a linear performance growth with the number of warps is governed by mean warp latency. Hence, it is recognized as the *latency bound* of warp throughput [74].

As the number of warps grows, warp schedulers start to operate an array of warps, multiple of which are eligible to be executed. The warp scheduler then selects one warp to be executed and the other warps remain stalled. Therefore, the initially linear growth of warp throughput according to Equation (8.11) starts to decrease and the real warp throughput starts to converge to its theoretical maximum known as *throughput bound*, see the scheme in Figure 8.20.

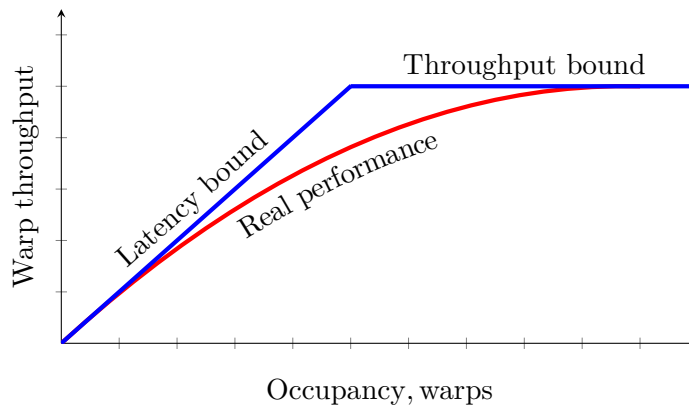


Figure 8.20: Bounds on warp throughput.

In practice, the warp latency and throughput bounds depend on mutually interacting properties of each particular device (compute capability) and each particular kernel, see e.g. [86]. The mean warp latency is closely related to latencies of arithmetic and memory instructions within the kernel. It has been also shown that ILP may strongly influence the instruction throughput within each warp and therefore the its latency.

Furthermore, it is vital to mind the arrangement of instructions in the code, i.e. the *instruction mix*. Similarly to performance drops due to any kind of branching code, the GPU also poorly services kernels with instructions of multiple types. For example, if the kernel alternates arithmetic and memory instructions, to issue the next arithmetic instruction, the scheduler is forced to wait until the previous memory instruction is finished. Hence, for attaining the unharmed instruction throughput would require twice as much warps. There might be even more incompatible instructions in the same kernel, e.g. when alternating between CUDA Cores, DP Units, SFUs, accessing global memory and shared memory banks. Moreover, such a kind of “untidy” code typically paralyzes the ability of the SM to utilize pipelines for ILP.

Latency hiding

From the above said, it is apparent that for reaching maximal solution performance, the goal is to obtain maximum warp throughput possible. Efforts to attain the maximum warp throughput (i.e. to reach the throughput bound) are also often understood as efforts to leave the latency bound. At that point, the device seemingly behaves as if the warp latency was zero, i.e. the *latency is hidden*. A change in number of warps does not affect warp throughput anymore. Vaguely said, the device is then not only highly *occupied* but also did reach its peak *workload throughput* (\approx maximum execution performance).

On the other hand, it is not uncommon that a kernel has such a large warp latency that cannot be hidden due to the limit of 32 warps per block. Reaching the throughput bound for such a demanding kernel would therefore require optimization efforts in pursuit to lower the warp latency by increasing instruction throughput within a warp, i.e. increase the slope of the latency bound.

Chapter 9

Parallel implementation of hyper-dimensional N-body system

The following chapter aims to describe the implementation of a massively parallel solution of the formerly discussed dynamical particle model. During the recent decade, researchers dealing with simulation of particle systems acquired a rather powerful computing platform with the development of general purpose computing on graphic processor units (GPGPU). With the great computational power offered by the GPGPU architecture, increasing number of formerly non-calculable problems are now being solved. Particle systems are nowadays simulated in numerous engineering and research fields. Molecular dynamics, material and mechanical engineering or astrophysics are only a few examples of these.

Namely the astrophysics simulations of vast scenarios of forming galaxies with tens of thousands of planets-particles are now possible to compute, see e.g. [87–89]. Unlike the systems of mutually attracting celestial bodies simulated by the astrophysicists, there exist similar particle models without a direct physical analogy to the purpose of their simulation. This is also the instance of the dynamical particle system as considered further in the work.

One of main tasks of this work is to implement an efficient solution of the proposed system of *mutually repelling* particles that will serve as an optimization tool for obtaining *uniformly* distributed point samples. Such optimized samples may find utilization in dozens of research problems, an interesting instance of which is the statistical sampling for numerical integration of an arbitrary function – Monte Carlo sampling.

Numerical integration of the Monte-Carlo type requires sampling of points that are uniformly distributed within a *design domain*. In this work, the design domain is considered to be the domain of sampling probabilities (values of the joint distribution function – the domain of copulas) which is a unit hypercube $[0, 1]^{N_{\text{var}}}$, where N_{var} is the dimension of the design domain and also the number of random variables of the integrated function.

A brief review of particle simulation algorithms is presented in Section 9.1 and the particle system of interest is set into the context of today’s GPGPU particle implementations and solution algorithms. Section 9.2 follows with an analysis of requirements on the solution implementation.

Hardware limitations rising from the unrestrained dimensionality of the particle system at hand are discussed in detail and reasons for both implementation approaches are justi-

fied, see Section 9.3. There, two different ways of data storage and the associated algorithms are presented. Finally, the performance of both parallel implementations is provided and further compared to the single-thread CPU implementation in Section 9.4.

9.1 Particle simulation algorithms

The first simulation of a dynamical particle system was performed in 1960 by von Hoerner [90]. Since then, particle simulations in various research fields were one of great driving forces of development of super and parallel computing.

The $\mathcal{O}(N^2)$ complexity of the brute-force all-pairs solution (here $N \equiv N_{\text{sim}}$ is the number of particles) has been lowered to $\mathcal{O}(N \log N)$ by the Barnes-Hut Treecode algorithm [91] which lowers the complexity by clustering the distant particles into larger groups and approximates their influence on the solved particle. The performance of such an approximation algorithm is however suited dominantly for particle systems where the distribution of particles is highly non-uniform.

Further reduction to the complexity of $\mathcal{O}(N)$ was achieved by the Fast Multipole Method (FMM) [92]. The FMM-type algorithm assembles hierarchical structures not only by clustering the remote particles but also the nearby particles using local expansions. For large particle systems, usage of the FMM algorithm also greatly reduces the number of summands and thus reduces the resulting computation error. In case of smaller particle systems with thousands of particles, the FMM can be shown to yield an intermediate performance between the all-pairs $\mathcal{O}(N^2)$ and the $\mathcal{O}(N)$ complexity, see [93].

These are, nevertheless, approximation-based algorithms which do not suffice for general research of particle systems such as the one proposed above. The particle system of interest here is being a subject of a constant investigation of influence of various interaction laws on the resulting distribution of particles. This means that any premature approximation of particle interaction might deliver confusing results. Not to mention physical reasoning of clustering particles within a periodically repeated domain. Therefore we consider the direct N_{sim} -body integration that involves the computation of all $\binom{N_{\text{sim}}}{2}$ forces between all pairs of particles.

The direct all-pairs summation is also beneficial for further research of the particle system at hand as we wish to study the evolution of distribution of the potential energy among all mutual interactions. Also, we aim to compute all mutual distances as their distribution is of our interest.

9.1.1 GPGPU computing of particle systems

Since the very first release of the Nvidia's Compute Unified Device Architecture (CUDA) in 2007, see [94], simulations of particle systems were among the first applications to exploit the novelty architecture of general purpose parallel computing.

One of the first implementations of GPU solution of a 3D particle system was the so-called Chamomile algorithm [95]. As used in the CUNBODY library [95, 96], the Chamomile algorithm considers parallelization of the *source* particles (particles acting on the particles being solved) between thread blocks, requiring a rather large final reduction. Latency of such a reduction might be a challenge to hide.

A more efficient approach was later published by the Nvidia itself [97], proposing parallelization of the *target* particles (particles being solved) among thread blocks. This implementation therefore does not require large reductions to be performed. Extensive utilization of registers along with *loop unrolling* is also proposed in pursuit of maximal performance.

The idea [97] of using the on-chip shared memory for circulation of all *source* particles while keeping descriptions of the *target* particles stored in registers/L1 cache is also used in the first of the two implementations presented in what follows. Performance of such a solution algorithm depends dominantly on the arithmetic performance of the GPU used and, consequently, on the ability to hide this arithmetic latency.

Note that term *latency hiding* as used further in the paper is meant as an effort to reach hardware's maximum throughput, see [74] and also recall Chapter 7 of this work. Typically, the more limiting factor is the latency of accessing the (global) memory rather than the latency of execution of arithmetics. However, for each particular code, their actual ratio may vary.

9.2 Requirements on solution implementation

Numerical simulation of the proposed particle system consists of several sub-steps which can be, up to certain degree, executed in parallel. This degree is limited dominantly by the nature of the problem at hand and by the possibilities of the hardware used. Currently, the Kepler architecture Nvidia Tesla K20c (GK110) and the Nvidia GTX 1080TI driven by Pascal (GP102) are being utilized.

A common problem of any parallel implementation is to control writing requests of threads in a way that multiple requests for writing into the same memory address cannot be executed simultaneously. This scenario is known as the *race condition*. When writing, threads can overwrite values computed and stored by other threads in an incorrect order, which typically renders following computations over such data incorrect. Handling such inconsistency in writing is of high importance in implementations requiring large reductions. Commonly, manual serialization of writing requests in code, parallel reductions or atomic operations are used, see [98] and [73]. Possible ways of solving concurrent writing requests are in-detail discussed and their performance compared by the author in [99].

The result of a code exhibiting a race condition cannot be predicted as the performance of threads is expected to be identical and the execution time of the same instruction by identically powerful threads depends on hardly predictable circumstances.

Furthermore, in case of the problem at hand, one has to bear in mind that unlike most of the conventional particle simulations, the complexity of the proposed system is not limited by the number of dimensions, N_{var} . As a matter of fact, the unknown number of particles, N_{sim} , and dimensions, N_{var} , at the time of compilation call for a quite general implementation.

With the theoretical size of the solved problem being arbitrary, the fast on-chip resources of the GPU (registers, L1 cache, shared memory) might not suffice. Therefore the second proposed implementation, see [99] and Section 9.3.2, ignores the possibility of using on-chip resources other than registers needed for conducting arithmetic operations and storing intermediate results. Performance of such an algorithm depends dominantly on the global memory bandwidth and bus size of the GPU used and, consequently, on the ability to hide

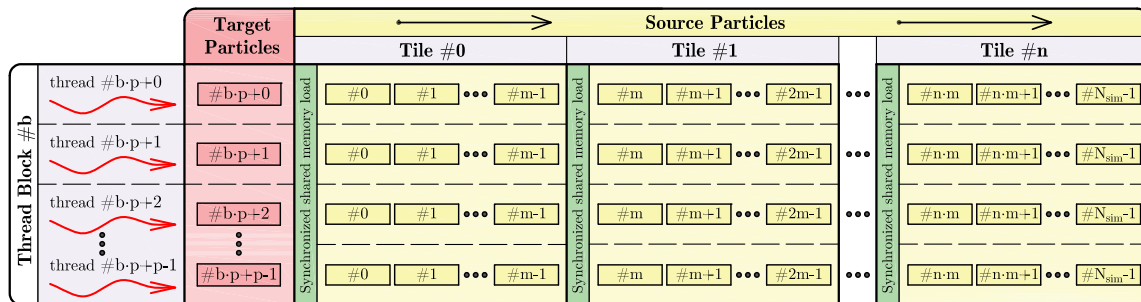


Figure 9.1: Solution procedure of the on-chip memory implementation.

this memory access latency.

9.3 Presented approaches to GPU solution

As has been indicated in what preceded, due to the essentially arbitrary extent (number of particles N_{sim} and dimensions N_{var}) of the particle system at hand, we further present two fundamentally different implementation approaches.

9.3.1 Implementation using on-chip resources

The first presented implementation of particle system simulation is inspired by the concept [97] that proposes to keep the descriptions of *target* particles statically loaded in registers/L1 cache and circulate the descriptions of *source* particles in the shared memory. In our case, however, the term *descriptions of particles* differs for *target* and *source* particles.

The *target* particles (particles, accelerations of which are to be computed) do not possess only unknown values of accelerations but we also wish to compute the radial stress each particle is experiencing and the amount of potential energy stored in interactions of each particle at any given time. This means that during solution, we need to keep stored the following descriptions of *target* particles on chip:

- N_{var} own coordinates of each particle,
- N_{var} unknown accelerations of each particle,
- one unknown radial stress of each particle,
- one unknown value of potential energy belonging to interactions of each particle.

It also appeared beneficial to store on chip the computed projections $\bar{\Delta}_{ij,v}$ of the mutual distance, \bar{L}_{ij} , between the currently solved pair of particles. This means storing additional N_{var} values on the chip. All this data, exclusive to each thread in the thread block, may be stored either in registers or the L1 cache/shared memory.

We have, nevertheless, serious objections towards using registers in this case:

- the number of dimensions, N_{var} , is not known at the compilation time, which might easily lead to uncontrolled register spilling into GPU's local memory,
- GPU's registers are not indexable (at least not conveniently), meaning they cannot handle indexed arrays and these are spilled into the local memory right away,
- for an unknown dimension, N_{var} , it is not even possible to arrange the data into vector structures stored in registers,

- in case of further optimization of data reuse, registers are not accessible by other threads in the thread block. However, it is worth noting that since the Kepler architecture (Compute capability 3.0+), the Nvidia GPUs are capable of fast exchange of data in registers of threads within the same warp using the `__shfl()` methods, recall Section 8.2.8.

Because of the above stated, we propose using the shared memory/L1 cache resources for storing descriptions of the *target* particles.

The descriptions of the *source* particles consist of their coordinates only, i.e. N_{var} coordinates of each particle. We much prefer to store these coordinates in the shared memory as they are to be accessed by all threads in each thread block.

It can be expected that at one point of the solution process, the loaded part of the *source* particles will represent coordinates of the *target* particles, which are already being loaded on-chip. In this case, it is beneficiary to store N_{var} coordinates of the *target* particles in the shared memory as well for they can be accessed right away, without requesting global memory accesses.

Obviously, the amount of the L1 cache/shared memory requested also depends on the number of threads per block which can be tuned up to certain degree. Nevertheless, it should be mentioned that eventually, for a large dimension, N_{var} , the described implementation will start spilling this data into local memory.

Moreover, the precision of arithmetics matters; empirically, we observe that the *float* precision (32 bit values) is sufficient for solution of the system at hand (not requiring a solution of a system of mutually dependent equations of motion). This means requesting half the resources otherwise needed for the *double* precision (64 bit values).

Before closing the subject of the on-chip resources, it should be noted, that each of 13 streaming multiprocessors (SM) of the Kepler Tesla K20c (GK110) possesses 64KB of configurable on-chip memory to be divided between L1 cache and shared memory. The GTX 1080TI (GP102) and all the GPUs of the Pascal architecture follow the concept of the previous Maxwell architecture, providing dedicated 96 KB of shared memory for each of its 28 SMs. The resources of L1 cache have been merged with the texture cache for Maxwells and Pascals. The L1 cache has been provided only with dedicated 28 KB on GP100 and GP102 cores.

Relying on the non-programmable caches is not convenient especially for larger systems. Therefore we lean towards exclusive usage of the shared memory. To conclude on the on-chip memory allocation approach: we store all $[(4 \cdot N_{\text{var}} + 2) \cdot \text{blockDim}]$ values in the shared memory of each SM. The next section discusses the on-chip solution algorithm.

Solution algorithm

As has been already discussed above, the proposed on-chip algorithm distinguishes the *target* particles (particles, unknown properties of which are to be computed) and the *source* particles (particles acting on the *target* particles).

Descriptions of the *target* particles are being held entirely in the shared memory, as justified in the previous section. The *target* particles are maintained by N_{sim} threads divided into $tb = (N_{\text{sim}}/p + 1)$ thread blocks, where $p = \text{blockDim}$. After loading p *target* particles into the shared memory, tiles of *source* particles are consecutively loaded into shared memory and their interaction with *target* particles is solved.

The *source* particles are divided into n tiles, each containing descriptions of m *source* particles. In our implementation, we prefer $m = p$ which leads to possibility of data reuse and also empirically seems optimal for hiding arithmetic latency when good occupancy of GPU is reached. Nevertheless, it might be beneficiary to set $m < p$ to save the on-chip memory in case of high dimension, N_{var} , or when using values of higher precision. A setting of $m > p$ leads to longer arithmetic occupancy of thread blocks which might be useful to hide the latency of global memory accesses if these are too expensive.

The described solution procedure as conducted by a generic thread block is illustratively depicted in Figure 9.1 and also further explained by Algorithm 1. As can be seen, the part of solution of interaction with individual tiles is serial for each thread (each particle). This might seem inefficient at first, but keeping the executed thread blocks busy with arithmetics helps to hide the latency of blocks waiting for global memory accesses.

The newly gained accelerations are stored into global memory and afterwards used for numerical integration using the semi-implicit Euler method, see Equation (6.8). Numerical integration kernel is executed by $N_{\text{var}} \cdot N_{\text{sim}}$ threads, each of which updates its own velocity $\dot{x}_{i,v}$ and coordinate $x_{i,v}$. The task of numerical integration therefore takes only a negligible fraction of the total execution time.

9.3.2 Implementation using global memory

The second presented implementation approach, as already briefly discussed, aims to rely completely on usage of the GPU's global memory. The main reason is that the *theoretical extent* of the particle system at hand is not limited in terms of the number of particles, N_{sim} , nor in the number of dimensions, N_{var} . Therefore the limited amount of on-chip resources is not going to suffice in scenarios of very high dimensions, N_{var} , and more so if the *double* precision format is used.

The global memory implementation as further described, exhibits a great global memory traffic, hiding the latency of which is a major task. The approach here is to unfold the parallelism of the solution entirely into global memory, executing the highest number of threads possible for each computational step. This way keeps the SMs as busy with simple arithmetics as possible. Of course, it is crucial to ensure that the global memory accesses are *coalesced* and bus utilization is maximized.

Such an implementation requires rather large reductions into the same global memory address. For handling these reductions, fast atomic additions `atomicAdd()` are used, see [73].

Despite the fact that there exist $\binom{N_{\text{sim}}}{2}$ mutual interactions (pairs) of N_{sim} particles in each dimension, it is necessary to realize that each interaction between any two particles i and j results in computing and writing two opposite accelerations, one for the particle i and one for the particle j . This in fact means that it is needed to compute and write $(N_{\text{sim}})^2$ (N_{sim} of which are zero) accelerations of particles in each dimension.

It is indeed possible to compute only $\binom{N_{\text{sim}}}{2}$ of the unique accelerations. One can attempt to use the symmetry of repulsive forces and *mirror* these known accelerations for the other particles in all respective pairs. One has to be aware, however, that such mirroring cannot be conducted between thread blocks on the chip and global memory has to be used for data exchange. In fact, the action of writing and reading in GPU's global memory is much more time expensive than on-chip parallel computing. Therefore it might be beneficiary to compute seemingly redundant data instead of further increasing the global memory workload

```

Input:
1 *gpuGlobMemAccels, *gpuGlobMemCoords,
  *gpuGlobMemStresses, *gpuGlobMemEnergies
  /* Allocate arrays in shared memory */
Shared memory allocation :
2 *ShMemTargetCoords, *ShMemSourceCoords,
  *ShMemTargetAccels, *ShMemPartialDistances, *ShMemTargetEnergies,
  *ShMemTargetStresses
  /* Loading static target coordinates */
3 forall respective threads in block do
  | load  $N_{\text{var}}$  coordinates of target particle from *gpuGlobMemCoords
  | into *ShMemTargetCoords
  end
  /* Interaction of source tiles with the target particles */
4 for tile → tilesCount do
  | /* loading source coordinates
5   | if source coords = target coords then
6   | | forall respective threads in block do
7   | | | temporarily switch the pointer *ShMemSourceCoords to *ShMemTargetCoords
8   | | | end
9   | | else
10  | | | forall respective threads in block do
11  | | | | load  $N_{\text{var}}$  coordinates from *gpuGlobMemCoords into *ShMemSourceCoords
12  | | | | end
13  | | | end
14  | | /* computing tile interaction
15  | | forall respective threads in block do
16  | | | forall particles in tile do
17  | | | | compute contributions of:  $N_{\text{var}}$  repelling forces, radial stress, potential energy
18  | | | | end
19  | | | end
  | end
end

```

Algorithm 1: Tile interaction in shared memory.

without providing any other arithmetic tasks to hide this memory latency.

The global memory implementation is divided into several sub-steps (kernel functions):

- computation of $N_{\text{var}} \cdot (N_{\text{sim}})^2$ projections $\bar{\Delta}_{ij,v}$ of mutual distances between all pairs of particles in each dimension v . Execution of this step can be conducted by $N_{\text{var}} \cdot (N_{\text{sim}})^2$ active threads, each of which stores its result into a unique global memory address,
- computation of $(N_{\text{sim}})^2$ absolute distances \bar{L}_{ij} between all pairs of particles. For obtaining a single value \bar{L}_{ij}^2 , it is needed to read and add up N_{var} projections $\bar{\Delta}_{ij,v}^2$. This sub-step therefore requires serialization of N_{var} writing procedures into each memory address containing the total squared distance \bar{L}_{ij}^2 ,
- computation of $N_{\text{var}} \cdot (N_{\text{sim}})^2$ repulsive accelerations $\ddot{x}_{ij,v}$ between all pairs of particles in each dimension. The result of this sub-step is a vector of $N_{\text{var}} \cdot N_{\text{sim}}$ total accelerations of each particle in each dimension. Therefore, it is required to sum N_{sim} repulsive accelerations for each particle in each dimension. Hence, this sub-step re-

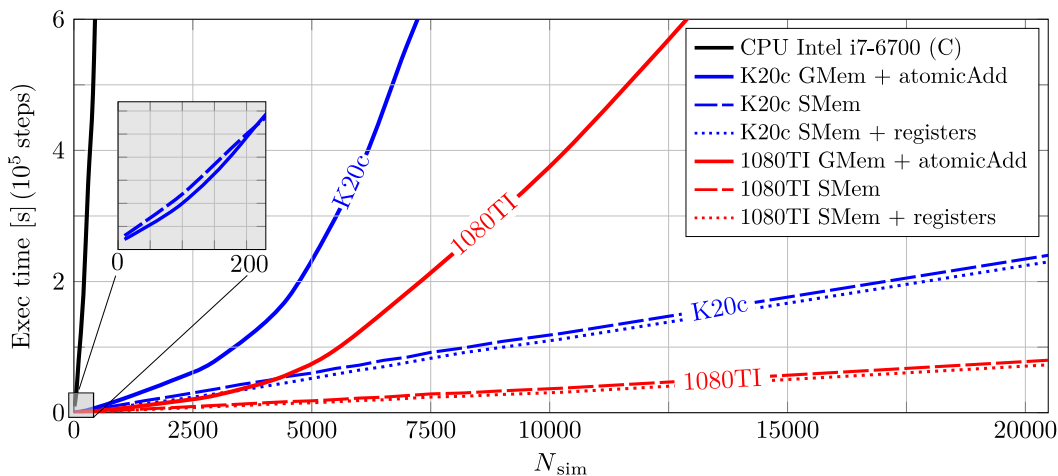


Figure 9.2: Performance comparison of the all-pairs $\mathcal{O}(N^2)$ solution while scaling N_{sim} in constant dimension of $N_{var} = 2$.

quires serialization of N_{sim} writing requests into each of $N_{var} \cdot N_{sim}$ global memory addresses.

- numerical integration of equations of motion using the semi-implicit Euler method; updating $N_{var} \cdot N_{sim}$ velocities $\dot{x}_{i,v}$ and coordinates $x_{i,v}$ of all particles in each dimension. Executed by $N_{var} \cdot N_{sim}$ active threads, this sub-step does not require any serialization of writing as each thread writes into its very own memory address.

Handling concurrent writing requests

In parallel computing, performance *bottlenecks* due to the need of serialization of writing are fairly common. In practice, there exist two most exploited approaches how to conduct serialization of writing of multiple threads into identical memory address.

One way of avoiding the concurrent writing of multiple threads into the same memory address is to eliminate such scenario entirely. Whenever an encounter of n threads writing into a single memory address is anticipated, it is instead possible to run a kernel function with a substitute thread, executing a loop of n instructions (those which would otherwise lead to thread collision in writing). A series of writing into the particular memory address is thus provided without possible code inconsistency.

A less firm approach how to avoid writing inconsistencies is to use GPU's *atomic operations*. Atomic (indivisible) operations are procedures implemented directly by the hardware manufacturer. These provide the possibility of conducting a read-modify-write task of a 32 or 64-bit value as a single uninterruptible action.

The manufacturer guarantees that during an atomic operation, no other threads can approach the memory address, or at least these cannot change the value stored. Until the Pascal architecture, Nvidia GPUs were capable of executing fast atomic operations only with values of the *float* precision. In case of the *double* precision values, the orders of magnitude slower `atomicCAS()` (compare-and-swap) method had to be manually implemented.

Approaches to handling concurrent writing requests are discussed in detail by the author in the conference paper [99] along with the speedup of using fast atomic operations.

9.4 Performance of parallelized solution

The following section offers performance benchmark of the two developed CUDA implementations when scaling the crucial parameters of the problem; the number of particles, N_{sim} , and the number of dimensions, N_{var} .

First, the performance of the massively parallel solution using Nvidia Tesla K20c and GTX 1080TI was compared with the single-thread implementation in the C programming language executed by the Intel i7-6700 CPU. All results presented in what follows are achieved using the *float* precision.

Figure 9.2 shows the computation time required when solving of 10^5 steps of the $\mathcal{O}(N^2)$ interaction while scaling the number of particles, N_{sim} , and keeping the dimension constant, $N_{\text{var}} = 2$. A significant speedup was reached when utilizing massive parallelization. Crucial is the qualitative difference of computation time rise when scaling the number of particles, N_{sim} .

The implementation using the on-chip shared memory (SMem) has shown very good performance in terms of execution time as well as its mild linear dependence on the number of particles, N_{sim} . Also, if the dimension, N_{var} , is known at the compilation time, it is possible to take advantage of partial usage of registers for storage of unknown particle accelerations, stresses and energies (SMem + registers) which leads to additional speedup.

The implementation relying on the use of the GPU's on-board global memory and fast atomic additions (GMem + atomicAdd) did not exhibit as good results compared to the shared memory implementation. The global memory solution performs worse both in execution time as well as in scaling with the number of particles. The only exception are scenarios up to around 200 particles, where the global memory implementation performs slightly better, see the inset in Figure 9.2. The reason is that for such a low number of particles, the partially serial solution in shared memory is not able to hide the arithmetic latency as well.

Next, we investigate the execution time when scaling both parameters of the problem; the number of particles, N_{sim} , as well as the number of dimensions, N_{var} . Figure 9.3a compares the execution time of solution of 10^3 steps of the $\mathcal{O}(N^2)$ interaction.

The CPU implementation is kept as a benchmark for shared memory implementations executed by the GTX 1080TI. The CUDA implementation delivers major improvement of performance as well as weaker execution time growth with increasing N_{sim} . Figure 9.3a shows that the parallel execution starts with linear dependency on N_{sim} . As the device starts to reach its peak bandwidth, the execution time scaling tends to $\mathcal{O}(N^2)$. As shown in Figure 9.3b, in the expected range of computation, the speedup in solution time by CUDA grows linearly with the number of particles, reaching up to $200\times$.

9.4.1 Bounds on parallel solution

The limitations of the shared memory implementation, see Section 9.3.1, are primarily set by the on-chip resources as each SM has to provide storage for $[(4 \cdot N_{\text{var}} + 2) \cdot \text{blockDim}]$ values for every thread block it is scheduled to maintain (96kB limit for GTX 1080TI).

The shared memory needed per thread block depends on the number of threads (warps) in each thread block. Therefore, the GPU performance and utilization also depend on this parameter. However, if tuned accordingly, the Pascal GPU approaches its peak bandwidth with each of its 28 SMs busy with around 64 warps and more, depending on the kernel

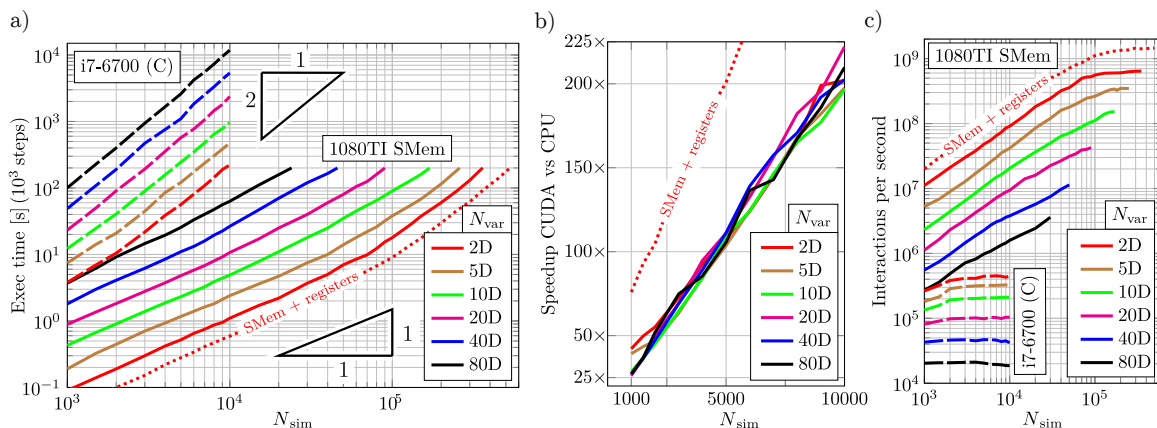


Figure 9.3: a) Performance comparison of the $\mathcal{O}(N^2)$ solution as executed by CPU (dashed) and GPU (solid and dotted lines) while scaling both N_{sim} and N_{var} . b) The achieved speedup of solution. c) Comparison of performance in interactions solved per second.

function, see [74]. Roughly, this means approaching GPU's peak bandwidth starting around 80 000 solved particles. This behavior has been observed in dimensions, N_{var} , from 2 to 10, see Figure 9.3c. Switching to multi-GPU after reaching a performance peak of a single GPU might be advised.

When increasing the dimensionality of the problem, the solution latency becomes more dependent on global memory bandwidth, bus size and also data structure. The total amount of shared memory requested, however, ultimately sets the upper bound on the size of the problem solved. Switching to a multi-GPU execution after reaching the shared memory limit might be advised.

In case of the global memory implementation, see Section 9.3.2, its limits are set by the size of global memory (11GB on GTX 1080TI). Register usage per thread in kernel functions is rather low (20 registers at most) compared to the maximum of 255 registers per thread on the GTX 1080TI.

9.5 Notes on parallel solution

Through this chapter, the parallel implementation of a dynamical system of mutually repelling particles has been presented. The particle system is assembled as a physical analogy of the Audze-Eglājs and ϕ optimization criteria for obtaining space-filling designs. From this purpose rises the unusual property of the arbitrary dimension of the design space. The dimension that is unknown at the time of compilation stipulates specific restrictions of utilization of GPU's fast on-chip resources. Two fundamentally different parallel implementations are therefore developed and their performance is compared to the initial single-thread CPU implementation.

The parallel implementation proposing utilization of the on-chip memory for a serialized interaction of tiles of particles is built on the concept tiled interaction of data. The on-chip memory allocation is however a subject of refinement due to the unknown dimensionality of the particle system. The usage of registers for storage vector structures is argued to be inappropriate. The usage of the shared memory is proposed for both maintaining the com-

puted data as well as for circulation of tiles of particles. Registers are consciously utilized only for maintaining intermediate values that do not depend on dimension to suppress register demands in high dimensions. Registers are used e.g. for conditional treatment of boundary conditions, temporary storage of frequently accessed values in computation of accelerations, and keeping values on chip during numerical integration.

Along with the on-chip solution, an entirely general implementation has been developed, utilizing solely GPU's on-board global memory. It investigates the approach of unfolding all the possible parallelism into the global memory and hiding the memory latency by computing all possible data in parallel. Large reductions required are handled by fast atomic additions.

Despite the shared memory resources being rather limited, the on-chip implementation turns out to be the best performing solution. Moreover, for a significant span of the number of particles, N_{sim} , the on-chip solution retains a $\mathcal{O}(N_{\text{sim}})$ complexity. Such a parallel execution of tiles of independent data also became a framework for other related implementation topics, see later in Chapter 12.

Part III

NUMERICAL EXPERIMENTS

Chapter 10

Selected properties of the dynamically optimized samples

Throughout Chapter 5, the change towards periodic boundary conditions of the Audze-Eglājs and ϕ_p optimization criteria has been proposed in pursuit of obtaining statistically uniform samples. To yield uniformly distributed sampling points in each sample, another enhancement of the ϕ_p criterion has been accomplished by reflecting the dimension of the design domain, N_{var} . By using the derived minimum value of the exponent $p \geq N_{\text{var}} + 1$ and/or a sufficient number of envelopes of periodically repeated images of particles, the presented dynamical system yields well distributed, statistically uniform samples. In other words, increasing the value of exponent leads to increasing of the density of the shortest distances.

Without using the periodical extension of the design space, the particles of the system tend to oversaturate the boundaries and corners of the design domain, as shown in Figure 10.2a. This behavior leads to non-uniformity of each sample and also to statistical non-uniformity across many samples. Inevitably, using such biased samples for statistical estimations leads to biased results, as will be shown further. However, even in the bordered design domain with intersite distance metric, rising the value of the exponent, p , leads to a major improvement in sample uniformity, see Figure 10.1. Note that such samples may still find a good use in fields other than Monte Carlo-type estimation (e.g. Kriging).

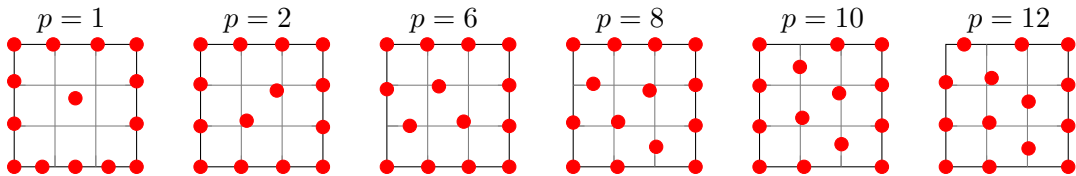


Figure 10.1: Optimal ϕ_p designs of $N_{\text{sim}} = 14$, $N_{\text{var}} = 2$ for various exponents, p , and the intersite Euclidean distance. Note the increased frequency of the shortest distances.

When considering the periodically repeated, borderless domain, the refined ϕ_p criterion tends to prefer self-similar patterns of uniformly distributed design points, see Figure 10.2b and also recall Figures 5.6 and 5.7. Despite that both the Euclidean distance metric and the formulation of energy potential are invariant with respect to rotation (shall not prefer

any directional setup), the dynamical particle system occasionally prefers designs aligned along individual dimensions. This is because the periodical extension of the design space is invariant only with respect to translations in the N_{var} -directions. This is especially the case where there is an exact number of particles, N_{sim} , to assemble a perfect and complete grid such as in Figure 10.2b. Due to this directional alignment, these otherwise very desirable samples exhibit a certain degree of the undesired collapsibility.

To remedy the issue of collapsibility, one can search for inspiration in the LHS method. It is indeed possible to use the refined ϕ_p criterion as an objective function for swapping of LHS-generated coordinates, similarly as in [41] and [100]. Typical designs obtained by such an optimization are shown in Figures 10.2c¹ and 10.2d, respectively. The time demands of such an optimization are, however, far beyond costs requested by the parallelized dynamical simulation. Moreover, the convergence to a minimum of the criterion/potential energy is much faster and more elegant in the dynamical simulation. It is because equations of motion guide all unrestrained particles to the optimal position at the same time. Furthermore, the dynamical simulation does not require computation of the value of the criterion at all; the behavior of the system is entirely contained in equations of motion. Conversely, the LHS-switching paired with simulated annealing performs only a single swap of coordinates of two points and needs to evaluate the criterion after each step. After all, the optimization using a dynamical system has been proposed as a counterpart of the rather inefficient LHS-switching. Both optimization methods of explicit dynamical system and simulated annealing possess incomparable parameters that influence their respective convergence and execution time. It might be therefore misleading to attempt to compare their specific execution times. Generally though, to achieve a well optimized sample, the LHS-switching algorithm [39] takes orders of magnitude longer.

Latinization of the dynamically optimized samples

To avoid collapsibility of dynamically optimized samples, we propose to perform the dynamical optimization as is and rectify the possible collapsibility afterwards, if it is of interest. A quite convenient approach to handle collapsibility of a sample is to perform its *latinization*, meaning to sort and rearrange the coordinates of sampling points in the sense of LHS method. In other words, to translate each sampling point along each dimension to the center (strata median, see [39]) of the closest LHS cell with coordinates:

$$x_{i,v}^{\text{LHS}} = \frac{\pi_i - 0.5}{N_{\text{sim}}}, \quad (10.1)$$

where π_i is a permutation of $i = 1, 2, \dots, N_{\text{sim}}$. Let us consider complete, ideal grids in the sense of ϕ_p criterion as in Figures 10.3a. To latinize a sample, the coordinates of points in each dimension must be replaced by their ranks that are subsequently re-scaled to the $(0, 1)$ interval using Equation (10.1). The problem with collapsible samples is that if multiple points occupy the identical coordinates, ordering these identical numbers essentially randomizes the design. As shown in Figure 10.3b, the resulting latinized designs cease to carry the original optimized pattern and their uniformity decreases towards plain random LHS.

¹Note that the LHS ϕ_p intersite design seems much more uniform than DYN ϕ_p intersite. This is because the prescribed LHS coordinates do not allow to oversaturate the boundary as much.

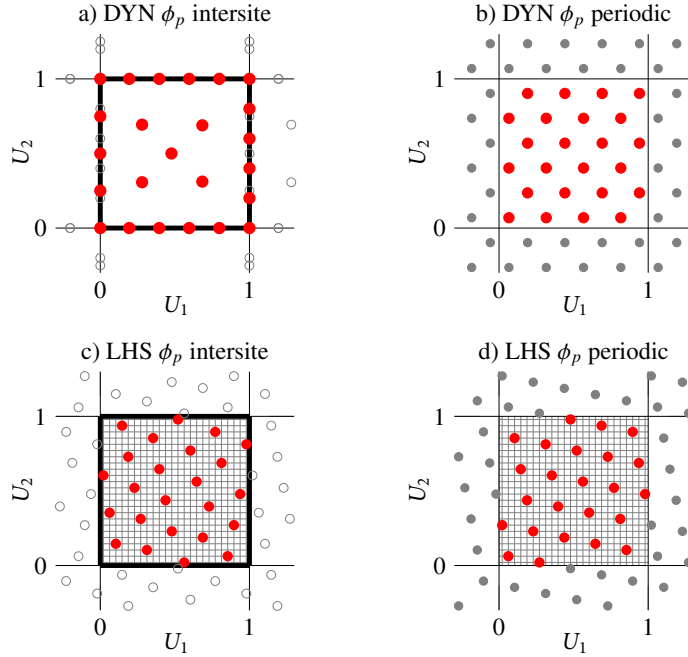


Figure 10.2: Samples optimized by using the refined ϕ_p criterion by dynamical simulation and LHS-switching.

Superior results can be obtained if the sample is slightly rotated before latinization. The angle of rotation shall not be too large to reorder the sample. On the other hand, it shall be large enough to produce distinguishably different coordinates of points that were in a row. As an empirical value, we propose this rotation to be approximately $\pm \arctg(\ell_{\text{char}})/4$. By performing such a rotation of a sample in all directions, the subsequent latinization yields well distributed LHS samples, see Figure 10.3c. These latinized samples tend to stay close to the original pattern from dynamical optimization while eliminating collapsibility.

For a significant portion of such post-processed samples, it is possible to make a bold conclusion that these carry a pattern that is identical to the ideally optimized LHS samples optimized by switching, compare samples in Figures 10.2c and 10.3 bottom right. However, such a property latinized samples from dynamical optimization is not generally valid. This is due to the fundamental difference between said optimization techniques: the LHS-switching optimizes directly firm LHS samples whereas the dynamically optimized samples are sorted to LHS after the optimization is done. Therefore, statistically, the latinized samples from the dynamical simulation cannot possess identical properties to LHS-switching and typically exhibit slightly larger variance of estimation.

It is worth noting that, as a side effect, any transformation to LHS also naturally “centers” the sample. This results in high consumption of sampling points to advance towards the edges of the domain. Nevertheless, points within a latinized sample can be additionally translated by random shifts along each dimension, either the whole sample as a rigid body or each sampling point by its own random shift. Such a random translation increases the variance of estimation as a price for an unbiased estimation. Centering of the sample, on the other hand, decreases the variance of estimation but biases the mean value of estimation.

Therefore, it depends on the particular problem at hand whether centered samples are desired or not.

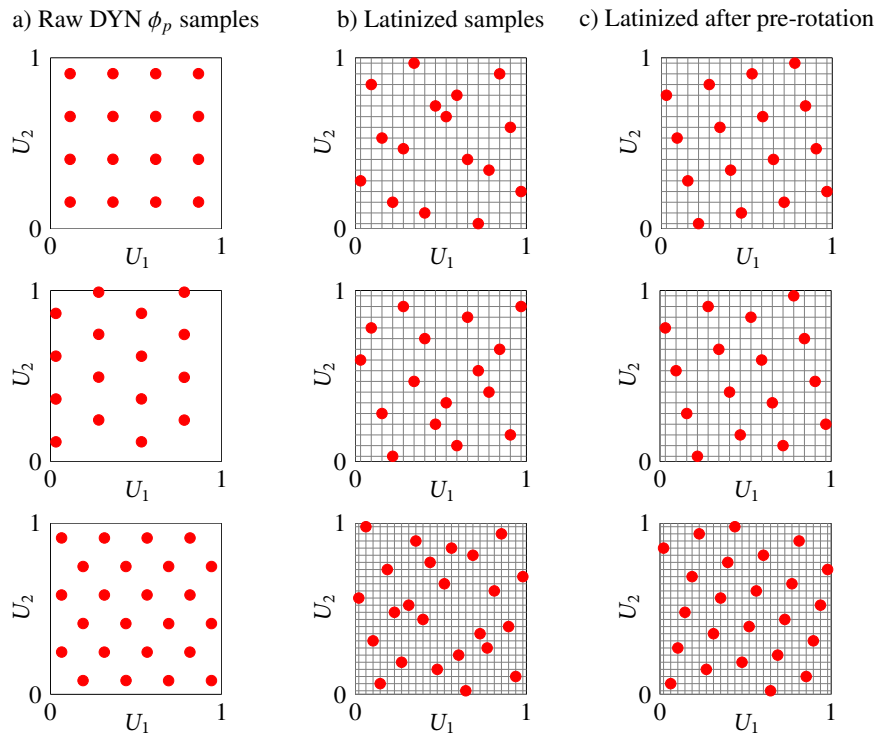


Figure 10.3: Rectification of collapsible grids obtained by dynamical simulation using latinization and pre-rotation.

Chapter 11

Numerical integration performance

Through the following chapter, the performance of the dynamically optimized samples will be tested in various use-cases of Monte Carlo-type approximation. To demonstrate the characteristic properties of dynamically optimized samples and the refined ϕ_p criterion, respectively, these samples will be compared to a plain Monte Carlo integration, LHS-random samples and LHS samples optimized by the Periodic Audze-Eglājs criterion [41]. The nomenclature of the sampling methods used is summarized in Table 11.1.

Marking	Optimization method used
MC RAND	Random Monte Carlo samples
LHS RAND	LHS samples with a random order of points evaluated at strata median, see e.g. [39].
DYN ϕ_p Intersite	Samples optimized by the dynamical system using the refined ϕ_p potential, intersite Euclidean metric, bordered design domain
DYN ϕ_p Periodic	Samples optimized by the dynamical system using the refined ϕ_p potential, shortest-distance Euclidean metric, periodic design domain
DYN ϕ_p Periodic + LHS	DYN ϕ_p Periodic samples latinized after pre-rotation
LHS PAE	Samples optimized by mutual swapping of LHS coordinates governed by Periodic Audze-Eglājs criterion as proposed in [41].

Table 11.1: Marking of compared optimization methods.

Because problems featuring various numbers of input random variables will be featured, it is desirable to select a measure of number of integration points that, rather than to the actual number of points, refers to the *characteristic saturation of the design domain* with integration points. Such a measure would allow to represent the results in a normalized scale. It has been discussed in Section 5.2 that the *characteristic length*, ℓ_{char} , does represent

such a measure, recall Equation (5.15):

$$\ell_{\text{char}} = \frac{1}{N_{\text{var}}\sqrt{N_{\text{sim}}}}. \quad (11.1)$$

Therefore, all results that follow throughout this chapter will use ℓ_{char} to express how the domain is filled with integration points. For convenience, additional axis with the corresponding values of N_{sim} will be also provided.

Each of the examples that follow aims to illustrate a distinctive scenario of using statistical approximation. Because both optimization algorithms and generation of random samples exhibit a degree of randomness, the estimated characteristics are treated as random variables and the results with given settings of N_{sim} and N_{var} are obtained N_{run} times. The number of runs, N_{run} , is set to 400, where not stated otherwise.

Averages (ave) and sample standard deviations (ssd) are calculated from N_{run} results obtained for any of the studied characteristics.

As will be shown, the benefits of using optimized samples vary depending on each particular problem. Therefore, only generalized conclusions will be drawn from the selected numerical results. Also, comments based on observations of related, elsewhere published or unpublished simulations done by the author will be amended where appropriate.

11.1 Approximation of definite integral

An approximation of a definite integral has been selected as a suggestive benchmark problem for study of the dynamically optimized samples. The three examples introduced in Section 3.1 will be used primarily for demonstration of the effect of introducing periodic boundary conditions into the ϕ_p criterion in order to achieve statistically uniform samples (unbiased estimation). The actual estimation of the mean value, μ_J , of the definite integral, J , follows Equation (3.2), recall:

$$J_{MC} = \hat{\mu}_J = \frac{1}{N_{\text{sim}}} \sum_{\text{sim}=1}^{N_{\text{sim}}} I_{\Omega}(\mathbf{x}_{\text{sim}}), \quad (11.2)$$

i.e. a sum of all points that belong into the region Ω . Whether a point is deemed to belong into the area of interest is determined by evaluating the respective Indicator function I_{Ω} , generally:

$$I_{\Omega}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in \Omega \\ 0 & \text{otherwise} \end{cases} \quad (11.3)$$

For each studied example, the Indicator function differs for each of the three cases:

- Example A:

$$I_{\Omega_A}(\mathbf{x}) = \begin{cases} 1 & \text{if } r < x_0 < (1-r) \wedge \\ & \wedge \frac{1}{2} - \sqrt{r^2 - (x_0 - \frac{1}{2})^2} < x_1 < \frac{1}{2} + \sqrt{r^2 - (x_0 - \frac{1}{2})^2} \\ 0 & \text{otherwise} \end{cases} \quad (11.4)$$

- Example B:

$$I_{\Omega_B}(\mathbf{x}) = \begin{cases} 1 & \text{if } 0 < x_0 < 2r \wedge \\ & \wedge (1-r) - \sqrt{r^2 - (x_0 - r)^2} < x_1 < (1-r) + \sqrt{r^2 - (x_0 - r)^2} \\ 0 & \text{otherwise} \end{cases} \quad (11.5)$$

- Example C:

$$I_{\Omega_C}(\mathbf{x}) = \begin{cases} 1 & \text{if } 0 < x_0 < r \wedge 1 - \sqrt{r^2 - x_0^2} < x_1 \\ 1 & \text{if } 0 < x_0 < r \wedge \sqrt{r^2 - x_0^2} < x_1 \\ 1 & \text{if } (1-r) < x_0 < 1 \wedge 1 - \sqrt{r^2 - (x_0 - 1)^2} < x_1 \\ 1 & \text{if } (1-r) < x_0 < 1 \wedge \sqrt{r^2 - (x_0 - 1)^2} < x_1 \\ 0 & \text{otherwise} \end{cases} \quad (11.6)$$

The goals of these *patch tests* are (i) to compare the efficiency of samples when used for estimation of mean value and standard deviation of definite integrals J_A , J_B , J_C and (ii) to demonstrate the need for statistical uniformity of used statistical samples.

For the examples presented in this section, the circle radius of $r = 0.25$ is considered. The exact solution of the definite integrals J_A , J_B , J_C is then:

$$\begin{aligned}
 J_A = J_B = J_C = \mu_J = \pi r^2 &= \int_0^1 \int_0^1 I_\Omega(\mathbf{x}) \overbrace{f_{\mathbf{X}}(x_1, x_2)}^{X \sim \mathcal{U}(0,1)} dx_1 dx_2 = \\
 &= \int_0^1 \int_0^1 I_\Omega(\mathbf{x}) dx_1 dx_2 = \iint_{\Omega} dx_1 dx_2 = 0.196\,349\dots
 \end{aligned} \tag{11.7}$$

Along with the mean value, μ_J , the standard deviation, $\sigma_J \equiv \sigma[J]$, of the random volume fraction will be estimated. The exact solution in the continuous form reads:

$$\begin{aligned}
 \sigma_J^2 &= \int_0^1 \int_0^1 (I_\Omega(\mathbf{x}) - \mu_J)^2 f_{\mathbf{X}}(x_1, x_2) dx_1 dx_2 = \int_0^1 \int_0^1 (I_\Omega(\mathbf{x}) - \mu_J)^2 dx_1 dx_2 = \\
 &= \int_0^1 \int_0^1 I_\Omega(\mathbf{x})^2 dx_1 dx_2 - 2 \int_0^1 \int_0^1 (I_\Omega(\mathbf{x}) \mu_J) dx_1 dx_2 + \int_0^1 \int_0^1 \mu_J^2 dx_1 dx_2 = \\
 &= \mu_J - 2\mu_J^2 + \mu_J^2 = \mu_J(1 - \mu_J) \Rightarrow \sigma_J = \sqrt{\mu_J(1 - \mu_J)} = 0.397\,235\dots
 \end{aligned} \tag{11.8}$$

Its estimated value, $\hat{\sigma}_J$, is computed as sample standard deviation:

$$\hat{\sigma}_J = \sqrt{\frac{1}{N_{\text{sim}}} \sum_{\text{sim}=1}^{N_{\text{sim}}} I_\Omega(\mathbf{x}_{\text{sim}})^2 - \hat{\mu}_J^2} = \sqrt{\hat{\mu}_J - \hat{\mu}_J^2} = \sqrt{\hat{\mu}_J(1 - \hat{\mu}_J)}. \tag{11.9}$$

11.1.1 Discussion of numerical results

In this basic set of examples, basic samples of MC RAND and LHS RAND are compared to DYN ϕ_p Intersite and DYN ϕ_p Periodic samples. For all configurations of **Examples A, B, and C**, the identical mean value, μ_J , and standard deviation, σ_J , should be obtained. The results are presented in Figures (11.1) and (11.2) for **Example A**, Figures (11.3) and (11.4) for **Example B**, and Figures (11.5) and (11.6) for **Example C**. After studying these results, one can note following conclusions on performance of estimation of mean value, μ_J , and standard deviation, σ_J :

- as expected, MC RAND samples provide an unbiased estimation of the mean value in all cases, but the estimation exhibits the largest standard deviation of estimation,
- LHS RAND samples, in general, do not yield an unbiased estimation of the mean value. Especially small samples suffer from strongly biased estimation. As the saturation of the domain grows, the estimated mean value continues to oscillate around the correct mean value. The standard deviation of estimation is never greater than that of random MC RAND samples, it typically stays at 70% of MC RAND,
- DYN ϕ_p Intersite samples, dynamically optimized within a bounded design domain, provide a strongly biased estimation of the mean value and standard deviation in all three cases due to oversaturated boundaries of the domain. As is apparent, the estimation does not even converge to the exact values when rising the number of integration

points. Therefore, it is strongly not recommended to use any kind of intersite-optimized samples including any combination of these with LHS for statistical estimation (integration) of Monte Carlo type, despite low variance of estimation. However, it is worth noting that such samples might often perform well in other use-cases such as building metamodels, see e.g. *kriging* [101] and [27],

- conversely, the DYN ϕ_p Periodic samples that use both the derived minimum value of the exponent, p , as well as the periodically repeated design domain, provide unbiased estimation of the mean value due to their property of statistical uniformity. Furthermore, due to well-optimized point layouts, the DYN ϕ_p Periodic yield significantly lower standard deviation of estimation in comparison to all shown methods. In fact, the ratio of standard deviation of DYN ϕ_p Periodic to MC RAND samples further tends to improve with the number of particles that can be optimized: starting at about 85% and lowering to around 20%. This is due to a faster decrease in variance (and standard deviation, respectively) when using DYN ϕ_p Periodic samples. Typically, such a phenomenon appears especially for small sample sizes, roughly of $\ell_{\text{char}} > 0.4$. This behavior will be even more apparent in the next example.

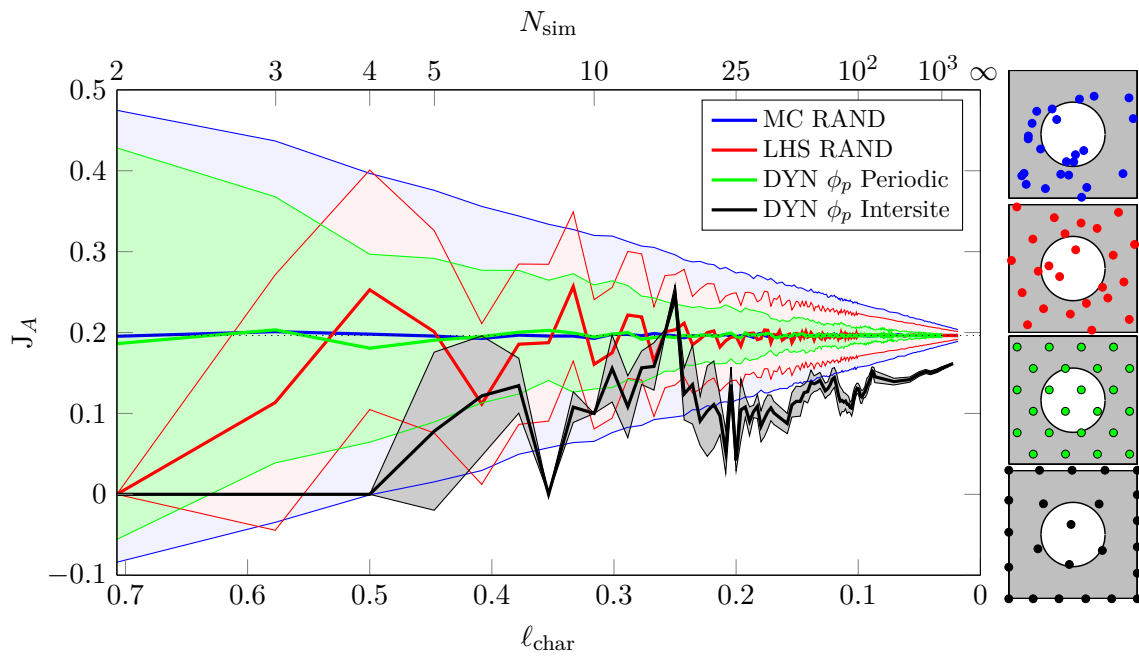


Figure 11.1: Estimated mean value of the integral J_A (ave, ave \pm ssd).

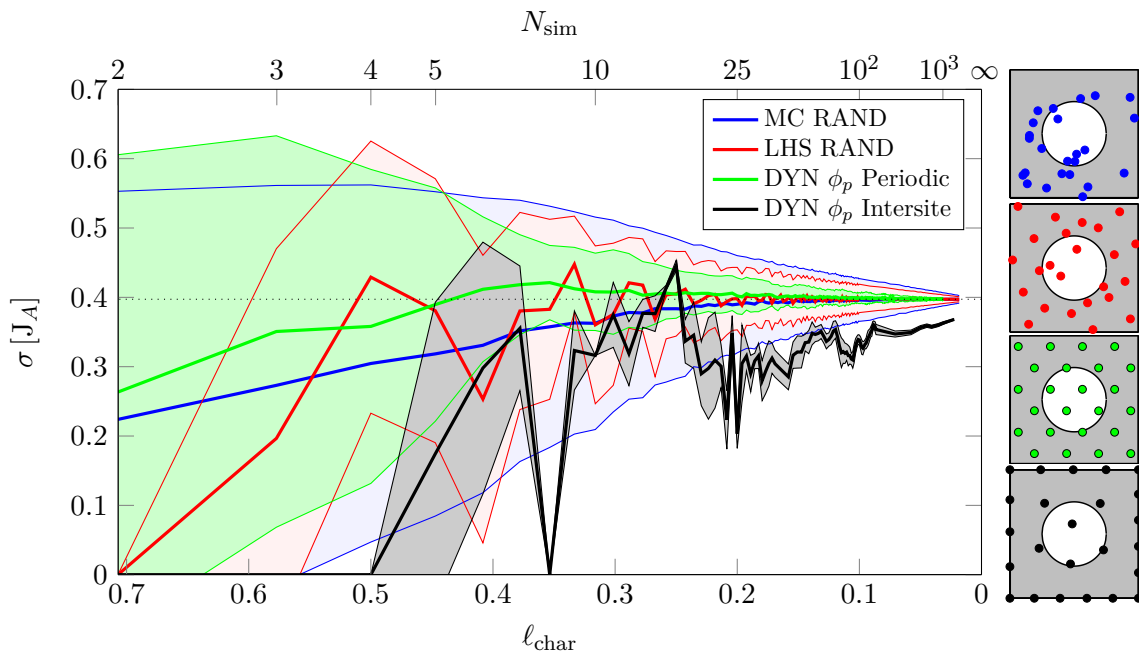


Figure 11.2: Estimated standard deviation of the integral J_A (ave, ave \pm ssd).

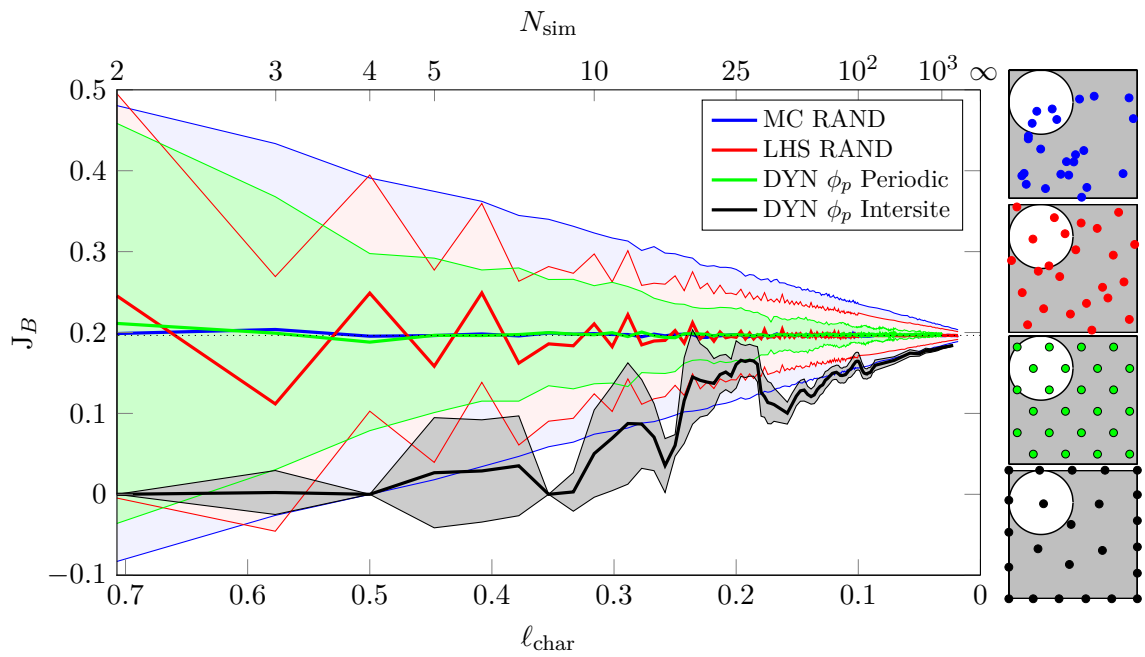


Figure 11.3: Estimated mean value of the integral J_B (ave, ave \pm ssd).

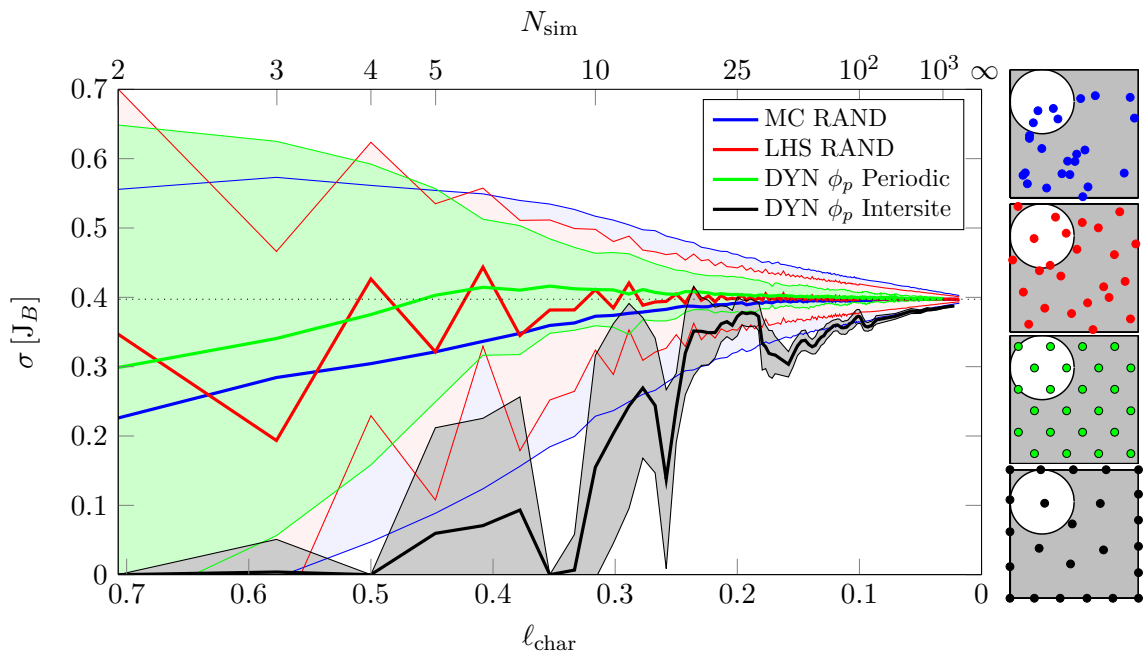


Figure 11.4: Estimated standard deviation of the integral J_B (ave, ave \pm ssd).

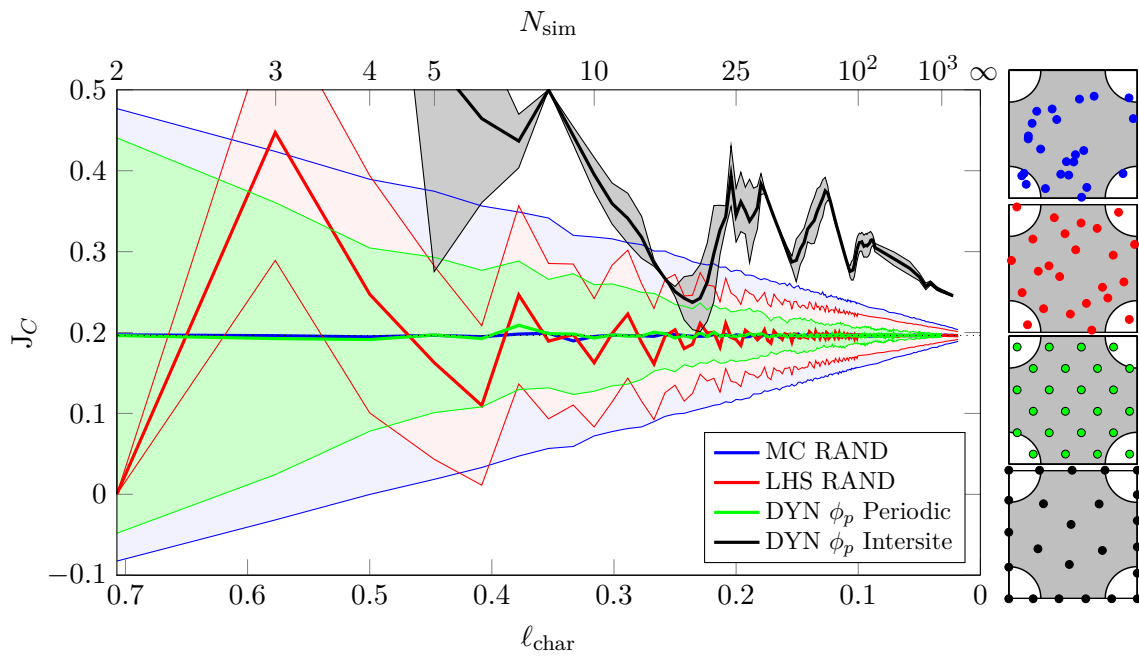


Figure 11.5: Estimated mean value of the integral J_C (ave, ave \pm ssd).

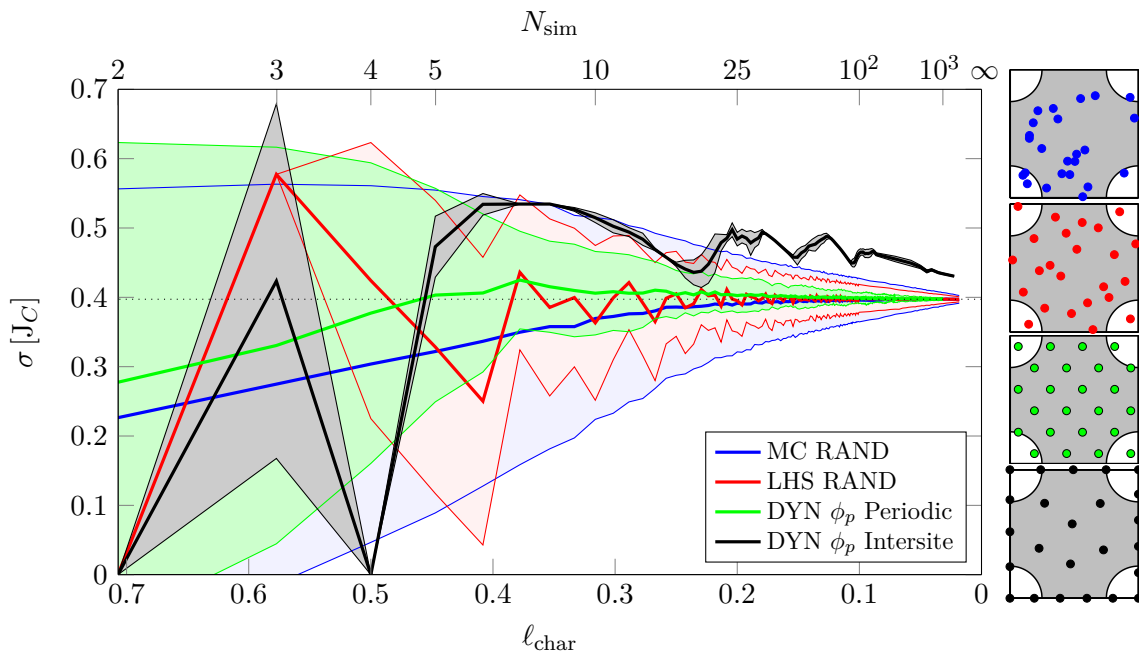


Figure 11.6: Estimated standard deviation of the integral J_C (ave, ave \pm ssd).

11.2 Sum of exponentials of normal variables

The second example aims to represent an approximation of characteristics of a function of random variables. The selected problem has been featured in [37] and also in [41] for evaluating of performance of LHS samples optimized by switching governed by the Audze-Eglājs criterion that considers periodically repeated design domain (periodic Audze-Eglājs – PAE). The studied example considers a function $g_{\text{exp}}(\mathbf{X})$ of input random vector \mathbf{X} :

$$g_{\text{exp}}(\mathbf{X}) = \sum_{v=1}^{N_{\text{var}}} \exp(-X_v^2) = \sum_{v=1}^{N_{\text{var}}} \exp\left(F_v^{-1}(u_v)^2\right). \quad (11.10)$$

The input random variables are considered as independent and identically distributed (IID) random variables of multivariate normal distribution $\mathcal{N}(0, 1)$. Each marginal X_v is described by its probability density function $f_v(x)$:

$$f_v(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \quad \text{for } -\infty \leq x \leq \infty, \quad (11.11)$$

and cumulative distribution function $F_v(x)$:

$$F_v(x) = \frac{1}{2} \left[1 + \operatorname{erf}\left(\frac{x-\mu}{\sqrt{2}\sigma}\right) \right] \quad \text{for } -\infty \leq x \leq \infty, \quad (11.12)$$

where erf is the *error function* defined as:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2) dt. \quad (11.13)$$

The inverse cumulative distribution function, $F_v^{-1}(u)$, required for the transformation of coordinates of sampling points from the space of sampling probabilities reads:

$$F_v^{-1}(u) = \mu + \sigma\sqrt{2} \operatorname{erf}^{-1}(2u - 1) \quad \text{for } 0 \leq u \leq 1. \quad (11.14)$$

The task of this example is, again, to estimate the mean value, μ_{exp} , and standard deviation, σ_{exp} , of the function g_{exp} in dimensions $N_{\text{var}} = 2$ and $N_{\text{var}} = 5$. The derivation of the exact values of μ_{exp} and σ_{exp} is provided in Appendix B of [41]:

$$\begin{aligned} \mu_{\text{exp}} &= N_{\text{var}} \frac{\sqrt{3}}{3}, \\ \sigma_{\text{exp}} &= \sqrt{N_{\text{var}}} \sqrt{\frac{\sqrt{5}}{5} - \frac{1}{3}}. \end{aligned} \quad (11.15)$$

In particular, for the two-dimensional case of g_{exp} , the values of $\mu_{\text{exp},2\text{d}}$ and $\sigma_{\text{exp},2\text{d}}$ read:

$$\begin{aligned} \mu_{\text{exp},2\text{d}} &= 2 \frac{\sqrt{3}}{3} \approx 1.154\,701, \\ \sigma_{\text{exp},2\text{d}} &= \sqrt{2} \sqrt{\frac{\sqrt{5}}{5} - \frac{1}{3}} \approx 0.477\,243. \end{aligned} \quad (11.16)$$

Finally, the values of $\mu_{\text{exp},5\text{d}}$ and $\sigma_{\text{exp},5\text{d}}$ of the five-dimensional g_{exp} are as follows:

$$\begin{aligned} \mu_{\text{exp},5\text{d}} &= 5 \frac{\sqrt{3}}{3} \approx 2.886\,751, \\ \sigma_{\text{exp},5\text{d}} &= \sqrt{5} \sqrt{\frac{\sqrt{5}}{5} - \frac{1}{3}} \approx 0.754\,587. \end{aligned} \quad (11.17)$$

11.2.1 Discussion of numerical results

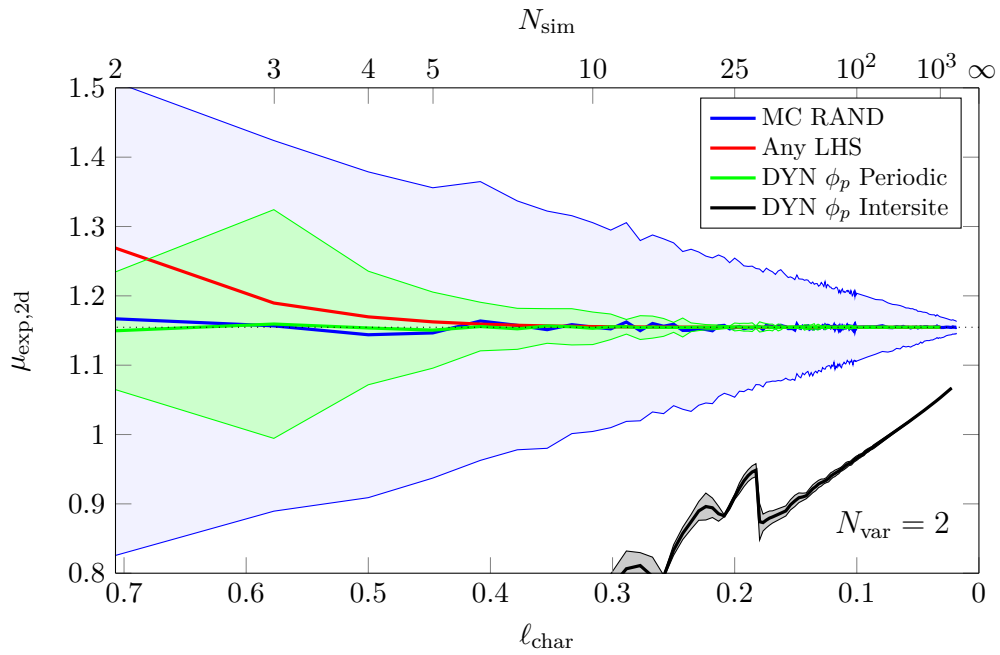
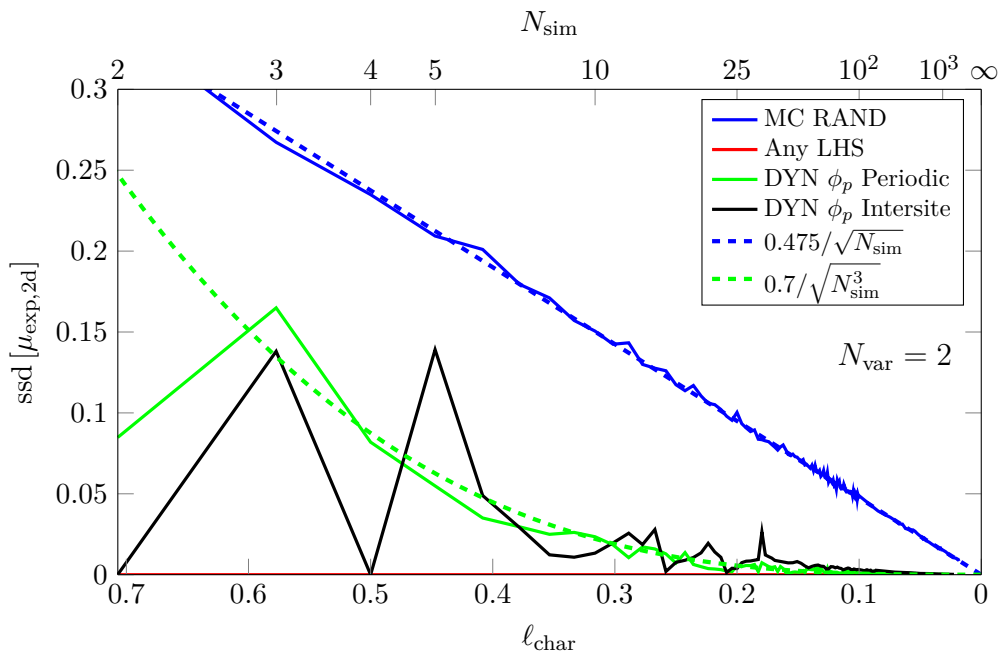
The numerical results of this example are presented in a similar way to what preceded. For both cases of $N_{\text{var}} = 2$ and $N_{\text{var}} = 5$, results of estimation of the mean value μ_{exp} (see Figures 11.7 and 11.11), and the standard deviation σ_{exp} (see Figures 11.9 and 11.13), are presented. Additionally, Figures 11.8, 11.10, 11.12, and 11.14 are provided, comparing standard deviation of estimation of the mentioned characteristics.

Thanks to all the plots being normalized by ℓ_{char} , one can observe a similar behavior of the dynamically optimized samples in both dimensions. The comparable range here begins around $\ell_{\text{char}} = 0.6$, where there starts to be enough particles for the dynamical system to assemble meaningful patterns.

Because the function $g_{\text{exp}}(\mathbf{X})$ represents a sum of independent marginals, any deterministic LHS sample is going to estimate the mean value μ_{exp} without any variance. In the two-dimensional case, the poor performance of DYN ϕ_p Intersite samples is also shown to remind the improvement of using periodically repeated design domain.

More than in the preceding example, the DYN ϕ_p Periodic samples bring reduction in variance of the estimated values. In fact, in both cases of $N_{\text{var}} = 2$ and $N_{\text{var}} = 5$, using the DYN ϕ_p Periodic samples leads to enhanced rate of convergence. If using DYN ϕ_p Periodic samples in the 2d example, the asymptotic rate of convergence attained is $\mathcal{O}(1/\sqrt{N_{\text{sim}}^3})$, whereas Monte Carlo samples exhibit the expected $\mathcal{O}(1/\sqrt{N_{\text{sim}}})$ rate of convergence. In the 5d variant of g_{exp} , using DYN ϕ_p Periodic samples yields rate of convergence of $\mathcal{O}(1/\sqrt{N_{\text{sim}}^2})$ as oppose to the characteristic $\mathcal{O}(1/\sqrt{N_{\text{sim}}})$ of MC RAND.

The results of estimation of the standard deviation σ_{exp} allow to compare the LHS samples to the latinized samples from dynamical optimization. Added are also the results the LHS PAE optimization proposed in [41]. As has been predicted earlier in Chapter 10, the samples latinized after dynamical optimization must exhibit a slightly higher standard deviation of estimation.


 Figure 11.7: Estimated mean value of $g_{\text{exp},2d}$ (ave, ave \pm ssd).

 Figure 11.8: Standard deviation of the estimated mean value of $g_{\text{exp},2d}$.

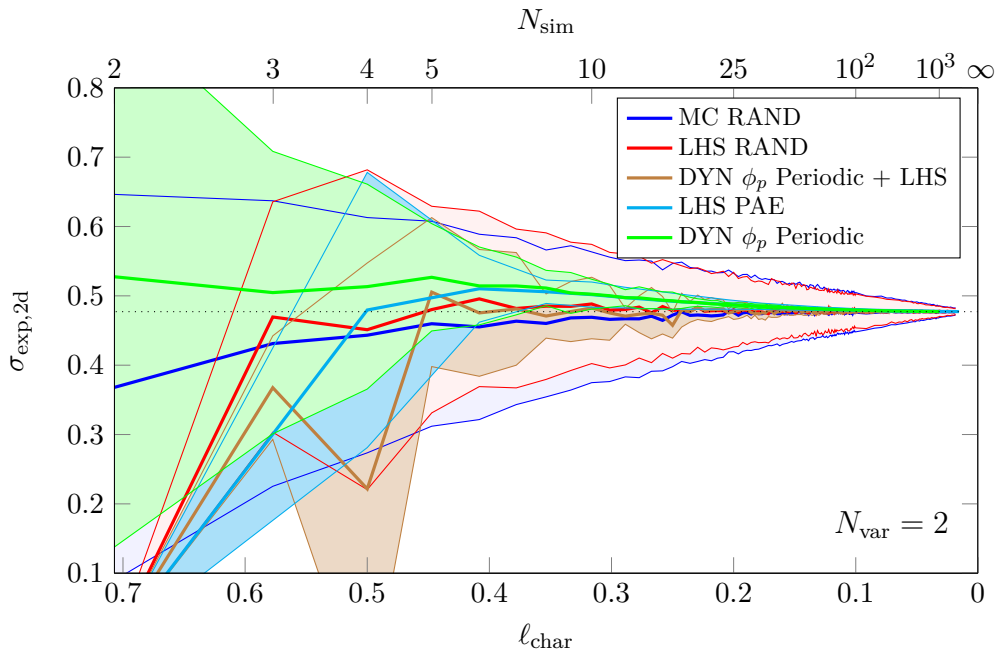


Figure 11.9: Estimated standard deviation of $g_{\text{exp},2d}$ (ave, ave \pm ssd).

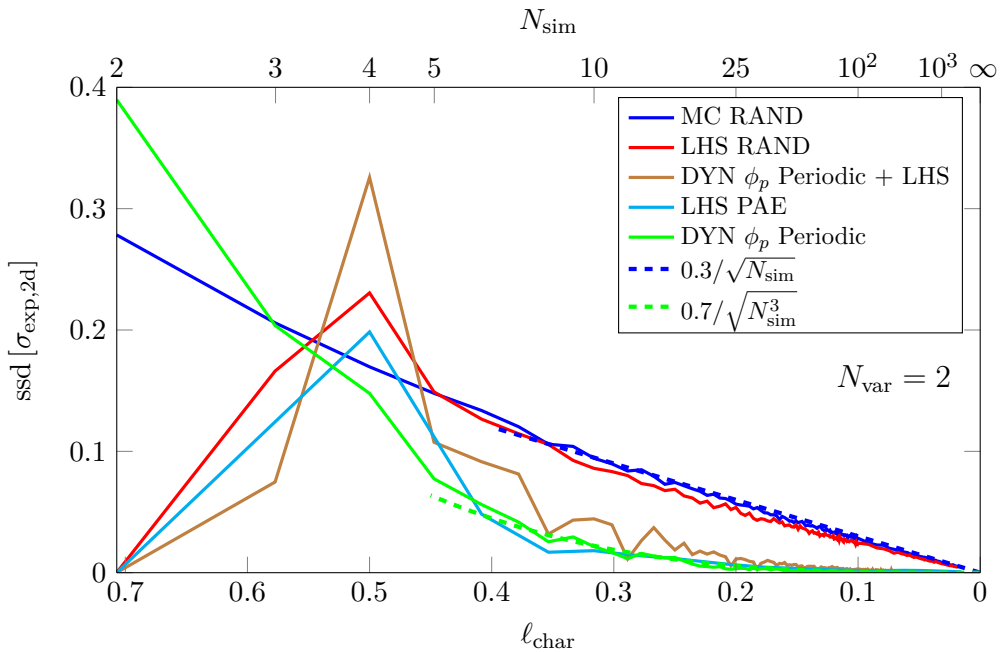


Figure 11.10: Sample standard deviation of the estimated standard deviation of $g_{\text{exp},2d}$.

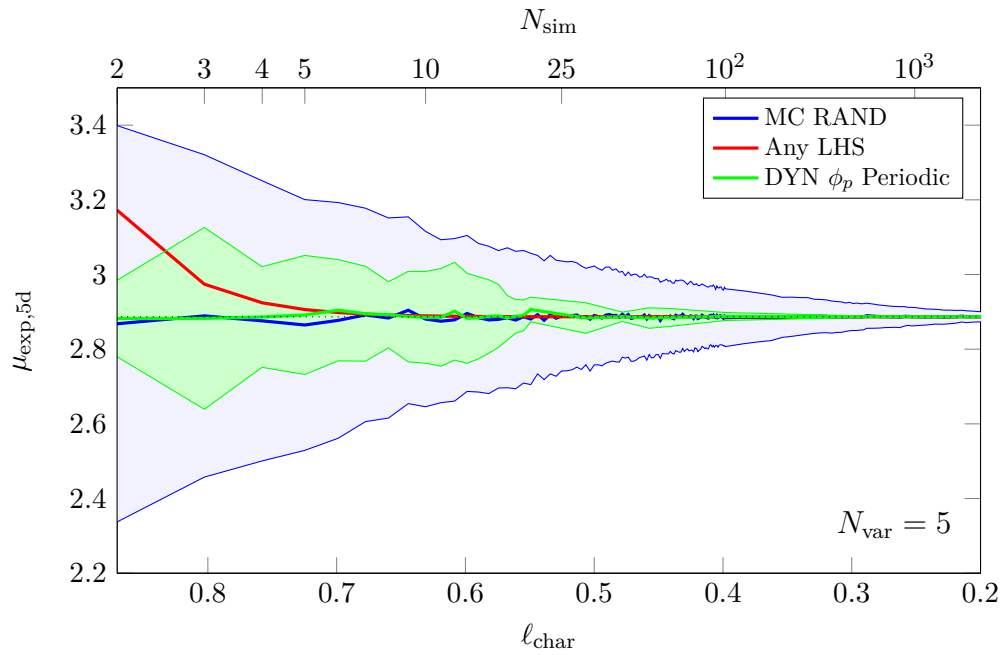


Figure 11.11: Estimated mean value of $g_{\text{exp},5d}$ (ave, ave \pm ssd).

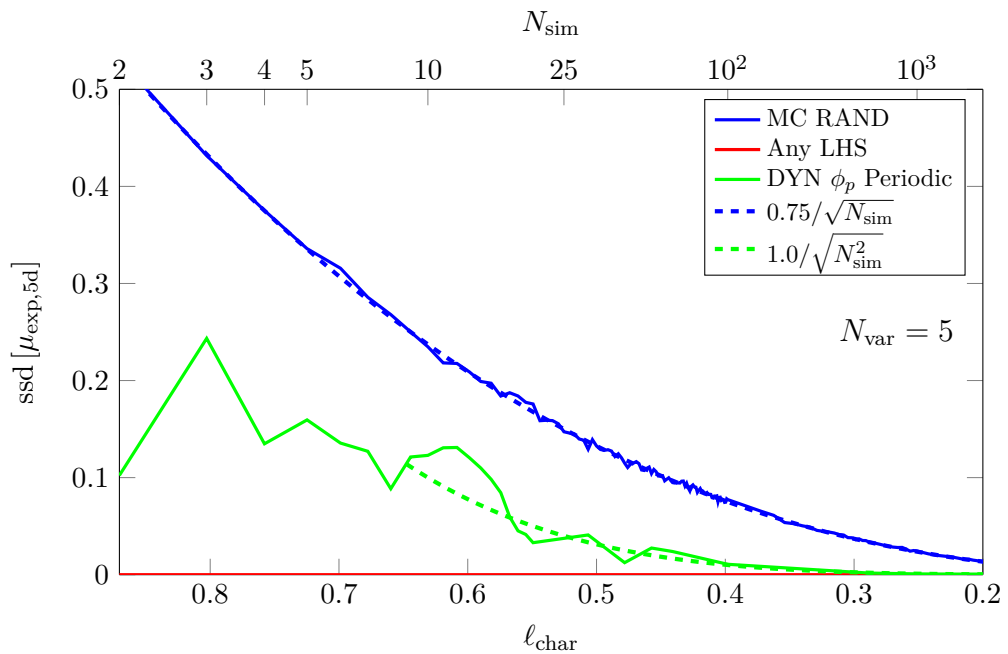


Figure 11.12: Sample standard deviation of the estimated mean value of $g_{\text{exp},5d}$.

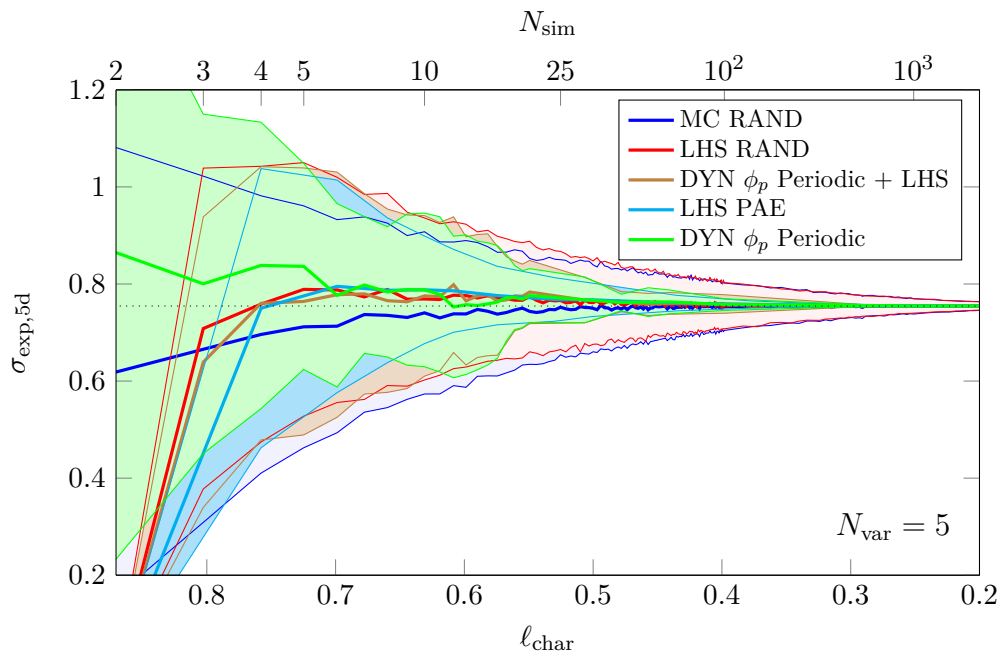


Figure 11.13: Estimated standard deviation of $g_{\text{exp},5\text{d}}$ (ave, ave \pm ssd).

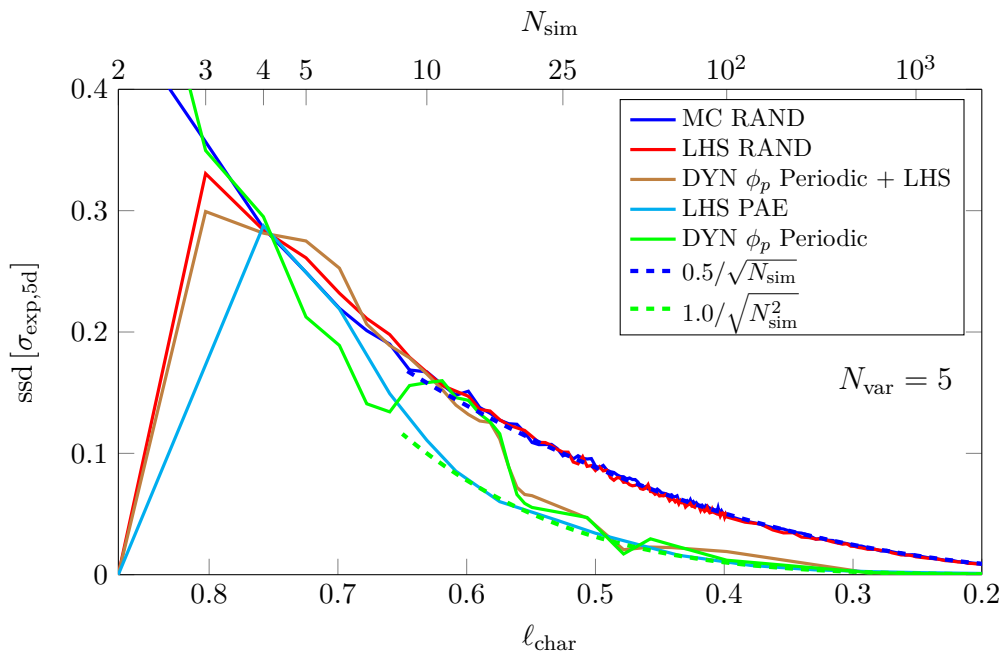


Figure 11.14: Sample standard deviation of the estimated standard deviation of $g_{\text{exp},5\text{d}}$.

11.3 Engineering example I - Failure of a truss model

Assume now an engineering problem of estimation of deflection and failure probability of a truss structure. The model is similar to the truss model used in [102] where it served as one of benchmark problems for comparison of methods for estimating structural reliability. The setup of the numerical experiment is illustrated in Figure 11.15.

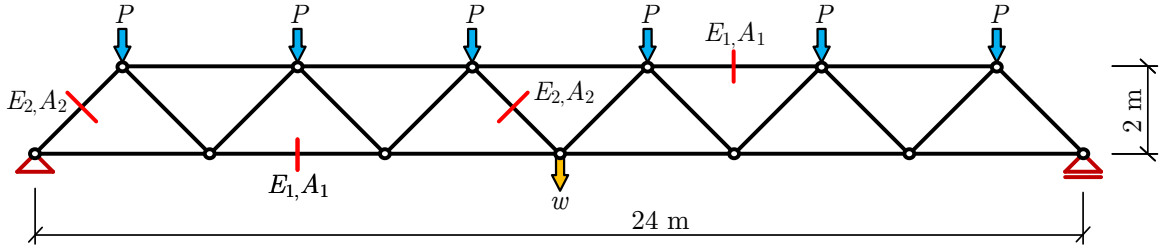


Figure 11.15: An illustration of the studied model of truss structure.

The input random vector of the model consists of five random variables: the material properties of horizontal members (Young's modulus E_1 and cross-section area A_1), the material properties of diagonal members (Young's modulus E_2 and cross-section area A_2) and the loading forces P in top joints. The properties of input random variables are summarized in Table 11.2. The task is to estimate the mean value and standard deviation of the midspan deflection w and the probability of failure, respectively. The failure is defined to occur when the deflection w exceeds the given value of 0.11 m.

Random variable			Distribution	Mean value	CoV
Young's modulus	E_1	[GPa]	Log-normal	210	0.1
Young's modulus	E_2	[GPa]	Log-normal	210	0.1
Cross-section area	A_1	[m ²]	Log-normal	2.0×10^{-3}	0.1
Cross-section area	A_2	[m ²]	Log-normal	1.0×10^{-3}	0.1
Load	P	[kN]	Gumbel	50	0.15

Table 11.2: Random variables of truss example and their properties.

The value of deflection w is solved by using method of virtual unit loads. The deflection w is therefore obtained as:

$$w = \sum_i \frac{N_i \bar{N}_i l_i}{E_i A_i}, \quad (11.18)$$

where N_i is the normal force in the i th member due to the load by forces P , \bar{N}_i is the normal force in the i th member due to virtual unit load at midspan, l_i is the length of the i th member, and E_i and A_i are the respective material and geometrical properties of the i th member.

11.3.1 Discussion of numerical results

The problem of failure probability of a truss structure has been selected to represent an engineering problem where the benefit of simply using uniform samples is expected to be rather low.

First, the results of estimation of mean value, μ_w , and standard deviation, σ_w , of deflection w are presented. Figure 11.16 shows the convergence of estimation of the mean value, μ_w , accompanied by Figure 11.17 that separately shows the sample standard deviation of the estimated mean value. The convergence of estimation of the standard deviation of deflection, σ_w , is shown in Figure 11.18. It is also complemented with Figure 11.19, separately showing the standard deviation of estimation of σ_w . Note that both kinds of LHS samples deliver a significant reduction in variance of the estimator, this is due the model (11.18) being a sum of weakly dependent terms, see asymptotically fitted curves in Figures 11.17, 11.19, and 11.21.

For a model that exhibits a rather low level of complexity, recall Equation (11.18), one can expect that samples with optimized uniformity will not provide as large reduction in variance of estimation. Indeed, throughout the results of estimation of both μ_w and σ_w , only a slight reduction in standard deviation of estimation can be concluded when comparing dynamically optimized samples to RAND samples.

Similar are the observations of estimation of the last approximated property of the system that is the probability of failure, expressed as $P(w > 0.11)$. The estimation of the mean value of failure probability, p_f , is provided in Figure 11.20. The standard deviation of this estimation is displayed separately in Figure 11.21.

In case of the studied truss model, the weak influence of using samples with optimized uniformity is considered to be due to (i) a relative simplicity of the linear model when estimating the deflection, w , and (ii) due to the known low performance of uniformly distributed samples for estimation of low probabilities. The estimation of probability failure is essentially not much different from estimation of definite integral: in both cases, the problem is governed by the specific Indicator function. Such “raw” samples with optimized uniformity are much more suitable for estimation of higher “probabilities” such as in the first example, see Section 11.1. Nevertheless, using optimized DYN ϕ_p Periodic samples requires about 1255 model evaluations to achieve an estimation of p_f with CoV of 10%. Compared to the 2320 simulations required by MC RAND samples, this means a computing time decrease by about 50%.

A different, much more suitable usage of such uniform samples is in coupling with, for instance, the Importance sampling method that aims to lower the variance of estimation of failure probability by transforming positions of sampling points to occur more frequently in the region of failure, see e.g. [103, 104]. The following Section is going to study the benefits of optimized samples along with the Importance sampling method.

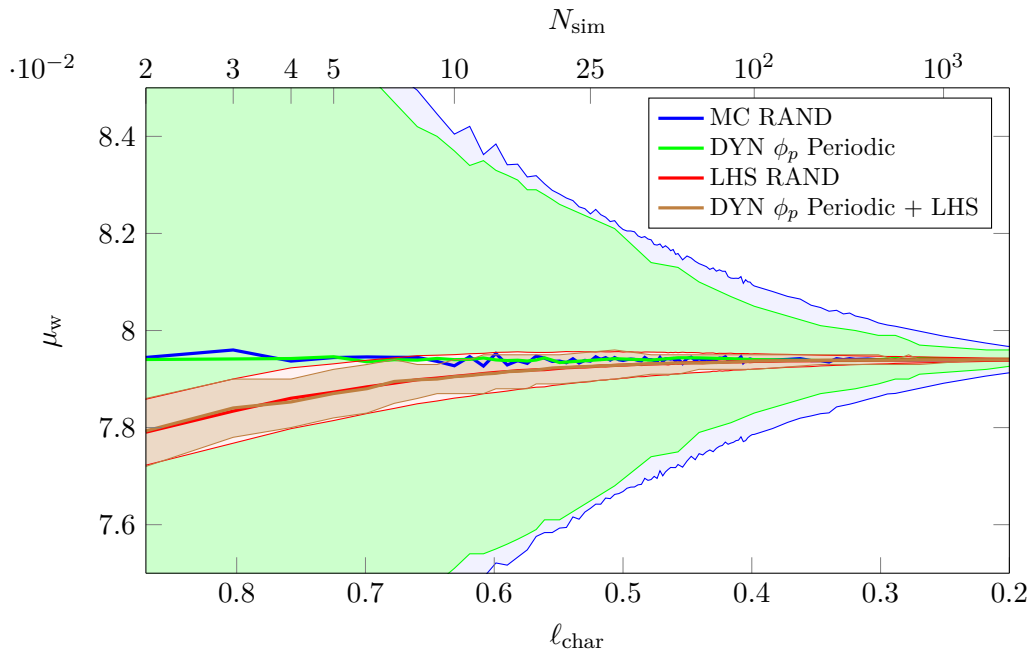


Figure 11.16: Estimated mean value of deflection w (ave, ave \pm ssd).

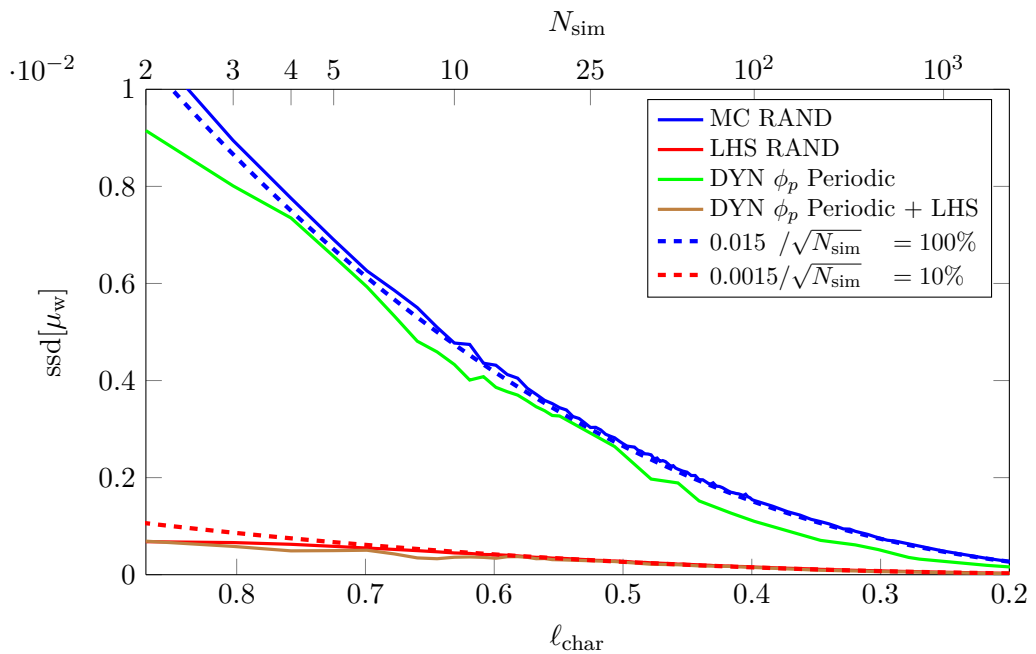


Figure 11.17: Sample standard deviation of the estimation of mean value of deflection w .

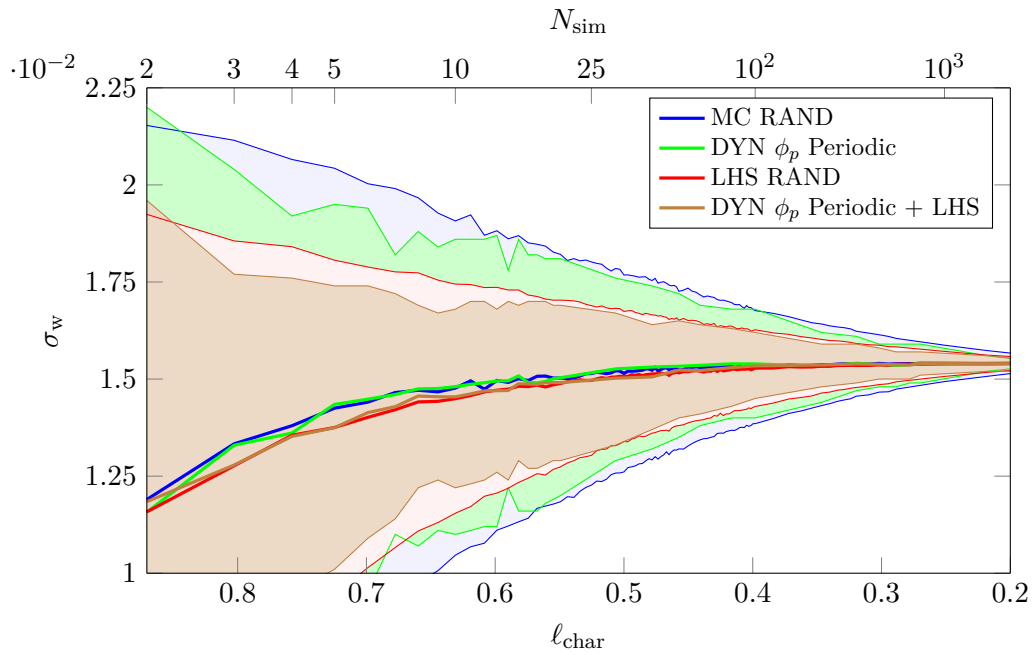


Figure 11.18: Estimated standard deviation of deflection w (ave, ave \pm ssd).

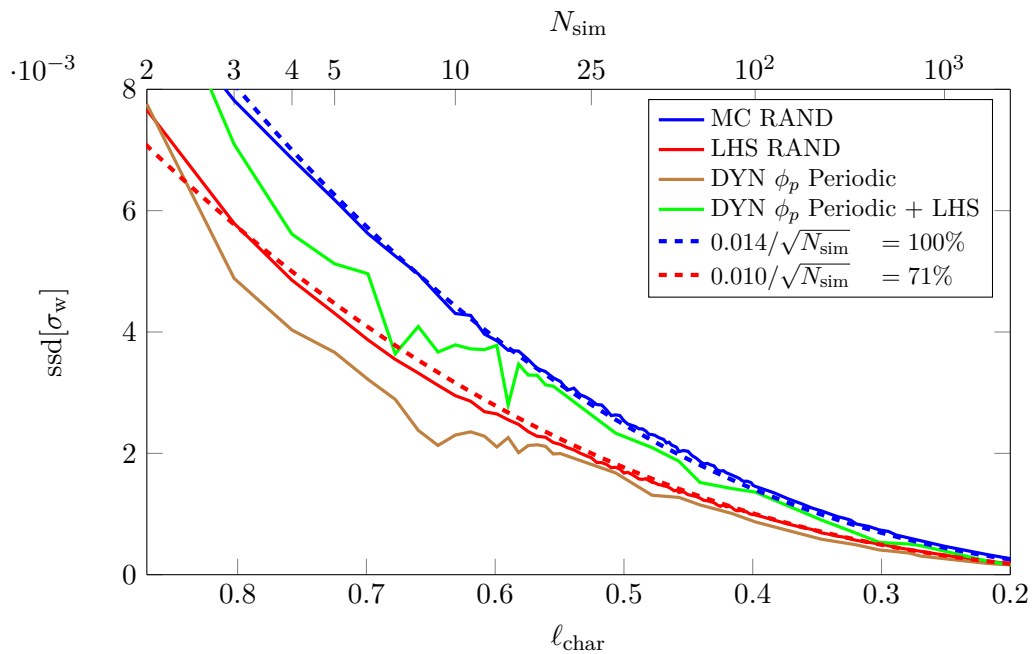


Figure 11.19: Sample standard deviation of the estimation of standard deviation of deflection w .

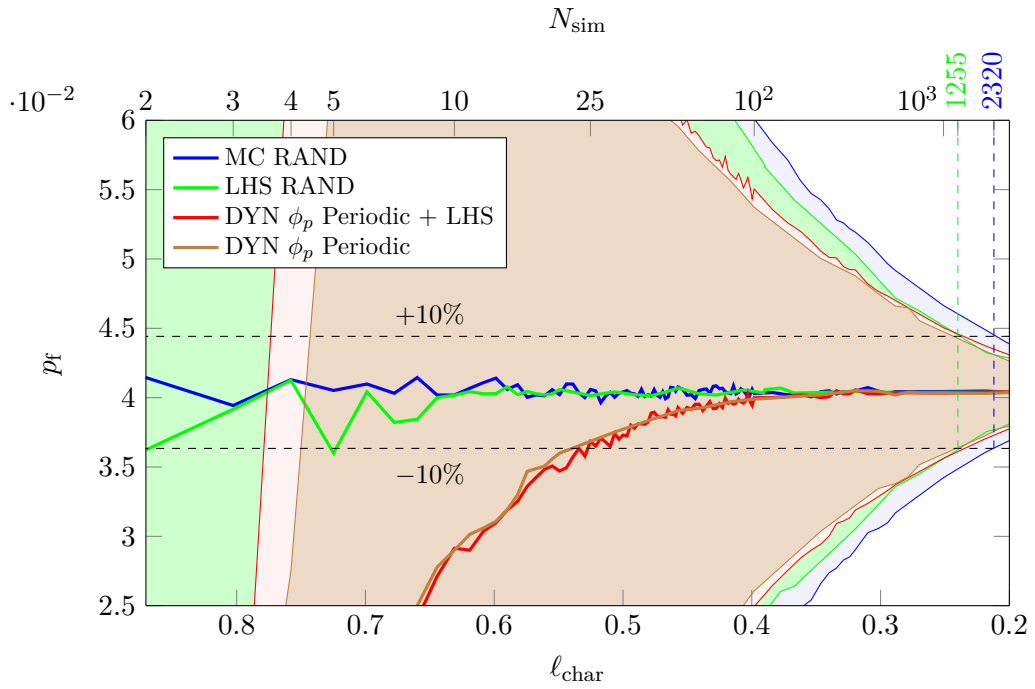


Figure 11.20: Estimated failure probability p_f (ave, ave \pm ssd).

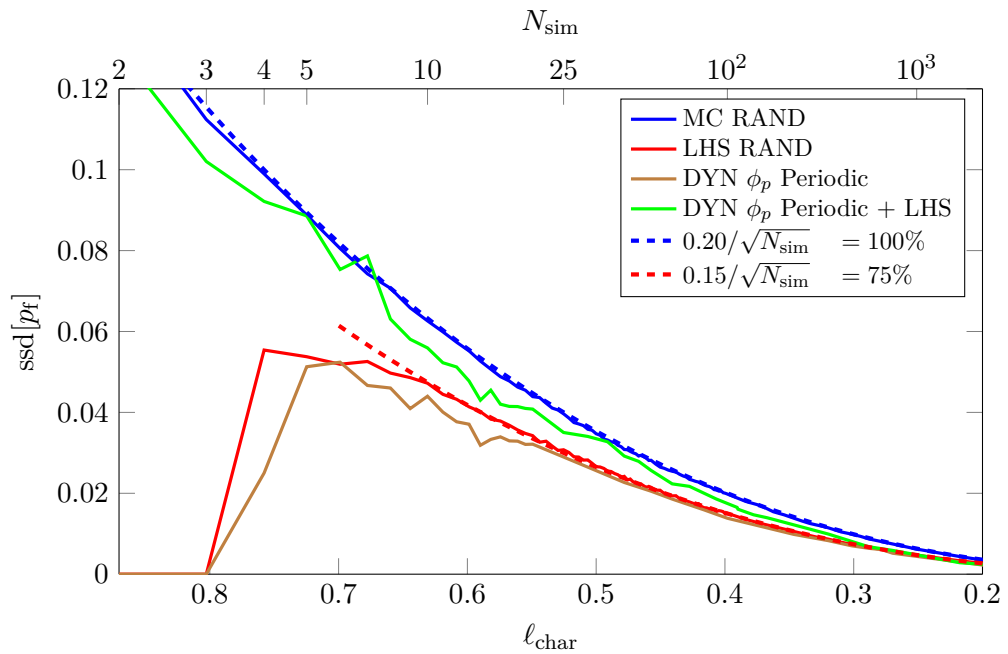


Figure 11.21: Sample standard deviation of estimation of failure probability p_f .

11.4 Engineering example I - A refined approach

It has been shown in the previous section 11.3 that for estimation of failure probability, the usage of statistical samples optimized to uniformly fill the entire design domain does not yield a significant variance reduction. Despite such samples perform very well for estimation of mean value, recall sections 11.1 and 11.2, the property of uniformity across the entire design domain does lead to efficient estimation of probability of failure that typically emerges within a minor subset of the domain. A practicing reliability engineer would therefore lean towards (i) a variance reduction technique such as Importance sampling (IS), see e.g. [103, 104], and (ii) would examine the possibilities of *reduction of the dimension* of the problem at hand. The following sections show how these steps allow to exploit the advantages of the dynamically optimized samples.

11.4.1 Dimension reduction

For a large portion of engineering problems, one can attempt to simplify the matter towards its essence by a conscious study of the system. In the case of the truss structure of interest, let us examine the five input random variables, see table 11.2, and their interaction in the model represented by Equation (11.18). It can be concluded that the log-normal random variables of E_1 and A_1 for horizontal members and E_2 and A_2 for the diagonal members are always featured together in a product. Each of these products of log-normally distributed random variables can substituted by a single random variable EA_{horiz} and EA_{diag} , respectively. That way, the problem of five-dimensional input vector would boil down to an entirely equivalent problem that has only three input random variables: the load P and two cross-sectional normal stiffnesses EA_{horiz} and EA_{diag} . The goal is to find the parameters of the new log-normal distributions of variables EA_{horiz} and EA_{diag} .

It can be shown that a product of two log-normal distributions E and A is, again, a log-normal distribution, here denoted as EA . The mean value, μ_{EA} , and standard deviation, σ_{EA} , of a product of two independent variables read:

$$\begin{aligned}\mu_{EA} &= \mu_E \mu_A, \\ \sigma_{EA} &= \sqrt{\mu_E^2 \sigma_A^2 + \mu_A^2 \sigma_E^2 + \sigma_E^2 \sigma_A^2}.\end{aligned}\tag{11.19}$$

That way, the properties of log-normal distributions of the substitute random variables EA_{horiz} and EA_{diag} can be obtained. Without any harm, the probabilistic truss model is now reduced to three random variables summarized in Table 11.3.

Random variable			Distribution	Mean value	c.o.v.
Cross-sectional stiffness	EA_{horiz}	[kN]	Log-normal	4.2×10^5	0.142
Cross-sectional stiffness	EA_{diag}	[kN]	Log-normal	2.1×10^5	0.142
Load	P	[kN]	Gumbel	50	0.15

Table 11.3: Three random variables of the reduced truss system.

The results of simulations before have been obtained using author's implemented generalized solver of truss models. For convenience in following usage in estimation of the mean

value, μ_w , and standard deviation, σ_w , the deflection (11.18) of the studied truss model can be simplified as follows:

$$g_1 = g_{w,5d}(\mathbf{X}) = P\left(\frac{552}{E_1 A_1} + \frac{50.912}{E_1 A_2}\right) \equiv g_{w,3d}(\mathbf{X}) = P\left(\frac{552}{EA_{\text{horiz}}} + \frac{50.912}{EA_{\text{diag}}}\right). \quad (11.20)$$

The *failure surface*, that forms the border between safe and failure regions, contains all points that fulfill the condition:

$$g_1 = 0.11. \quad (11.21)$$

For estimation of failure probability one can use the simple formulation of:

$$g_2 = 0.11 - g_1. \quad (11.22)$$

Further reduction of the dimension of the problem might be possible by executing *design point search*. The result is the position of the most probable point on the failure surface. Its position in standard Gaussian space reveals the sensitivities of individual random variables: the α -sensitivities. The analysis has been performed using the probabilistic software FReET [40]. The result of an α sensitivity analysis is a vector respective sensitivity coefficients, α_i , related with each input random variable. The results of sensitivity analysis of the problem at hand are shown in Table 11.4.

Random variable	α
Stiffness coef. EA_{horiz}	-0.539
Stiffness coef. EA_{diag}	-0.0888
Load P	0.838

Table 11.4: Results of sensitivity analysis of the truss model.

It can be concluded that the system, i.e. the deflection w , is least sensitive to the normal stiffness of diagonal members. Neglecting the random variable EA_{diag} (that is setting EA_{diag} as constant equal to $\mu_{EA_{\text{diag}}}$) would be therefore the candidate step for further reduction to two-dimensional problem. However, note that this decision would already corrupt the behavior of the model to some extent. Such a reduction is not of interest in this example and the system as shown in Table 11.3 is going to be used further.

11.4.2 Importance sampling approach

Importance sampling is a mathematical construct that aims to decrease the variance of estimation of failure probability by transforming positions of sampling points to occur more frequently in the region of failure. The idea behind Importance sampling is to use a *biased sampling distribution* of sampling points, otherwise pursued unbiased uniform distribution.

To conduct the process of Importance sampling (biased re-sampling of statistical samples) as done in this example, we use an isoprobabilistic transformation into Gaussian space. The first step is to transform the statistical sample into the Gaussian space of multivariate, standard normal distribution $\mathcal{N}(0, 1)$ by using its inverse cumulative distribution function

for each dimension (marginal). The next goal is to *bias* the mean value and/or standard deviation of the current standard normal distribution of sampling points to achieve more frequent presence of sampling points within the failure region. The question is now how to determine the characteristics of the biased sampling distribution $F^* = \mathcal{N}(\mu_{F^*}, \sigma_{F^*})$.

A suitable candidate for the mean value of the sampling distribution is the position of a *design point*, here denoted as D . The design point is defined to be a point on the failure surface possesses the highest probability $\prod_{v=1}^{N_{\text{var}}} f_v(x_v)$, see Figure 11.23. The coordinates of the design point need to be solved by an iterative computation, possible to execute e.g. by the FReET software. In the physical space of the design points of the 5d and the reduced 3d models are located on coordinates:

$$\begin{aligned} D_{5d}^{\text{phys}} &= \{P, E_1, A_1, E_2, A_2\} = \\ &= \left\{ 62.14 \times 10^3, 1.953 \times 10^{10}, 1.860 \times 10^{-3}, 2.066 \times 10^{10}, 9.840 \times 10^{-4} \right\}, \quad (11.23) \\ D_{3d}^{\text{phys}} &= \{P, EA_{\text{horiz}}, EA_{\text{diag}}\} = \left\{ 62.14 \times 10^3, 203.3 \times 10^6, 363.2 \times 10^6 \right\}. \end{aligned}$$

After transformation into the space of sampling probabilities, \mathbf{U} -space, the obtained coordinates of the design points of the 5d and 3d truss models read:

$$\begin{aligned} D_{5d}^u &= \{u_P, u_{E_1}, u_{A_1}, u_{E_2}, u_{A_2}\} = \{0.932, 0.249, 0.249, 0.456, 0.456\}, \\ D_{3d}^u &= \{u_P, u_{EA_1}, u_{EA_2}\} = \{0.932, 0.169, 0.437\}. \end{aligned} \quad (11.24)$$

Coordinates of the design point in the standard Gaussian space, that are considered to be the mean value of IS distribution, μ_{F^*} , then are:

$$\begin{aligned} D_{5d}^g &= \{g_P, g_{E_1}, g_{A_1}, g_{E_2}, g_{A_2}\} = \{1.491, -0.678, -0.678, -0.112, -0.112\}, \\ D_{3d}^g &= \{g_P, g_{EA_1}, g_{EA_2}\} = \{1.491, -0.959, -0.158\}, \end{aligned} \quad (11.25)$$

These coordinates of the design point show not only the α -sensitivity but can also be used for the FORM approximation of probability of failure, p_f :

$$\beta^{\text{HL}} = \sqrt{\sum_{v=1}^{N_{\text{var}}} g_v^2} = 1.78 \Rightarrow p_f^{\text{FORM}} = \Phi(\beta) = 0.040384. \quad (11.26)$$

The standard deviation of the sampling distribution, σ_{F^*} , is here kept unchanged, equal to one. The sampling distribution is hence defined, $F^* = \mathcal{N}(D^g, 1)$. Finally, to obtain the desired biased statistical sample in this case, it suffices to conduct translation of sampling points, i.e. to add the coordinates of the design point to the coordinates of each sampling point. Generally though, a transformation using inverse CDF of the sampling distribution $F^* = \mathcal{N}(\mu_{F^*}, \sigma_{F^*})$ is required.

After transformation from the Gaussian space back into the probability hypercube $[0, 1]^{N_{\text{var}}}$ using the cumulative distribution function of multivariate standard normal distribution, such a modified sample is ready to be used for execution of numerical experiments, see Figures 11.22 and 11.23. The process of Importance sampling as conducted for the reduced truss example is depicted in Figure 11.22 in the subspace of the reduced model that contains variables U_1 and U_2 . These dimensions of are considered to sample the variables of P and EA_{horiz} , respectively.

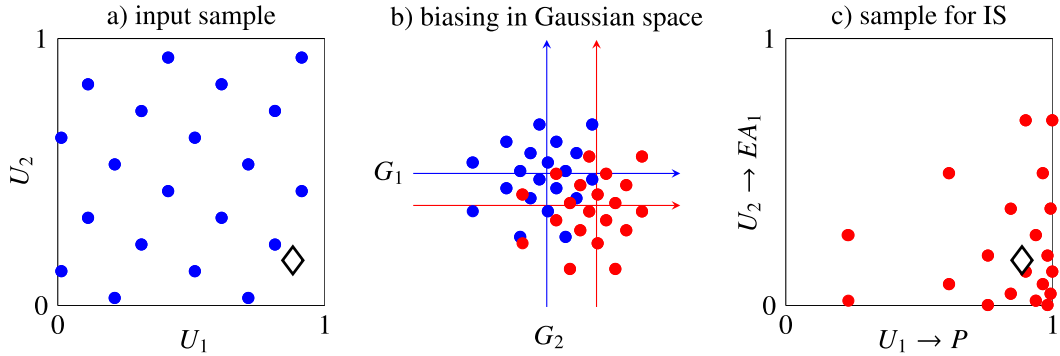


Figure 11.22: The process of transformation of optimized samples (blue) using the sampling density of Importance sampling method to achieve a sampling points concentrated in closer proximity of the design point. The position of the design point is highlighted.

The estimation of the failure probability using sampling with the original density $f_{\mathbf{X}}(\mathbf{x})$ reads:

$$p_f \approx \frac{1}{N_{\text{sim}}} \sum_{\text{sim}=1}^{N_{\text{sim}}} I[g(\mathbf{x}_{\text{sim}})], \quad \text{where } \mathbf{x}_{\text{sim}} = F_{\mathbf{X}}^{-1}(\mathbf{u}_{\text{sim}}), \quad (11.27)$$

that is by summing all the events of failure divided by the total number of simulations, N_{sim} , does no longer suffice. Instead, in the spirit of Importance sampling, the result attained in each sampling point is *weighed*:

$$p_f \approx \frac{1}{N_{\text{sim}}} \sum_{\text{sim}=1}^{N_{\text{sim}}} I[g(\mathbf{x}_{\text{sim}})] \frac{f_{\mathbf{X}}(\mathbf{x}_{\text{sim}})}{f_{\mathbf{X}}^*(\mathbf{x}_{\text{sim}})}, \quad \text{where } \mathbf{x}_{\text{sim}} = F_{\mathbf{X}}^{-1}(\mathbf{u}_{\text{sim}}). \quad (11.28)$$

The ratio of $f_{\mathbf{X}}(\mathbf{x}_i)/f_{\mathbf{X}}^*(\mathbf{x}_i)$ is the ratio of values of the original density function and the sampling density. Since the process is performed in the Gaussian space, both densities are computed there.

The benefit of using Importance sampling will be further demonstrated for the initial 5d truss model as well as for its reduced 3d variant.

11.4.3 Discussion of numerical results

By bringing the sampling patterns closer to the region of failure, the samples with optimized uniformity tend to retrieve their advantage in comparison with estimation using plain Monte Carlo or LHS samples. Indeed, this is also the case of the truss model at hand. The attained convergence of estimation of failure probability by using Importance sampling is provided in Figures 11.24 and 11.26 for the original 5d model and the reduced 3d model, respectively. Similarly to the preceding plots, the complementary Figures 11.25 and 11.27 compare the standard deviation of the estimation of failure probability.

The presented results convey several important notions (also recall Figure 11.20 for comparison). First of all, the biased mean value of estimation using samples of LHS type has been rectified closer to an unbiased estimation. This is due the fact that Importance sampling tends to remedy the lack of border coverage by LHS samples. Importance sampling

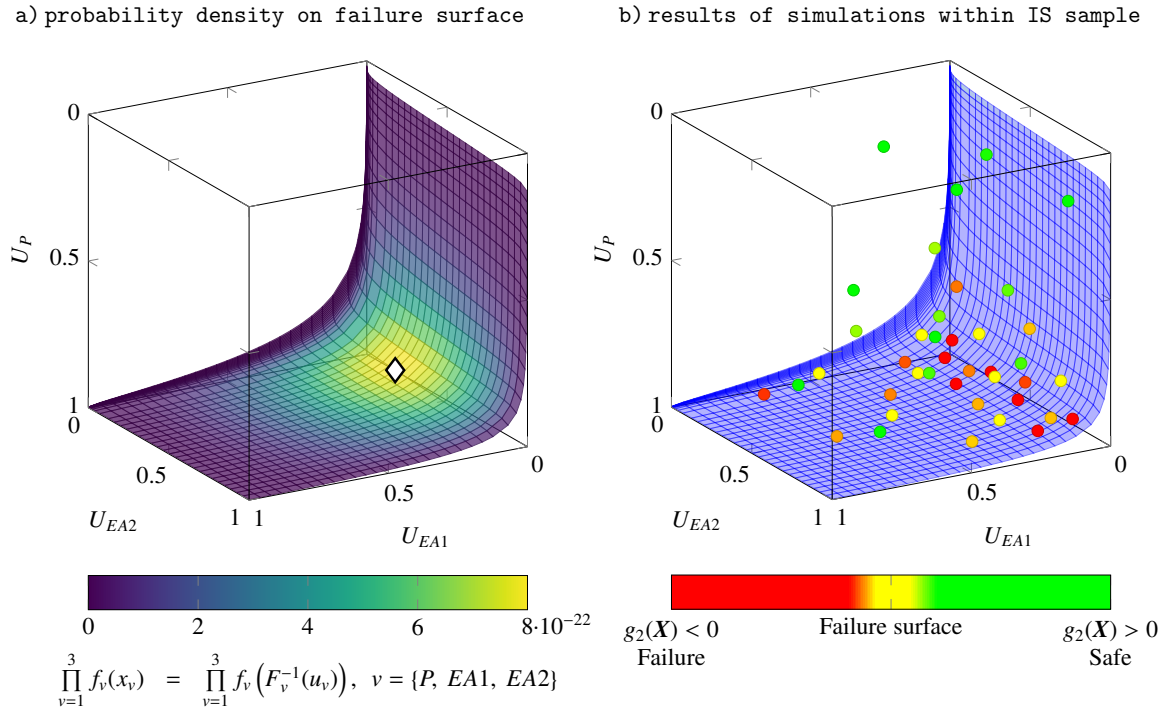


Figure 11.23: a) The probability on the failure surface of the reduced 3d model and the position of the design point in the probability space. b) Results of simulations in sampling points of the IS sample from Figure 11.22c. The color scale of points shows the value of $g_2(\mathbf{X})$ to represent the position of each sampling point with respect to the failure surface.

aims to relocate a larger portion (approximately 50%) of sampling points into the failure region. As long as a sampling point is located within the failure region, the estimation of failure probability is not sensitive to the distance of the point from failure surface, it gathers only the binary information about failure occurrence.

The LHS RAND sampling combined with IS further lowers the standard deviation of estimation of MC RAND. This is because not only the LHS RAND samples are no longer statistically biased as much (in terms of estimation of p_f), the individual LHS RAND samples also retain a higher degree of uniformity than entirely random Monte Carlo. That way, the probability of sampling about 50% of points in safe region and another 50% in failure is higher, therefore the standard deviation of the estimation is lower.

The effect of using Importance sampling combined with optimized statistical samples can be also seen in Figures 11.24 and 11.26. The standard deviation of estimation of failure probability is lowered by ca. 35% when using samples with optimized uniformity. Both the raw DYN ϕ_p Periodic samples as well as their latinized versions yield very similar results when used for Importance sampling. This is, again, due to the estimation of failure probability being insensitive to possible collapsibility and IS making up for the lack of border coverage by LHS.

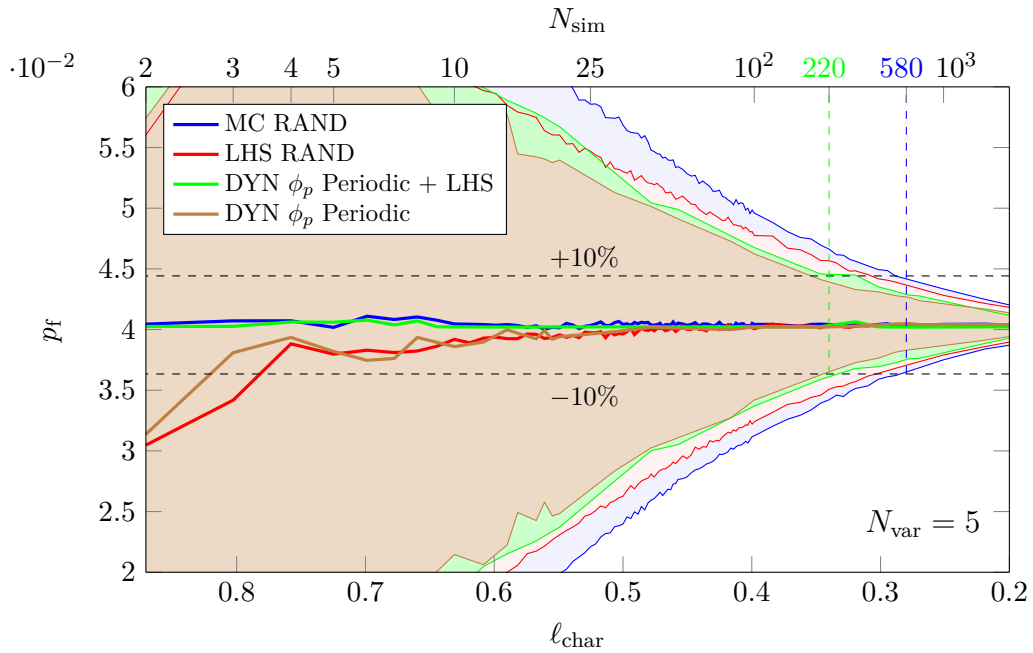


Figure 11.24: Estimated failure probability p_f using IS in the original 5d space (ave, ave \pm ssd).

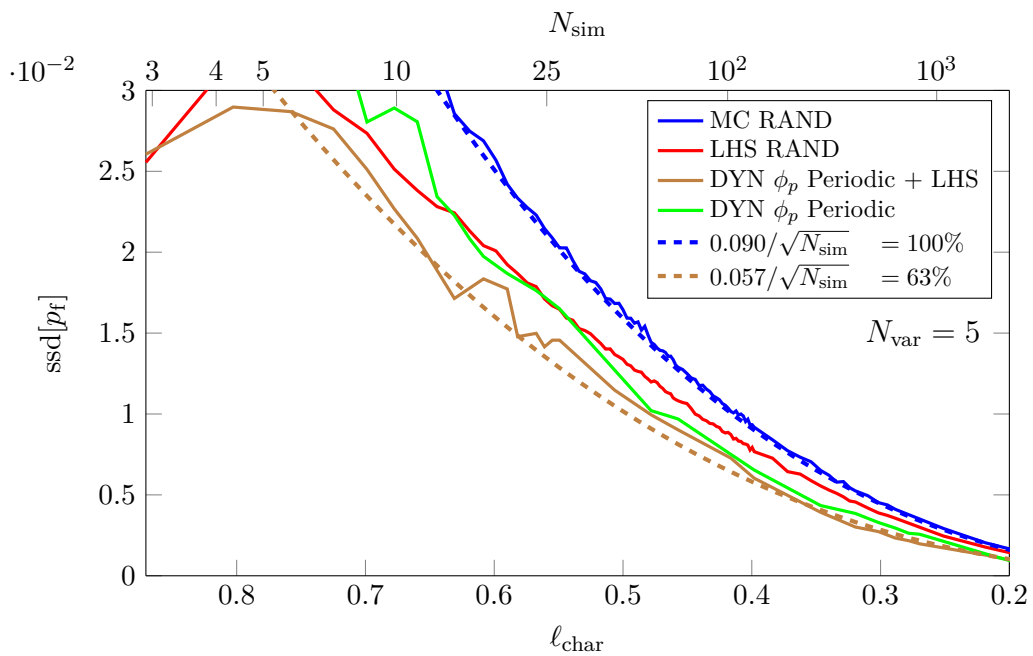


Figure 11.25: Sample standard deviation of estimation of failure probability p_f using IS in the original 5d space.

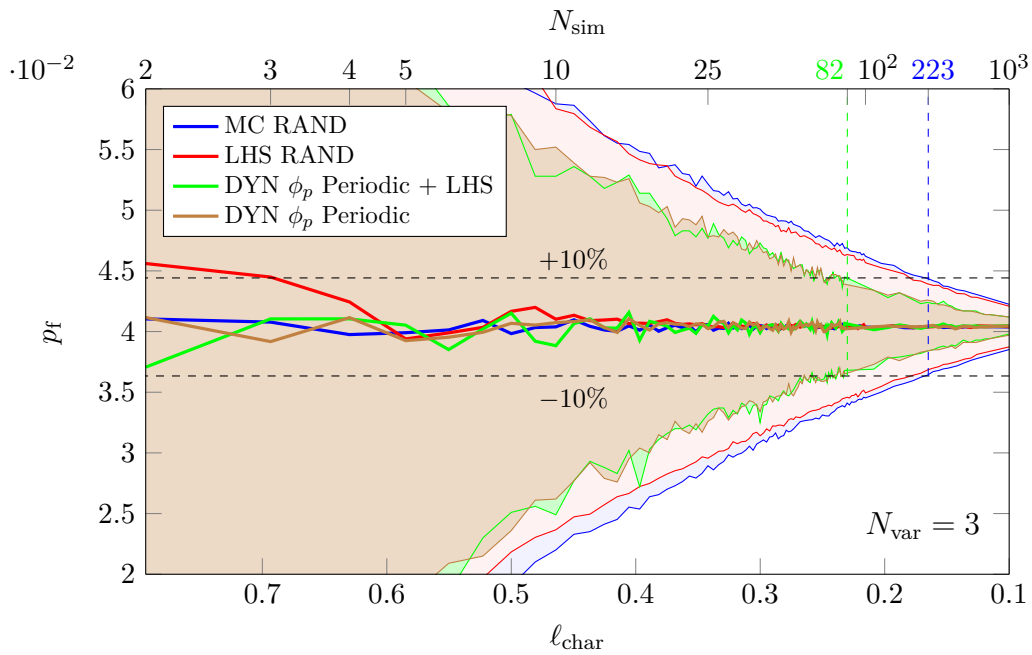


Figure 11.26: Estimated failure probability p_f using IS in the reduced 3d space (ave, ave \pm ssd).

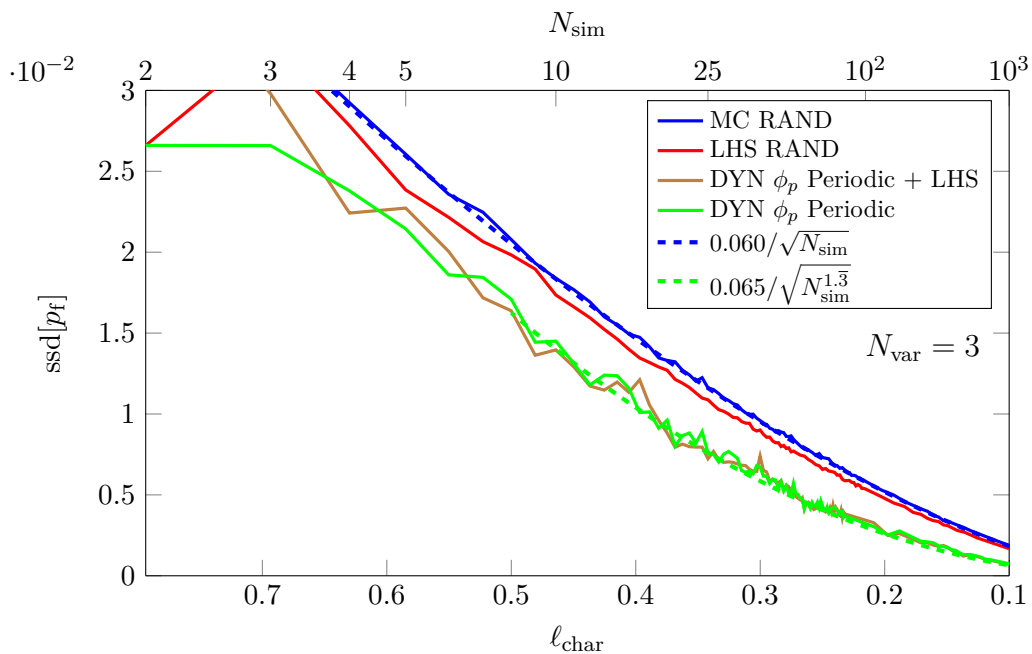


Figure 11.27: Sample standard deviation of estimation of failure probability p_f using IS in the reduced 3d space.

11.4.4 Comparison of presented approaches to estimation of p_f

As an ending note of the truss failure example, a comparison of behavior of each sampling method across all examined approaches of estimating failure probability the system is presented. For each individual sampling method, the convergence of estimation of failure probability is plotted as attained in the 5d space of the original problem and the benefit of using IS in 5d and the reduced 3d space.

Through the Figures 11.28, 11.29, 11.30, and 11.31, one can study the effect of reducing the dimension and IS for MC RAND, LHS RAND, and both versions of DYN ϕ_p samples. It is no longer desirable to plot the results in dependence on ℓ_{char} that represents the saturation of the domain with sampling points because the value of ℓ_{char} differs for each N_{var} , recall Equation (11.1). Instead, the following plots must use a universal measure that is the number of integration points (particles), N_{sim} , which in practical sense represents the number of numerical experiments required.

For each sampling method and for each estimation approach (5d, 5d+IS, 3d+IS) is highlighted the number of model evaluations required for obtaining an estimation with CoV lower than 10%.

By comparing results of MC RAND and DYN ϕ_p samples, it can be seen that only by using dynamically optimized samples in the original 5d problem, the required number of model evaluations decreases from ca 2300 to 1200 simulations. That means decreasing the required computing time by ca. 50%. Similar comparison of the LHS RAND and latinized DYN ϕ_p samples gives a decrease in required N_{sim} of about 15%. This is due the LHS samples being inherently resistant to uniformity optimization along borders (where the failure usually appears).

When considering the usage of Importance sampling, i.e. transforming the samples towards the design point, the variance of estimation of all sampling methods decreases. In the case of using IS in the original 5d space, usage of DYN ϕ_p samples instead of MC RAND samples lowers the number of required simulations from 580 to 220 simulations (about 62% decrease). If considering IS in the reduced 3d space, the 223 simulations required for MC RAND samples can be reduced to 82 simulations by using DYN ϕ_p samples (about 63% decrease).

The performance of the comparable LHS and latinized DYN ϕ_p samples for in IS becomes more differentiated. This is due the majority of sampling points transformed around the design point. Therefore, the uniformity of samples becomes relevant again. If using IS in the original 5d domain, the usage of latinized DYN ϕ_p samples requires about 183 model evaluations for reaching CoV of 10%. Compared to 370 model evaluations required by LHS RAND samples, using the optimized samples yields ca. 50% decrease of necessary computing effort. In the reduced 3d space, latinized DYN ϕ_p samples reduce the number of simulations required by LHS RAND from 180 to 75 model evaluations, i.e. by 58%.

It can be concluded that both variants of the dynamically optimized samples perform well also in this engineering example, yielding a rather significant decrease of required computing time. Moreover, the possibility of sample size extension one-by-one remains absolutely possible. This allows to incrementally include additional optimized sampling points if further decrease of estimation error is desired, see later in Chapter 12, Section 12.2.

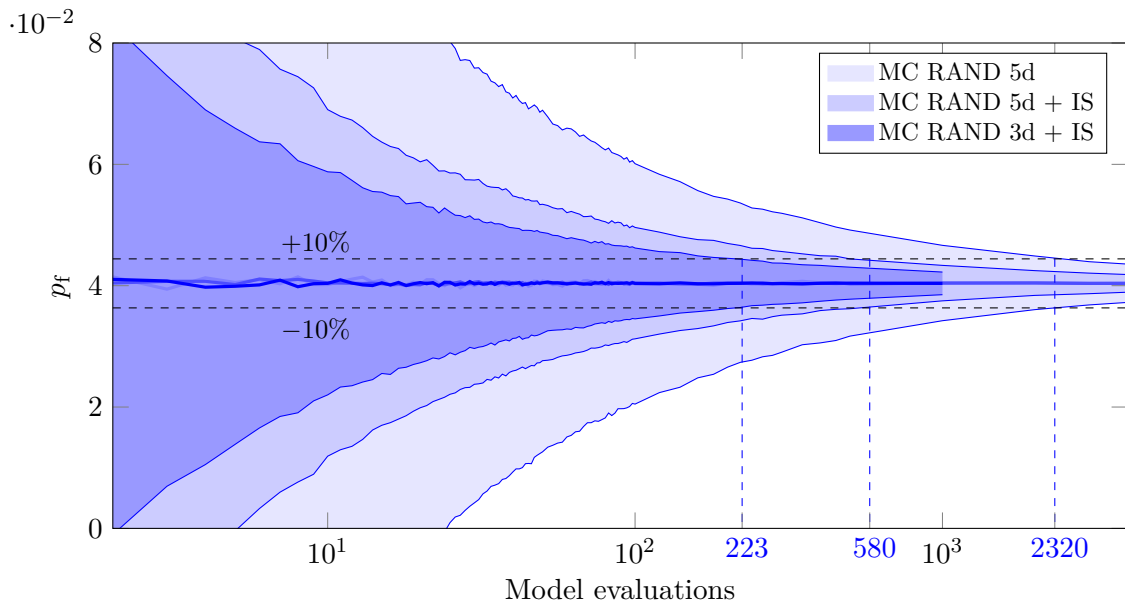


Figure 11.28: Monte Carlo approximation: comparison of convergence of estimated failure probability, p_f , in the 5d space of the original problem and the benefit of using IS in 5d and the reduced 3d space (ave, ave \pm ssd).

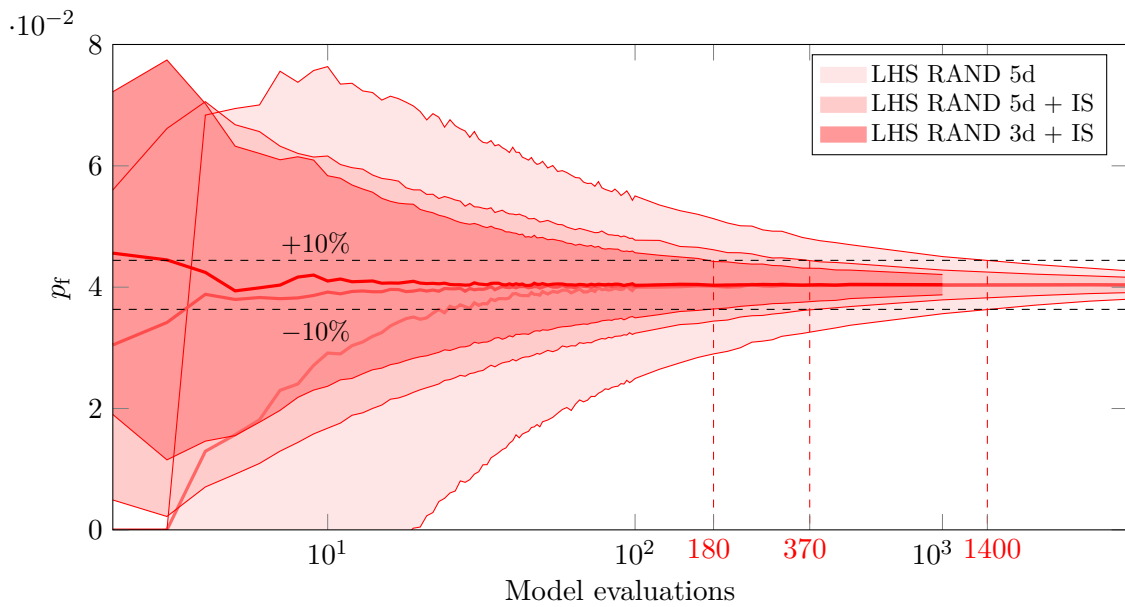


Figure 11.29: Random LHS approximation: comparison of convergence of estimated failure probability, p_f , in the 5d space of the original problem and the benefit of using IS in 5d and the reduced 3d space (ave, ave \pm ssd).

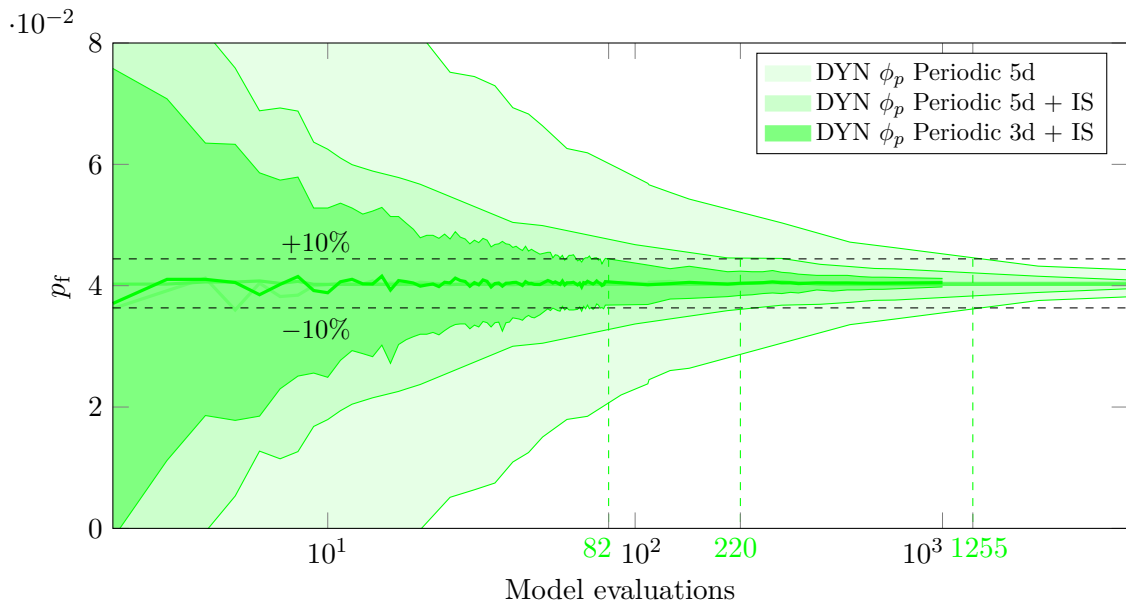


Figure 11.30: DYN ϕ_p Periodic samples: comparison of convergence of estimated failure probability, p_f , in the 5d space of the original problem and the benefit of using IS in 5d and the reduced 3d space (ave, ave \pm ssd).

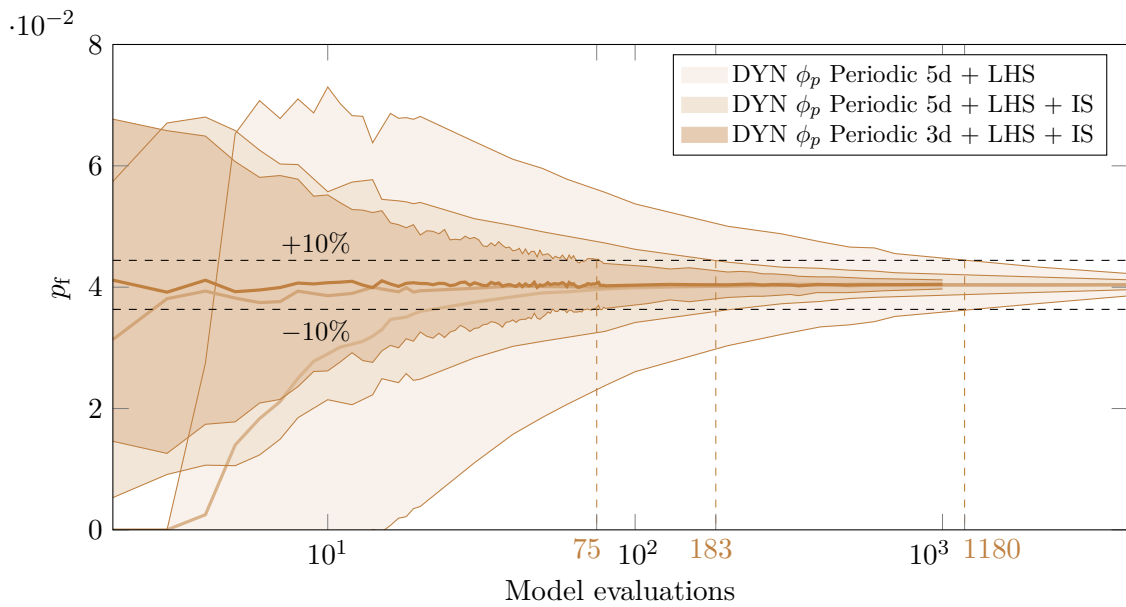


Figure 11.31: Latinized DYN ϕ_p Periodic samples: comparison of convergence of estimated failure probability, p_f , in the 5d space of the original problem and the benefit of using IS in 5d and the reduced 3d space (ave, ave \pm ssd).

11.5 Engineering example II - FEM model of fracture

As the last presented benchmark problem, a rather demanding numerical model of a three-point bending test of a notched concrete specimen has been selected. Such a problem represents a practical example of a *black box* system that is considered beyond any control of the engineer. Note that otherwise, the topic of nonlinear fracture mechanics is entirely orthogonal to this work.

The selected example has been also used in [37] as a test problem for samples optimized by LHS-switching in pursuit to minimize sample correlation. Such concrete specimens have been also tested experimentally in [105].

The numerical simulations of a three-point bending test setups are conducted with the help of the OOFEM software [106–108]. The failure due to cracking is simulated by nonlinear finite element analysis using an isotropic damage model for tensile failure (Idm1) that is implemented within the OOFEM software. The isotropic damage model considers the stiffness degradation to be isotropic, that is the stiffness decrease does not prefer any direction and takes place independently on loading direction. The model is local, using the damage law with exponential softening adjusted according to the element size (crack-band approach). The setup of the numerical experiment is illustrated in Fig. 11.32.

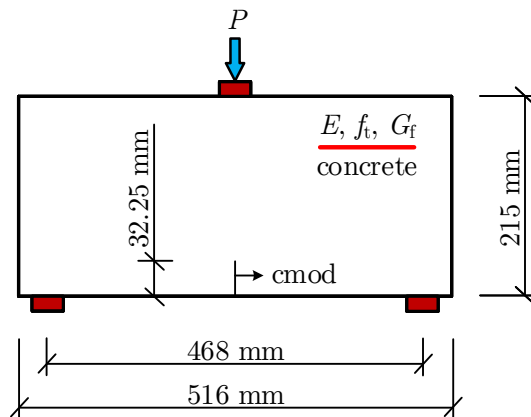


Figure 11.32: An illustration of experimental setup of three-point bending of a notched specimen.

Three material properties are considered as independent random variables: tensile strength, f_t , Young's modulus E and fracture energy G_f . Their properties are provided in Table 11.5. The analysis aims at estimating the 5th percentile of the maximum loading force, $P_{\max, 0.05}$.

Random variable	Distribution	Mean value	CoV
Young's modulus E [GPa]	Normal	36.5	0.15
Tensile strength f_t [MPa]	Normal	4.38	0.2
Fracture energy G_f [N/m]	Normal	60	0.3

Table 11.5: Random variables of FEM model example and their properties.

As the final example of the work, the FEM model aims to represent a real system with large computational demands. The execution of a single simulation of the FEM model takes roughly about 1 hour of CPU computing time depending on the particular values of material properties. Computing capabilities of a practicing engineer are considered to be limited and/or certainly a subject of minimization.

It is therefore naturally required to use a rather low number of sampling points (executed simulations). Hence, the benchmark is here defined as follows: to estimate the 5th percentile of maximum loading force, $P_{\max, 0.05}$, with using a single sample that contains 40 sampling points ($N_{\text{sim}} = 40$, $N_{\text{var}} = 3$). This can be considered a relevant practical scenario where one desires to obtain the most accurate result at restricted computational or other resources, here set to roughly 40 hours of computing time.

To provide a statistical evaluation of such an estimation, simulations for each sampling method are performed across 30 its samples ($N_{\text{run}} = 30$). That results in $40 \times 30 = 1200$ simulations needed for each of the four sampling methods that will be compared. In total, this means performing about 4800 hours (≈ 200 days) worth of CPU computation for purposes of this work. In this matter, the author wishes to thank Dr. Václav Sadílek for his support that made it possible to execute such a heavy computation and to collect back the results.

11.5.1 Discussion of numerical results

The estimation of the mean value of the 5th percentile of the maximum loading force, $P_{\max, 0.05}$, has been conducted by utilizing the simulation results through two different approaches:

- the first approach to estimate $P_{\max, 0.05}$ uses only the raw results. For each sampling method, the 40 obtained values of maximum force, $P_{\max, i}$, have been sorted and the second lowest force of all ($2/40 = 0.05$) is declared to be the estimation of $P_{\max, 0.05}$. Similarly, as a reference value of the unknown exact solution of $P_{\max, 0.05}$ was chosen the 60th lowest value of the 1200 runs of MC RAND samples (denoted as $P_{\max, 0.05}^{\text{MC}, 60\text{th}}$). Such an approach is considered to be a more genuine estimation than the latter,
- secondly, for the estimation of $P_{\max, 0.05}$, the 40 values of maximum force, $P_{\max, i}$, obtained from each sampling method have been fitted using the maximum likelihood method with density functions of two selected distributions: (i) the normal distribution and (ii) the 2-parametric Weibull minimum distribution. After that, as the approximation of the value of 5th percentile, $P_{\max, 0.05}$, is used the 5th percentile of the fitted distributions. As a reference value of the unknown exact solution is used the 5th percentile of the respective distributions fitted to the results of the 1200 MC RAND simulations ($P_{\max, 0.05}^{\text{N}, 0.05}$ and $P_{\max, 0.05}^{\text{W}, 0.05}$, respectively).

Before comparing the performance of sampling methods in estimating $P_{\max, 0.05}$, let us examine the chosen approaches to approximating the reference solution. Figure 11.33 shows a histogram of values of maximum loading force, $P_{\max, i}$, obtained from all 1200 Monte Carlo simulations. The 60th lowest attained value of $P_{\max, 0.05}$ that is used as an approximation of the reference solution from the raw data, yields a force of $P_{\max, 0.05}^{\text{MC}, 60\text{th}} = 3325$ N.

One can see that the histogram of peak loading forces exhibits a certain degree of negative skewness. Therefore, the fitted normal distribution (with zero skewness) will be probably not able to capture its left tail quite accurately, resulting in value of $P_{\max, 0.05}^{\text{N}, 0.05} = 3361$ N. Because

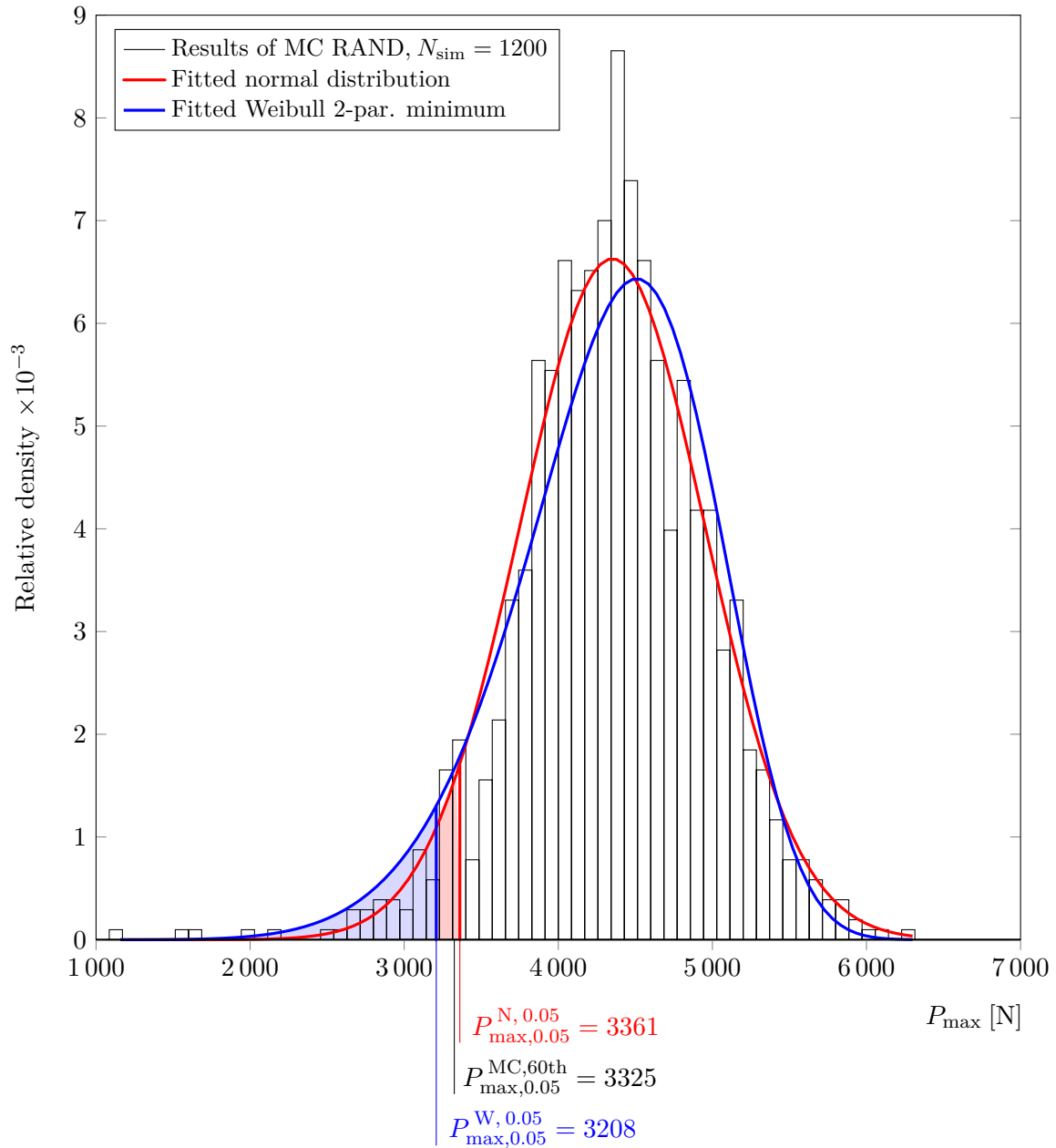


Figure 11.33: Histogram of results of 1200 Monte Carlo simulations that have been used to obtain reference solutions of $P_{\text{max},0.05}$.

the left tail of the histogram is especially important for estimating the 5th percentile, the 2-parametric Weibull minimum distribution has been also fitted as a possible candidate distribution with non-zero skewness. The fitted distribution with the value of $P_{\max, 0.05}^{\text{W}, 0.05} = 3208 \text{ N}$, however, seems overly left-skewed (skewness of -0.57) compared to the skewness of -0.33 estimated from the set of 1200 MC results.

By studying the goodness-of-fit of the fitted distributions, one can not conclude that either of the fitted distributions suits the histogram perfectly. Therefore, the further results related to both fitted distributions shall be rather understood as complementary to the estimation using raw data from numerical experiments. Moreover, the values of $P_{\max, 0.05}^{\text{N}, 0.05}$ and $P_{\max, 0.05}^{\text{W}, 0.05}$ surround the approximation from raw data, $P_{\max, 0.05}^{\text{MC}, 60\text{th}}$. Hence, it is considered as the closest, probably least biased approximation of the 5th percentile of maximum loading force, $P_{\max, 0.05}$.

The results of estimation of the 5th percentile of maximum loading force, $P_{\max, 0.05}$, using (i) second lowest force of each sample ($N_{\text{sim}} = 40$) are presented in Figure 11.34. The results of the estimation using (ii) 5th percentile of normal and Weibull distributions fitted to results of each sample are shown separately in Figure 11.35.

The mean value of each estimation is provided with its corresponding value of coefficient of variation (CoV). Despite these statistics are based only on 30 runs (30 different point samples), they provide at least illustrative representation of performance of each sampling method.

In the obtained results, one can compare the optimized samples to their relevant counterparts. That is to compare MC RAND samples to DYN ϕ_p Periodic, and LHS RAND samples to the latinized versions of DYN ϕ_p Periodic samples. In all instances, the dynamically optimized samples yield a reduction in variance of estimation, lowering CoV by 2% up to 30% in comparison to random MC or LHS samples. Also, on average, the 5th percentile estimated by dynamically optimized samples are observed to be slightly closer to their respective reference solution. However, given the standard deviation of each estimation, it is not enough for making a significant conclusion.

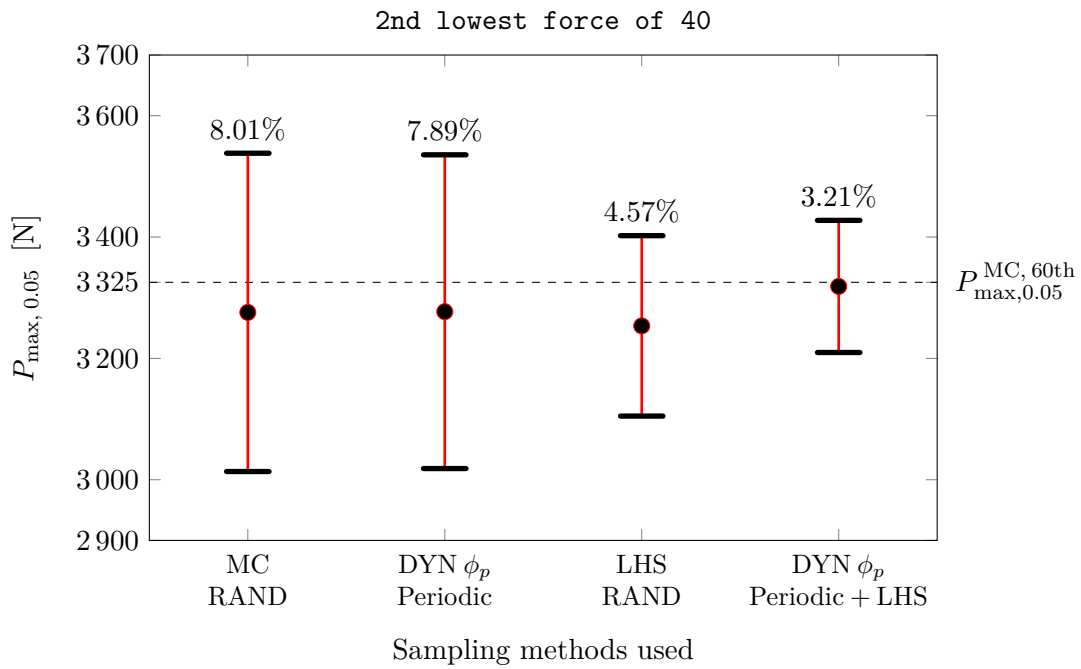


Figure 11.34: Estimation of the 5th percentile of the peak loading force, $P_{\max, 0.05}$, using the second lowest value of each 40 simulations (ave, ave \pm ssd).

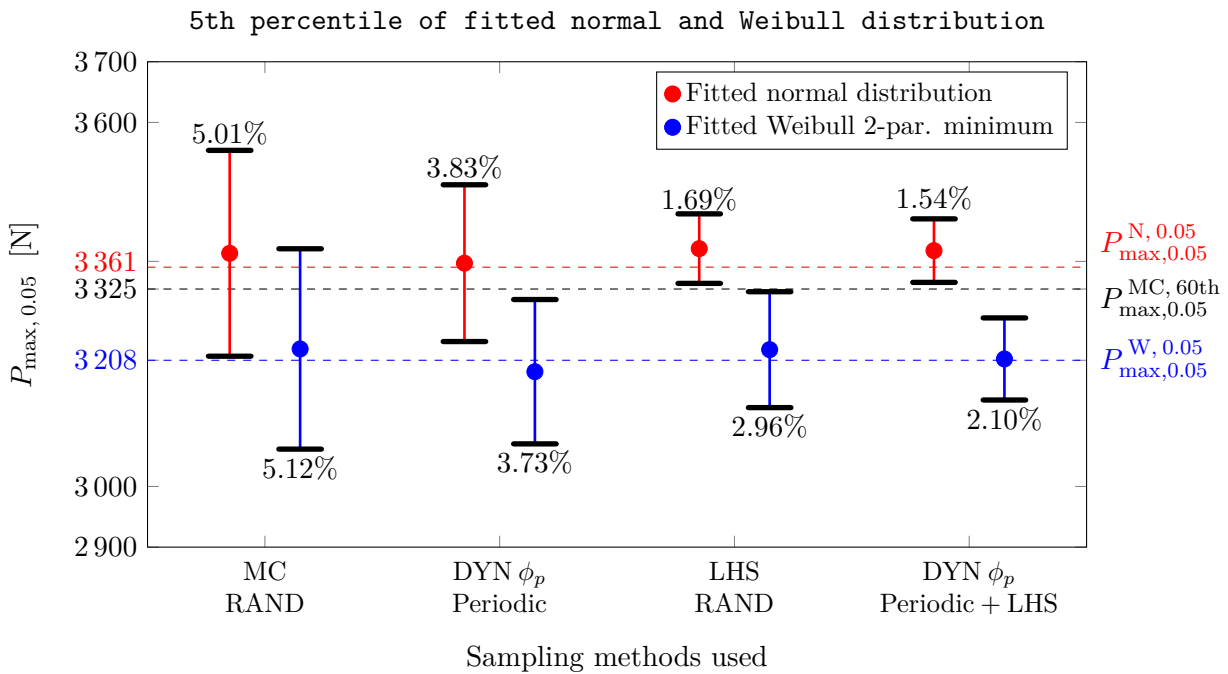


Figure 11.35: Estimation of the 5th percentile of the peak loading force, $P_{\max, 0.05}$, using 5th percentile of fitted normal and Weibull minimum distributions (ave, ave \pm ssd).

Chapter 12

Related topics

This closing chapter aims to shortly discuss selected secondary topics that have been encountered during the development of this work. The tendencies towards an efficient parallel solution of these tasks are a natural result of the gained knowledge and experience of the author concerning the CUDA platform.

12.1 Fast parallelized approximation of Voronoï diagram

The Voronoï diagram is a classical construct of computational geometry. Consider a domain of a general dimension containing N points (in this work, let us continue with the nomenclature of N_{var} and N_{sim}). The task of the Voronoï tessellation is to divide the given space into N_{sim} convex polyhedrons of dimension N_{var} that encompass regions where all points are closer to the point N_i than to any other of N_{sim} *generating* points.

The construction of a Voronoï diagram, see [109, 110], associated with a given set of points has become a classical task of computational geometry utilized in dozens of research and engineering fields. The geometric sense of Voronoï diagrams is used for solving tasks like searching for the nearest neighbor, the largest empty circle or estimating the roundness of an object. Technical applications include e.g. construction of meshes in bordered or periodically repeated domains, see [111] and [112]. In the field of Design of Experiments, which is of interest here, a Voronoï diagram associated with a given point set (point sample) is one of possible geometric constructions used for evaluation of uniformity of the point sample and possibly for further optimization of the point layout, see [20]. Additionally, it has been also proposed to use the volume of each cell in the Voronoï diagram as the weight of the corresponding integration point as used in the weighted numerical integration of Monte Carlo type [113].

The construction of the exact Voronoï diagram is a demanding task, the result of which are convex polyhedrons in the number equal to the number of points in the point sample, further denoted as N_{sim} . In the worst cases, the solution complexity approaches $\mathcal{O}(N_{\text{sim}}^2)$, see e.g. the QuickHull algorithm [114]. Moreover, the computational demands crucially grow with the dimension of the design domain, further denoted as N_{var} . On top of these computations, the eventual enumeration of volumes of all N_{var} -dimensional cells, see [115], has to be executed.

The presented contribution proposes a fast parallelized implementation of approximation

of volumes of the Voronoï cells without the need of constructing the Voronoï diagram itself. This approach turns out to be useful in cases where the actual shape of the diagram is not of interest but the resulting cell volumes have to be perpetually computed for optimization purposes.

The essence of the solution method is based on dividing the entire design space into well distributed sub-regions. As further used for demonstration, division using a regular orthogonal grid is one of possible options. Then, center of gravity of each sub-region is further considered to be a node within a regular orthogonal grid used in the approximation process, see e.g. [45]. All of N_{nds} created nodes are defined with their spatial coordinates. From now, we follow the definition of the Voronoï diagram. Position of each node \mathbf{u} can be compared with position of each point \mathbf{x}_i from the N_{sim} input points of the Voronoï diagram. Their mutual Euclidean distance $d_i(\mathbf{u})$ is computed and the particular node \mathbf{u} is deemed to belong to the Voronoï cell V_i associated with its corresponding sampling point \mathbf{x}_i if the following condition is met:

$$V_i = \left\{ \mathbf{u} \in \mathbb{R}^{N_{\text{var}}} \mid \forall j \neq i : d_i(\mathbf{u}) \leq d_j(\mathbf{u}) \right\}. \quad (12.1)$$

The actual approximated volume V_i of each Voronoï cell is then a result of a simple summation of known volumes of all respective sub-regions deemed to belong to each cell i :

$$V_i \approx \sum_{n=1}^{N_{\text{nds}}} V_{n,i}. \quad (12.2)$$

It will be shown later that with gradually finer mesh of approximated nodes, see Figure 12.1, the volumes of all Voronoï cells tend to converge to the exact solution.

As it might be apparent from the already discussed, the described solution is highly suitable for a massively parallel execution. The presented concept of parallel execution is based on author's experience achieved throughout this work. The actual implementation approach proposes to understand the nodes of the mesh as target points which interact with the source points which are here represented by the N_{sim} input points of the approximated Voronoï diagram. Descriptions (N_{var} coordinates) of all nodes are loaded into the GPU's shared memory which resides on the graphic chip. Then, tiles of source points are consecutively loaded into shared memory and a simple task of comparing mutual distances of target and source points is executed. For each target point, index of its closest source point is stored. The result of such an algorithm is an array of N_{nds} nodes of the mesh and an integer index of their respective closest point from the N_{sim} points of the approximated Voronoï diagram. The remaining task is a parallel reduction of this array of N_{nds} integers into an array of N_{sim} frequencies belonging to each Voronoï cell. Assuming a constant known volume of all sub-regions in the mesh, these frequencies can be interpreted as the cell volumes V_i , see equation 12.2, after being multiplied by the sub-region volume.

Similarly, coordinates of centers of gravity of Voronoï cells can be rather efficiently approximated. In a bordered design domain, the coordinate in the v th dimension of a center of gravity of the i th Voronoï cell V_i is simply an average of coordinates of nodes belonging to the cell V_i :

$$\text{cg}_{i,v} = \frac{1}{N_{V_i}} \sum_{j=1}^{N_{V_i}} x_{j,v}. \quad (12.3)$$

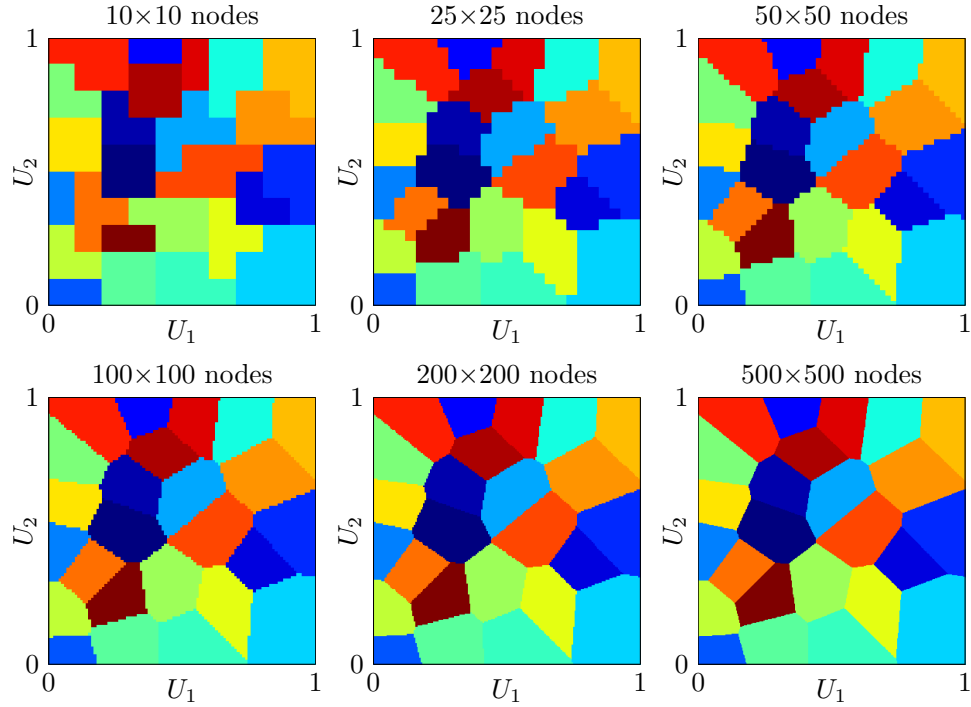


Figure 12.1: Illustration of convergence of normalized volumes of 25 Voronoï cells.

In the case of a periodically repeated design domain, such plain averaging is not possible. The approximated coordinates of centers of gravity need to respect the boundary conditions during the computation. The $(j + 1)$ -th step of averaging must be conducted as follows:

$$\begin{aligned}
 &\text{if } (|(x_{j,v} + 1) - \text{cg}_{i,v}(j)| < 0.5) && x_{j,v} + = 1; \\
 &\text{else if } (|(x_{j,v} - 1) - \text{cg}_{i,v}(j)| < 0.5) && x_{j,v} - = 1; \\
 &\text{cg}_{i,v}(j + 1) = \frac{1}{j + 1} \left(\text{cg}_{i,v}(j) \cdot j + x_{j,v} \right).
 \end{aligned} \tag{12.4}$$

The following content briefly discusses the computational complexity of the solution in dependence on the number of Voronoï cells (or N_{sim}), the dimensionality of the design domain N_{var} and the size of the orthogonal mesh used. The graph in Figure 12.2 plots the dependency of the execution time in milliseconds in dependence on the number of Voronoï cells. The plots are provided for several dimensions from $N_{\text{var}}=2$ up to 20 dimensions. These measurements were conducted for various mesh sizes with number of nodes from 10^3 to 10^6 . From the benchmark, a linear dependence on the number of Voronoï cells is apparent. Also, weak dependence on the dimension of the design domain is observed. This conclusion is of high importance when compared to the QuickHull algorithm which exhibits a steep performance drop when rising the dimension, N_{var} .

Due to its promising performance, the presented parallel solution seems highly suitable for the repetitive usage in the field of sample optimization or weighted Monte Carlo integration. The advantage of the observed linear dependency on the number of Voronoï cells can be explained by an insufficient workload for the GPU that results from the nature of the

algorithm: the crucial extent of the task solved is the number of nodes N_{nds} that is being parallelized. Executed consecutively are the tiles of the N_{sim} points of the Voronoi diagram, number of which is clearly orders of magnitude lower than N_{nds} .

The graph in Figure 12.2b shows the convergence of the solution of approximation of volumes of a Voronoi diagram containing 25 cells ($N_{\text{sim}}=25$) in a two-dimensional design space ($N_{\text{var}}=2$). Volumes V_i of Voronoi cells are normalized to their corresponding exact volumes computed in advance. With gradually finer mesh, it can be seen that all the particular volumes of the approximated Voronoi diagram rather quickly converge to the exact solution. This is also due to the boundaries between the Voronoi cells being linear functions. Moreover, it is shown that the mean value of normalized cell volumes is unbiased. The resulting approximated Voronoi diagrams from computations in Figure 12.2b are the ones shown in Figure 12.1.

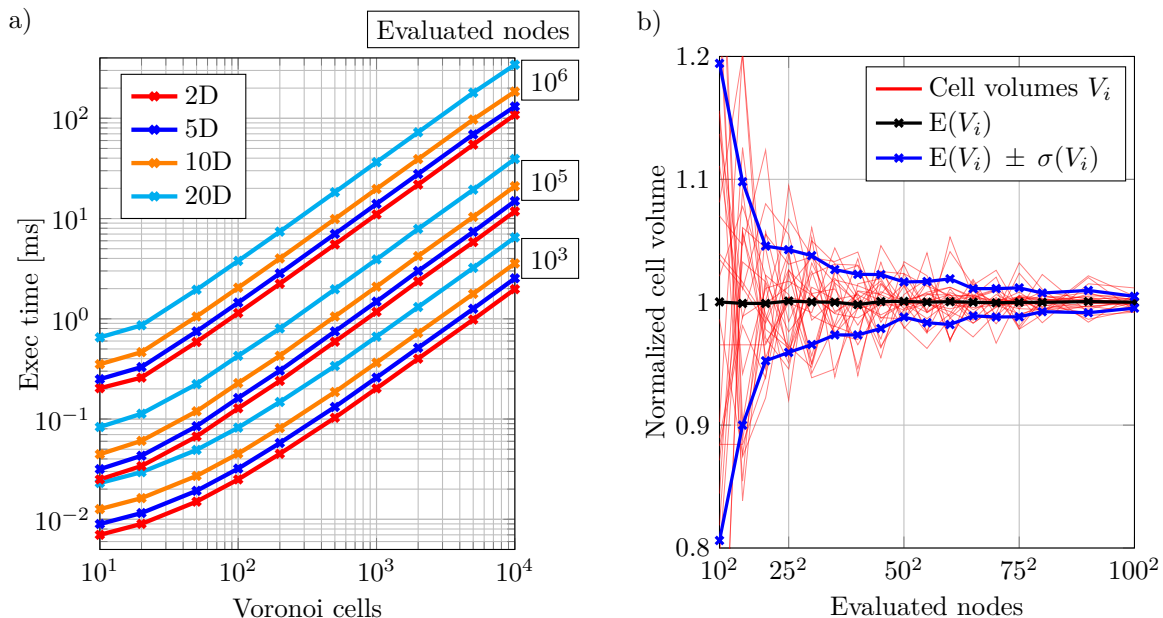


Figure 12.2: a) Growth of execution time in dependence on number of Voronoi cells and number of evaluated nodes used for approximation. b) Convergence of normalized volumes of Voronoi cells to the exact solution.

12.2 Sample size extension

One of advantages of the developed dynamical particle system that serves for optimization of statistical samples is also the possibility of *sample size extension*. In practice, this is the case when, after finishing the experiments specified in a prepared sample, a decision is made to conduct additional realizations on top of the already done. Besides the requirement for a maximum amount of new information from these additional experiments, it is natural to demand these additional experiments (sampling points) to respect the positions of the already conducted experiments.

Such requirements for new sampling points lead to the need of placing each new point into the largest unoccupied region, i.e. the *largest empty hyper-circle*. Such a problem can be solved by constructing a hyper-dimensional Delaunay triangulation [116]. The task is then to find the largest hyper-circle that is inscribed between any $(N_{\text{var}}+1)$ points. Its center can be roughly considered to be the center of the empty region, i.e. to be the position of the new integration point. The complexity of such a task however quickly diverges beyond practical possibilities for it is needed to check $\binom{N_{\text{sim}}}{N_{\text{var}}+1}$ of such possible N_{var} hyper-circles.

However, the research conducted due to this work has shown that the task of searching for the largest empty area can be solved without employing extensive computational geometry. One can instead study the definition of the potential of the dynamical particle system. More precisely, one can understand the largest empty area as a site where the additional particle (integration point) would experience the lowest repelling forces from other particles, i.e. it would experience the lowest radial stress. If knowing the distribution of potential energy across the design domain, one can easily identify such a site.

Building on what has been already developed in this work, the distribution of potential energy can be rather efficiently obtained. The algorithm for fast approximation of Voronoi diagram can be fairly simply modified. Instead of finding the closest particle to each node in the underlying mesh, each node now acts as a hypothetical particle for which all the repulsive forces from “real” (constrained) particles are summed in absolute values. Such a result for a system of 23 particles is shown in Figure 12.3 top left. The darkest blue color represents the area with least potential energy accumulated. The node with the lowest value is then considered as the position of the added integration point.

The procedure of such a consecutive addition of sampling points in periodic space is demonstrated in Figure 12.3 top row. In the bottom row, the respective Voronoi diagrams of each sample are provided. The presented algorithm of sample extension one-by-one is clearly superior e.g. to the sample extension of LHS samples in [37].

If the grid of nodes for evaluation of potential energy is fine enough, the node with the lowest value can be directly deemed to be the the new integration point. If there is a request to find the coordinates of the added point more precisely, the dynamical system naturally allows to constrain the movement of particles that represent already executed experiments and to let the new particle to find its equilibrium position. The study of properties of samples optimized by gradual sample extension is an interesting topic on its own and is not meant to be a part of this work.

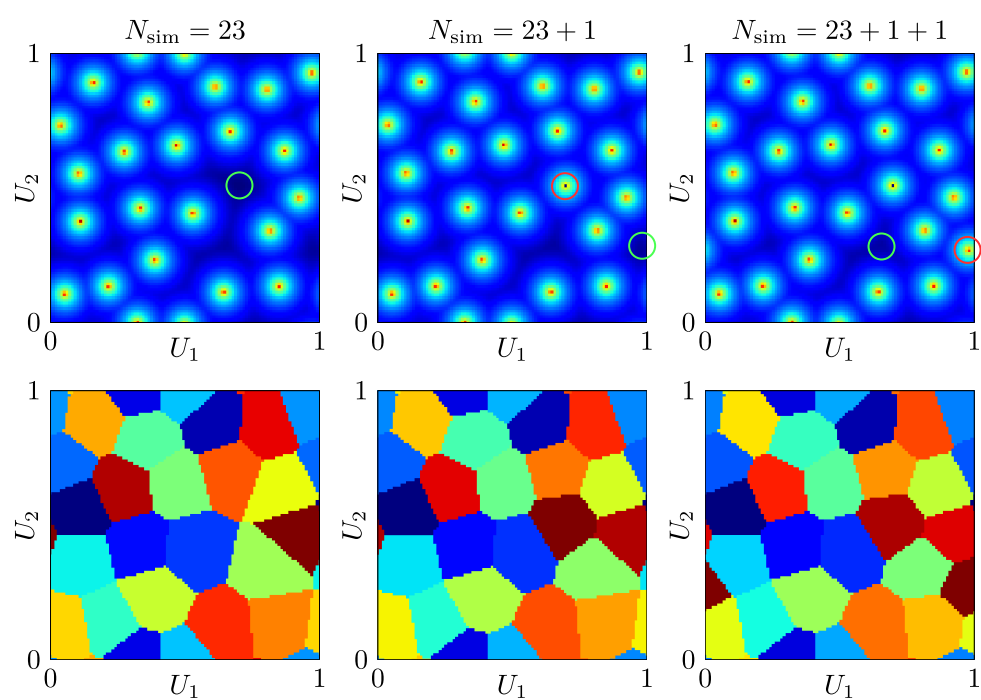


Figure 12.3: Identifying the largest empty area by evaluating the distribution of potential energy.

Chapter 13

Concluding remarks

The presented doctoral thesis was devoted to development of a new efficient tool for optimization of point samples that are spatially and statistically uniform. The primary use-case of these point sets is intended to be the usage as optimized sets of integration points in statistical analyses of computer models using Monte Carlo type integration. Due to formerly unseen computing power of contemporary hardware, feasible numerical simulations of vast and detailed models are becoming increasingly important, aiming to complement or even replace expensive physical experiments. Analogically to evaluating a function in conveniently distributed integration points, it is highly desirable to execute realizations of numerical experiments in configurations of input random variables that maximize the amount of new information gained from each simulation.

The first task of the work has been a survey of literature concerning properties of existing statistical sampling methods a criteria used for evaluation and/or optimization of uniformity of point sets. An increased attention has been devoted to the distance-based ϕ_p criterion [21] that in a generalized sense proposes to evaluate the uniformity of a point set by a scalar value of a sum of inverse mutual distances between all point pairs. A substantial part of this work is dedicated to the refinement of the entirely general definition of the ϕ_p criterion aiming at:

- reaching *statistical uniformity* of optimized samples. This means that the probability of appearance of a sampling point is equal all across the design domain. That way, using such samples shall yield statistically unbiased estimation,
- attaining *well distributed* points within each individual point sample, preferably in *noncollapsible* patterns, leading to a decrease in variance of Monte Carlo estimation,
- attaining *self-similar* patterns of design points, that is to obtain ideally optimized patterns for any sample size, N_{sim} , and dimension N_{var} ,
- developing a criterion that also *takes into account the dimension* of the problem, N_{var} .

It can be concluded that these goals have been completed by (i) incorporating the periodic extension of the design domain as first suggested in [41], (ii) deriving the minimal value of the exponent p within the ϕ_p criterion that results in self-similarity of the criterion value itself and also in self-similar patterns in arbitrary dimension, N_{var} , and finally, by (iii) introducing latinization of samples as a possible post-processing step.

As the next cornerstone of the work can be considered the task of actually materializing the notion of the related Audze-Eglājs criterion [28] that suggests to understand the evaluated

set of points as a system of mutually repelling particles. Hence, the value of the refined ϕ_p criterion has also been declared to represent the *potential energy* of a dynamical system of mutually repelling particles that interact within a periodically repeated space. After deriving the kinetic energy of particles, the Lagrangian of the system was assembled and the equations of motion of the particle system have been obtained. The actual numerical simulation of the derived dynamical particle system is a task with heavy computational demands, far beyond capabilities of a single CPU thread.

A substantial effort has been therefore allocated to author's self-study of the massively parallel computing platform Nvidia CUDA. Since the beginning of doctoral studies, the capabilities of CUDA hardware architecture and CUDA C programming language have been continuously studied. The coherent block of notes on the complex CUDA platform that is a part of this work, is intended as an accessible text for those interested in CUDA programming. Apart from the acquired knowledge, this effort turned out fruitful in development of an efficient solution of the hyper-dimensional particle system. This GPGPU solver allowed feasible optimization of samples of larger size and dimension.

The point samples optimized by the dynamical particle system using the refined ϕ_p criterion as potential energy in periodically repeated design domain tend to exhibit the desired properties defined above. Their performance in various use-cases of Monte Carlo-type estimation is presented in the closing part of this work. The dynamically optimized samples were compared against their non-periodic variant, the LHS-PAE samples optimized by mutual swapping of LHS coordinates [41], the plain Monte Carlo samples and randomized LHS samples.

Through the examples of (i) estimation of a definite integral and (ii) estimation of characteristics of a function of a random vector, a significant reduction in variance of estimated values has been concluded when using the optimized samples. Another presented example aimed at (iii) estimating of failure probability of a structure. The purpose was to demonstrate that samples with optimized uniformity across the entire design domain do not immediately bring a large reduction in variance of estimation of failure probability that typically occurs a minor subregion of the domain. The intuitive approach of using the Importance sampling method has been conducted. That way, the optimized samples regain possibility to utilize the advantage of uniformity leading to a decreased variance in the estimation of failure probability. The last presented example was dedicated to (iv) analysis of a computationally expensive, black-box FEM model. Also in this benchmark set to estimate the 5th percentile of the peak loading force by using a limited number of simulations, the dynamically optimized samples yielded estimation with statistically lower variance than the sampling methods compared.

It can be stated that the dynamically optimized samples may not necessarily surpass the LHS samples optimized by combinatorial optimization (switching) as in [39]. However, their performance in Monte Carlo estimation is not worse and the efficiency of the dynamical optimization is far superior to the combinatorial approach. The conclusions above indicate that initial objectives of the work were accomplished.

Lessons learned

As has been demonstrated, the samples optimized by the dynamical particle system using the refined ϕ_p criterion are especially suitable for usage in estimation of first two statistical

moments. For estimation of failure probability, their advantageous properties are more recognizable after transforming the sampling points into the proximity of the region of interest. Similar would be the case of estimating higher statistical moments.

In the course of this work, a significant issue of numerical instability has been encountered while simulating systems in high dimensions. Despite the implemented tools for numerical stabilization of the system (recall Section 6.1) perform as expected, with rising dimension of design space is connected the increase in the exponent in the interaction law of particles ($F_{ij} = L_{ij}^{-(N_{\text{var}}+2)}$). Therefore such a significant increase in gradient the interaction function requires a crucial decrease of the time step of the simulation to capture the movement of particles correctly enough and also to precisely enumerate the positions of particles in the optimized, steady-state.

13.1 Future work

One of future tasks directly connected to the present work is to create an open-access database to offer the optimized point samples for other research fellows.

The topics of the closest publications on sight are those mentioned in Chapter 12. Specifically the next paper in preparation concerns replacing the Qhull algorithm by the fast parallelized approximation of Voronoï diagram in relevant sample optimization techniques (such as the Minimax criterion [20]). Also, demonstrating the performance of samples gradually extended by the developed sample extension algorithm is considered as important, especially for engineering applications.

Due to the described numerical issues of the dynamical particle system governed by a power law interaction in high dimensions, the attention of the research group is focused on developing a similar N-body system that is not as numerically sensitive. The ultimate goal is to develop a pseudo-dynamical particle system that is governed by the Maximin criterion, which is the limit case of the ϕ_p criterion ($p = \infty$).

Another future topic is the development of an efficient tool for sampling optimization that yields *directionally invariant*, hyper-dimensional samples. Such samples, invariant with respect to rotation, are considered desirable when dealing with functions that contain strong interactions of input random variables.

One of goals for more distant future is to expand the gained knowledge into another field of interest. That is to begin the development of an efficient solver for lattice particle models of materials that will utilize the modern massively parallel platforms (such as CUDA [3] or OpenCL [2]) in the largest possible extent.

About author

Personal data

Jan Mašek
Institute of Structural Mechanics
Faculty of Civil Engineering
Brno University of Technology
Veveří 95, 635 00 Brno

Email: masek.j@fce.vutbr.cz
Website: www.fce.vutbr.cz/STM/masek.j

Specialization, research interests

Mathematical modeling based on physical discretization of continuum.
Parallelization of solution using the nVidia CUDA architecture.
Large deflections, chaotic behavior of deterministic dynamical systems.

Programming skills

JAVA, C, CUDA C/C++, \LaTeX , HTML, CSS, Python

Teaching Experience

BD001	Fundamentals of Structural Mechanics, Czech
BD002	Elasticity and Plasticity, Czech

Courses and other activities

06/2018	ECCOMAS Advanced Course on Computational Structural Dynamics, Institute of Thermomechanics, Czech Academy of Sciences, Prague, CZ
01/2018	Computational challenges in the reliability assessment of engineering structures. Netherlands Organization for Applied Scientific Research, Delft, NL
10/2017	ANSYS Mechanical APDL 1, SVS FEM Brno, CZ
10/2017	ANSYS Mechanical APDL 2, SVS FEM Brno, CZ
11/2017	ANSYS Mechanical APDL Dynamics, SVS FEM Brno, CZ

Participation in research projects

01/2018 – 12/2018	STM FAST	Research of topological features of discrete processes and their use for optimization of statistical sampling. (FAST-J-18-5254)
01/2018 – 12/2018	UTKO FEKT, STM FAST	Low-energy device for secured transmission from physical measurements according to Industry 4.0. (FAST/FEKT-J-18-5365)
01/2017 – 12/2017	STM FAST	Development of theory and metodics for optimized design of experiment using dynamical particle system. (FAST-J-17-4564)
01/2016 – 12/2018	STM FAST	Development of advanced sampling methods for statistical analysis of structures. (GA16-22230S)
01/2016 – 12/2016	STM FAST	Optimization of design of experiment using a discrete dynamical particle system. (FAST-J-16-3486)
01/2016 – 12/2016	UTKO FEKT, STM FAST	Aggregation gateway for secured data transmissions from instantaneous measurements of physical quantities. (FAST/FEKT-J-16-3344)
06/2015 – 12/2015	UTKO FEKT	Smart Multi-Purpose Home Gateway 2.0. Proof of concept demonstrator. Research and development of OSGi-based smart hub platform. (2015_HS18557077)

Software development

Since 2016	CUDA Hypercube	CUDA C / JAVA application serving for optimization of point samples for design of experiment.
Since 2014	GTDiPS GUI	Free JAVA application serving for advanced transformations of large multidimensional point sequences. Utilized by the staff and students of the Department of Structural mechanics. Developed in cooperation with doc. Frantík.

Intellectual properties

2017	Utility model	Aggregation gateway for secured data transmissions from instantaneous measurements of physical quantities.
------	---------------	--

List of author's publications

- [1] J. Mašek and M. Vořechovský. “Parallel implementation of hyper-dimensional dynamical particle system on CUDA”. In: *Advances in Engineering Software* (2018). DOI: 10.1016/j.advengsoft.2018.03.009.
- [2] J. Mašek and M. Vořechovský. “Parallel Implementation of Dynamical Particle System for CUDA”. In: *P. Iványi, B.H.V. Topping, G. Várady, (Editors), "Proceedings Of The Fifth International Conference On Parallel, Distributed, Grid And Cloud Computing For Engineering"*. Civil-Comp Press, Stirlingshire, UK, Paper 34, 2017. DOI: 10.4203/ccp.111.34.
- [3] J. Mašek and M. Vořechovský. “Formulation of potential for dynamical particle system applied to Monte Carlo sampling”. In: *Proceedings of 2nd International Conference on Uncertainty Quantification in Computational Sciences and Engineering*. 2017. DOI: 10.7712/120217.5384.17000.
- [4] M. Vořechovský, J. Mašek, and J. Eliáš. “Dynamical Model of Interacting Particles for the Construction of Audze–Eglajs Designs in a Periodic Space”. In: *Proceedings of the 12th International Conference on Structural Safety and Reliability (ICOSSAR2017), Vienna, Austria* (2017).
- [5] J. Mašek, P. Frantík, and M. Vořechovský. “Parallelized implementation of dynamical particle system”. In: *AIP Conference Proceedings*. Vol. 1863. 1. AIP Publishing. 2017, p. 100005. DOI: 10.1063/1.4992281.
- [6] J. Mašek and M. Vořechovský. “Performance of parallelized solution of dynamical particle system while considering the contribution of atomic operations”. In: *Proceedings of conference of postgradual students Juniorstav 2017* (2017), p. 8.
- [7] J. Mašek and M. Vořechovský. “Approximation Of Volumes Of Voronoï Cells Using Parallel Solution”. In: *Proceedings of conference of postgradual students Juniorstav 2018* (2018), p. 4.
- [8] J. Mašek and M. Vořechovský. “On Influence of Interaction Laws of Dynamical Particle System For Sample Optimization”. In: *Proceedings of scientific conference "Structural Reliability & Modelling in Mechanics 2017"*. VSB-TU Ostrava, Faculty of Civil Engineering, Ostrava, Czech republic, 2017, p. 10. ISBN: 978-80-248-4010-9. DOI: 10.1515/tvsb-2017-0016.
- [9] J. Mašek and M. Vořechovský. “On Influence of Interaction Laws of Dynamical Particle System For Sample Optimization”. In: *Transactions of the VŠB–Technical University of Ostrava, Civil Engineering Series*". VSB-TU Ostrava, Faculty of Civil Engineering, Ostrava, Czech republic, 2017, p. 10. ISBN: 978-80-248-4010-9. DOI: 10.1515/tvsb-2017-0016.
- [10] J. Mašek, P. Frantík, and M. Vořechovský. “Design of experiment using simulation of a discrete dynamical system”. In: *Transactions of the VŠB–Technical University of Ostrava, Civil Engineering Series*. Vol. 16. 2. 2016, pp. 125–134. DOI: 10.1515/tvsb-2016-0023.

- [11] J. Mašek, P. Frantík, and M. Vořechovský. “Design of experiment using a discrete dynamical system simulation”. In: *Proceedings of scientific conference "Structural Reliability & Modelling in Mechanics 2016"*. VSB-TU Ostrava, Faculty of Civil Engineering, Ostrava, Czech republic, 2016, p. 10. ISBN: 978-80-248-3917-2. DOI: 10.4203/ccp.111.34.
- [12] P. Mašek, J. Mašek, P. Frantík, R. Fujdiak, A. Ometov, J. Hošek, S. Andreev, P. Mlýnek, and J. Mišurec. “A harmonized perspective on transportation management in smart cities: the novel IoT-driven environment for road traffic modeling”. In: *Sensors* 16.11 (2016). DOI: 10.3390/s16111872.
- [13] P. Mašek, J. Hošek, K. Zeman, M. Štusek, D. Kovač, P. Čika, J. Mašek, S. Andreev, and F. Kröpfl. “Implementation of true IoT vision: survey on enabling protocols and hands-on experience”. In: *International Journal of Distributed Sensor Networks* 12.4 (2016). DOI: 10.1155/2016/8160282.
- [14] I. Havlíková, P. Frantík, J. Mašek, J. Sobek, H. Šimonová, V. Veselý, Z. Keršner, S. Seitl, I. Merta, and A. Schneemayer. “Evaluation of Fracture Tests of Concrete Specimens via Advanced Tool for Experimental Data Processing”. In: *Applied Mechanics & Materials* 821 (2016). DOI: 10.4028/www.scientific.net/AMM.821.585.
- [15] M. Štusek, J. Hošek, D. Kovač, P. Mašek, P. Čika, J. Mašek, and F. Kröpfl. “Performance analysis of the OSGi-based IoT frameworks on restricted devices as enablers for connected-home”. In: *Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2015 7th International Congress on*. IEEE. 2015, pp. 178–183. DOI: 10.1109/ICUMT.2015.7382424.
- [16] P. Mašek, K. Zeman, D. Uhlíř, J. Mašek, C. Bougiouklis, and J. Hošek. “Multi-Radio Mobile Device: Evaluation of Hybrid Node Between WiFi and LTE Networks”. In: *International Journal of Advances in Telecommunications, Electrotechnics, Signals and Systems* (2015). DOI: 10.11601/ijates.v4i2.118.
- [17] J. Mašek. “Modeling of free beam loaded with follower force”. In: *Proceedings of conference of postgradual students Juniorstav 2014* (2014), p. 8.

References

- [1] P. P. Gelsinger. “Microprocessors for the new millennium: Challenges, opportunities, and new frontiers”. In: *Solid-State Circuits Conference, 2001. Digest of Technical Papers. ISSCC. 2001 IEEE International*. IEEE. 2001, pp. 22–25.
- [2] K. Group. *The open standard for parallel programming of heterogeneous systems*. <https://www.khronos.org/opencv/>. Since 2009.
- [3] Nvidia. *Compute unified device architecture programming guide*. 2007.
- [4] A. Forrester, A. Keane, et al. *Engineering design via surrogate modelling: a practical guide*. John Wiley & Sons, 2008.
- [5] R. A. Fisher. *The design of experiments*. Oliver and Boyd; Edinburgh; London, 1937.
- [6] V. L. Anderson and R. A. McLean. *Design of experiments: a realistic approach*. Vol. 5. CRC Press, 1974.
- [7] N. Metropolis and S. Ulam. “The Monte Carlo method”. In: *Journal of the American statistical association* 44.247 (1949), pp. 335–341.
- [8] N. Metropolis. “The beginning of the Monte Carlo method”. In: *Los Alamos Science Special Issue* (1987), pp. 125–130.
- [9] J. M. Hammersley and D. C. Handscomb. “General principles of the Monte Carlo method”. In: *Monte Carlo Methods*. Springer, 1964, pp. 50–75.
- [10] I. M. Sobol’. “The Monte Carlo Method”. In: (1974).
- [11] Wikipedia contributors. *Moment (mathematics) — Wikipedia, The Free Encyclopedia*. [https://en.wikipedia.org/w/index.php?title=Moment_\(mathematics\)&oldid=838597128](https://en.wikipedia.org/w/index.php?title=Moment_(mathematics)&oldid=838597128). [Online; accessed 12-June-2018]. 2018.
- [12] J. F. Koksma. “Een algemeene stelling uit de theorie der gelijkmatige verdeeling modulo 1”. In: *Mathematica B* 11 (1942/1943), pp. 7–11.
- [13] H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. Philadelphia, Pennsylvania: Society for Industrial and Applied Mathematics, 1992. ISBN: 0-89871-295-5.
- [14] V. R. Joseph. “Space-filling designs for computer experiments: A review”. In: *Quality Engineering* 28.1 (2016), pp. 28–35.
- [15] K. Smith. “On the standard deviations of adjusted and interpolated values of an observed polynomial function and its constants and the guidance they give towards a proper choice of the distribution of observations”. In: *Biometrika* 12.1/2 (1918), pp. 1–85.
- [16] D. Titterington. “Optimal design: Some geometrical aspects of D-optimality”. In: *Biometrika* 62.2 (1975), pp. 313–320.
- [17] P. F. de Aguiar, B. Bourguignon, M. Khots, D. Massart, and R. Phan-Thau-Luu. “D-optimal designs”. In: *Chemometrics and Intelligent Laboratory Systems* 30.2 (1995), pp. 199–210.
- [18] J. P. Van de Geer. *Some aspects of Minkowski distance*. Leiden University, Department of Data Theory, 1995.

- [19] Wikipedia contributors. *Minkowski distance* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Minkowski_distance&oldid=834553041. [Online; accessed 14-June-2018]. 2018.
- [20] M. Johnson, L. Moore, and D. Ylvisaker. “Minimax and maximin distance designs”. In: *Journal of Statistical Planning and Inference* 2.26 (1990), pp. 131–148. ISSN: 0378-3758. DOI: 10.1016/0378-3758(90)90122-B.
- [21] M. D. Morris and T. J. Mitchell. “Exploratory designs for computational experiments”. In: *Journal of Statistical Planning and Inference* 43.3 (1995), pp. 381–402. ISSN: 0378-3758. DOI: 10.1016/0378-3758(94)00035-T.
- [22] P. Shirley et al. “Discrepancy as a quality measure for sample distributions”. In: *Proc. Eurographics*. Vol. 91. 1991, pp. 183–194.
- [23] H. Niederreiter. *Random number generation and quasi-Monte Carlo methods*. Vol. 63. Siam, 1992.
- [24] K.-T. Fang and Y. Wang. *Number-Theoretic Methods in Statistics*. 1st edition. London ; New York: Chapman and Hall/CRC, 1993. ISBN: 9780412465208.
- [25] K.-T. Fang and C.-X. Ma. “Wrap-Around L_2 -Discrepancy of Random Sampling, Latin Hypercube and Uniform Designs”. In: *Journal of Complexity* 17.4 (2001), pp. 608–624. ISSN: 0885-064X. DOI: 10.1006/jcom.2001.0589.
- [26] K.-T. Fang, C.-X. Ma, and P. Winker. “Centered L_2 -discrepancy of Random Sampling and Latin Hypercube Design, and Construction of Uniform Designs”. In: *Mathematics of Computation* 71.237 (2000), pp. 275–296. ISSN: 1088-6842. DOI: 10.1090/S0025-5718-00-01281-3.
- [27] V. R. Joseph and Y. Hung. “Orthogonal-maximin Latin hypercube designs”. In: *Statistica Sinica* (2008), pp. 171–186.
- [28] P. Audze and V. Eglājs. “New approach for planning out of experiments”. In: *Problems of Dynamics and Strengths* 35 (1977). (in Russian), pp. 104–107.
- [29] H. Niederreiter. “Low-discrepancy and low-dispersion sequences”. In: *Journal of Number Theory* 30.1 (1988), pp. 51–70. ISSN: 0022-314X. DOI: 10.1016/0022-314X(88)90025-X.
- [30] I. Sobol’. “Uniformly distributed sequences with additional uniformity properties”. In: *USSR Comput. Math. and Math. Phy* 16 (1976), pp. 236–242.
- [31] J. Halton. “On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals”. In: *Numerische Mathematik* 2.1 (1960), pp. 84–90. ISSN: 0029-599X. DOI: 10.1007/BF01386213.
- [32] P. Bratley, B. L. Fox, and H. Niederreiter. “Implementation and tests of low-discrepancy sequences”. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 2.3 (1992), pp. 195–213.
- [33] J. M. Hammersley. “Monte Carlo methods for solving multivariable problems”. In: *Annals of the New York Academy of Sciences* 86.1 (1960), pp. 844–874.

-
- [34] W. J. Conover. “On a Better Method for Selecting Input Variables”. unpublished Los Alamos National Laboratories manuscript, reproduced as Appendix A of “Latin Hypercube Sampling and the Propagation of Uncertainty in Analyses of Complex Systems” by J.C. Helton and F.J. Davis, Sandia National Laboratories report SAND2001-0417. 1975.
- [35] M. D. McKay, W. J. Conover, and R. J. Beckman. “A comparison of three methods for selecting values of input variables in the analysis of output from a computer code”. In: *Technometrics* 21 (1979), pp. 239–245. DOI: 10.1080/00401706.1979.10489755.
- [36] M. Stein. “Large sample properties of simulations using Latin hypercube sampling”. In: *Technometrics* 29.2 (1987), pp. 143–151.
- [37] M. Vořechovský. “Hierarchical refinement of Latin hypercube samples”. In: *Computer-Aided Civil and Infrastructure Engineering* 30.5 (2015), pp. 394–411.
- [38] M. Grant, S. Boyd, and Y. Ye. *CVX: Matlab software for disciplined convex programming*. 2008.
- [39] M. Vořechovský and D. Novák. “Correlation control in small sample Monte Carlo type simulations I: A Simulated Annealing approach”. In: *Probabilistic Engineering Mechanics* 24.3 (2009), pp. 452–462. ISSN: 0266-8920. DOI: 10.1016/j.probengmech.2009.01.004.
- [40] D. Novák, M. Vořechovský, and B. Teplý. “FReET: Software for the statistical and reliability analysis of engineering problems and FReET-D: Degradation module”. In: *Advances in Engineering Software* 72 (2014), pp. 179–192. DOI: 10.1016/j.advengsoft.2013.06.011.
- [41] J. Eliáš and M. Vořechovský. “Modification of the Audze–Eglājs criterion to achieve a uniform distribution of sampling points”. In: *Advances in Engineering Software* 100 (2016), pp. 82–96.
- [42] M. Vořechovský and J. Eliáš. “Improved formulation of Audze–Eglājs criterion for space-filling designs”. In: *ICASP12, the 12th International Conference on Applications of Statistics and Probability in Civil Engineering held in Vancouver, Canada on July 12-15, 2015*. Ed. by T. Haukaas. The University of British Columbia. 2015, pp. 1–8. DOI: 10.14288/1.0076173. URL: <http://hdl.handle.net/2429/53478>.
- [43] M. Liefvendahl and R. Stocki. “A study on algorithms for optimization of Latin hypercubes”. In: *Journal of Statistical Planning and Inference* 136.9 (2006), pp. 3231–3247.
- [44] J. Mašek and M. Vořechovský. “Formulation of potential for dynamical particle system applied to Monte Carlo sampling”. In: *Proceedings of 2nd International Conference on Uncertainty Quantification in Computational Sciences and Engineering*. 2017. DOI: 10.7712/120217.5384.17000.
- [45] V. Sadílek and M. Vořechovský. “Evaluation of pairwise distances among points forming a regular orthogonal grid in a hypercube”. In: *Journal of Civil Engineering and Management* (2018), in press.
- [46] E. Athanassoula, E. Fady, J. Lambert, and A. Bosma. “Optimal softening for force calculations in collisionless N-body simulations”. In: *Monthly Notices of the Royal Astronomical Society* 314.3 (2000), pp. 475–488.

- [47] J. S. Kilby. *Miniaturized electronic circuits*. US Patent 3,138,743. June 1964.
- [48] R. N. Noyce. *Semiconductor device-and-lead structure*. US Patent 2,981,877. Apr. 1961.
- [49] G. Moore. “Moore’s law”. In: *Electronics Magazine* 38.8 (1965), p. 114.
- [50] M. Flynn. “Flynn’s taxonomy”. In: *Encyclopedia of parallel computing*. Springer, 2011, pp. 689–697.
- [51] Nvidia. *Nvidia Launches the World’s First Graphics Processing Unit: GeForce 256*. http://www.nvidia.com/object/IO_20020111_5424.html. 1999.
- [52] A. Peleg and U. Weiser. “MMX technology extension to the Intel architecture”. In: *IEEE micro* 16.4 (1996), pp. 42–50.
- [53] K. Group. *OpenGL - The Industry’s Foundation for High Performance Graphics*. <https://www.opengl.org/>. Since 1992.
- [54] Microsoft. *Microsoft DirectX / Direct3D*. [https://msdn.microsoft.com/cs-cz/library/windows/desktop/hh309466\(v=vs.85\).aspx](https://msdn.microsoft.com/cs-cz/library/windows/desktop/hh309466(v=vs.85).aspx). Since 1996.
- [55] Nvidia. *Nvidia CUDA C/C++*. <https://developer.nvidia.com/cuda-toolkit>. Since 2007.
- [56] PGI. *PGI CUDA Fortran Compiler*. <http://www.pgroup.com/resources/cudafortran.htm>. Since 2009.
- [57] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. “PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation”. In: *Parallel Computing* 38.3 (2012), pp. 157–174. ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.09.001.
- [58] Nvidia. *Nvidia PyCuda*. <https://developer.nvidia.com/pycuda>. Since 2012.
- [59] Nvidia. *Nvidia Anaconda Accelerate*. <https://developer.nvidia.com/anaconda-accelerate>. Since 2014.
- [60] Nvidia. *GPU-accelerated Libraries for Computing*. <https://developer.nvidia.com/gpu-accelerated-libraries>.
- [61] D. Foley and J. Danskin. “Ultra-performance Pascal GPU and NVLink interconnect”. In: *IEEE Micro* 37.2 (2017), pp. 7–17.
- [62] Nvidia. *Nvidia Pascal GP100 Architecture Whitepaper*. <https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf>. 2016.
- [63] Nvidia. *Nvidia Kepler GK110 Architecture Whitepaper*. <https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>. 2012.
- [64] Nvidia. *CUDA Toolkit Documentation*. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Since 2007.
- [65] B. W. Coon and J. E. Lindholm. *System and method for managing divergent threads in a SIMD architecture*. US Patent 7,353,369. Apr. 2008.

-
- [66] B. W. Coon and J. E. Lindholm. *System and method for managing divergent threads using synchronization tokens and program instructions that include set-synchronization bits*. US Patent 7,543,136. June 2009.
- [67] B. W. Coon, J. E. Lindholm, P. C. Mills, and J. R. Nickolls. *Processing an indirect branch instruction in a SIMD architecture*. US Patent 7,761,697. July 2010.
- [68] B. W. Coon, J. R. Nickolls, J. E. Lindholm, and S. D. Tzvetkov. *Structured programming control flow in a SIMD architecture*. US Patent 7,877,585. Jan. 2011.
- [69] Wikipedia contributors. *Dynamic random-access memory* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Dynamic_random-access_memory&oldid=840605351. [Online; accessed 2-June-2018]. 2018.
- [70] Wikipedia contributors. *Static random-access memory* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Static_random-access_memory&oldid=843792364. [Online; accessed 2-June-2018]. 2018.
- [71] R. Farber. *CUDA application design and development*. Elsevier, 2011.
- [72] Nvidia. *Parallel Thread Execution ISA*. <http://docs.nvidia.com/cuda/parallel-thread-execution/index.html>.
- [73] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. John Wiley & Sons, 2014.
- [74] V. Volkov. “Understanding Latency Hiding on GPUs”. PhD thesis. University of California, Berkeley, 2016.
- [75] G. Ruetsch and M. Fatica. *CUDA Fortran for scientists and engineers: best practices for efficient CUDA Fortran programming*. Elsevier, 2013.
- [76] N. Wilt. *The CUDA handbook: A comprehensive guide to GPU programming*. Pearson Education, 2013.
- [77] S. Cook. *CUDA programming: a developer’s guide to parallel computing with GPUs*. Newnes, 2012.
- [78] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. “Demystifying GPU microarchitecture through microbenchmarking”. In: *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 235–246.
- [79] Nvidia. *Nvidia Volta Architecture Whitepaper*. <http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>. 2018.
- [80] Wikipedia contributors. *Taylor series* — *Wikipedia, The Free Encyclopedia*. 2018. URL: https://en.wikipedia.org/w/index.php?title=Taylor_series&oldid=839466422.
- [81] M. Andersch, J. Lucas, M. Alvarez-Mesa, and J. Ben. “Analyzing GPGPU Pipeline Latency”. In: *Tenth International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems*. Fiuggi, Italy. 2014, p. 1.
- [82] NVIDIA. “GPU occupancy calculator”. In: *CUDA SDK (2010)*. URL: http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls.

- [83] J. D. Little. “A proof for the queuing formula: $L = \lambda W$ ”. In: *Operations research* 9.3 (1961), pp. 383–387.
- [84] J. D. Little. “OR FORUM—Little’s Law as Viewed on Its 50th Anniversary”. In: *Operations research* 59.3 (2011), pp. 536–549.
- [85] V. Volkov. “Better performance at lower occupancy”. In: *Proceedings of the GPU technology conference, GTC*. Vol. 10. San Jose, CA. 2010, p. 16.
- [86] V. Volkov and J. W. Demmel. “Benchmarking GPUs to tune dense linear algebra”. In: *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*. IEEE. 2008, pp. 1–11.
- [87] R. G. Belleman, J. Bédorf, and S. F. P. Zwart. “High performance direct gravitational N-body simulations on graphics processing units II: An implementation in CUDA”. In: *New Astronomy* 13.2 (2008), pp. 103–112.
- [88] E. Gaburov, S. Harfst, and S. P. Zwart. “SAPPORO: A way to turn your graphics cards into a GRAPE-6”. In: *New Astronomy* 14.7 (2009), pp. 630–637.
- [89] S. Dindar, E. B. Ford, M. Juric, Y. I. Yeo, J. Gao, A. C. Boley, B. Nelson, and J. Peters. “Swarm-NG: A CUDA library for Parallel n-body Integrations with focus on simulations of planetary systems”. In: *New Astronomy* 23 (2013), pp. 6–18.
- [90] S. von Hoerner. “Die numerische Integration des n-Körper-Problemes für Sternhaufen. I”. In: *Zeitschrift für Astrophysik* 50.10-11 (1960), pp. 184–214.
- [91] J. Barnes and P. Hut. “A hierarchical O ($N \log N$) force-calculation algorithm”. In: *nature* 324.6096 (1986), pp. 446–449.
- [92] L. Greengard and V. Rokhlin. “A fast algorithm for particle simulations”. In: *Journal of computational physics* 73.2 (1987), pp. 325–348.
- [93] C. A. White and M. Head-Gordon. “Derivation and efficient implementation of the fast multipole method”. In: *The Journal of Chemical Physics* 101.8 (1994), pp. 6593–6605.
- [94] D. Kirk et al. “Nvidia CUDA software and GPU parallel computing architecture”. In: *ISMM*. Vol. 7. 2007, pp. 103–104.
- [95] T. Hamada and T. Iitaka. “The chamomile scheme: An optimized algorithm for n-body simulations on programmable graphics processing units”. In: *arXiv preprint astro-ph/0703100* (2007).
- [96] T. Hamada. “Internals of the cunbody-1 library: particle/force decomposition and reduction”. In: *AstroGPU 2007, Princeton* (2007).
- [97] L. Nyland, M. Harris, J. Prins, et al. “Fast n-body simulation with CUDA”. In: *GPU gems* 3.31 (2007), pp. 677–695.
- [98] M. Harris. “Optimizing CUDA”. In: *SC07: High Performance Computing With CUDA* (2007).
- [99] J. Mašek and M. Vořechovský. “Parallel Implementation of Dynamical Particle System for CUDA”. In: *P. Iványi, B.H.V. Topping, G. Várady, (Editors), "Proceedings Of The Fifth International Conference On Parallel, Distributed, Grid And Cloud Computing For Engineering"*. Civil-Comp Press, Stirlingshire, UK, Paper 34, 2017. DOI: 10.4203/ccp.111.34.

-
- [100] M. Vořechovský and J. Eliáš. “Modification of the MaxiMin and phi-criteria to achieve a uniform distribution of sampling points”. In: *Technometrics* (2018), in review.
- [101] J. P. Kleijnen. “Kriging metamodeling in simulation: A review”. In: *European journal of operational research* 192.3 (2009), pp. 707–716.
- [102] S. H. Lee and B. M. Kwak. “Response surface augmented moment method for efficient reliability analysis”. In: *Structural safety* 28.3 (2006), pp. 261–272.
- [103] U. Bourgund and C. Bucher. “Importance sampling procedure using design points (ispud)-a user’s manual”. In: *Institute of Engineering Mechanics, University of Innsbruck* (1986).
- [104] C. G. Bucher. “Adaptive sampling—an iterative fast Monte Carlo procedure”. In: *Structural safety* 5.2 (1988), pp. 119–126.
- [105] C. G. Hoover, Z. P. Bažant, J. Vorel, R. Wendner, and M. H. Hubler. “Comprehensive concrete fracture tests: description and results”. In: *Engineering fracture mechanics* 114 (2013), pp. 92–103.
- [106] B. Patzák. “OOFEM - an object-oriented simulation tool for advanced modeling of materials and structures”. In: *Acta Polytechnica* 52 (2012), pp. 59–66.
- [107] B. Patzák and Z. Bittnar. “Design of object oriented finite element code”. In: *Advances in Engineering Software* 32.10-11 (2001), pp. 759–767.
- [108] B. Patzák and D. Rypl. “Object-oriented, parallel finite element framework with dynamic load balancing”. In: *Advances in Engineering Software* 47.1 (2012), pp. 35–50.
- [109] G. Voronoï. “Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Deuxième mémoire. Recherches sur les paralléloèdres primitifs.” In: *Journal für die reine und angewandte Mathematik* 134 (1908), pp. 198–287.
- [110] F. Aurenhammer. “Voronoi diagrams – a survey of a fundamental geometric data structure”. In: *ACM Computing Surveys (CSUR)* 23.3 (1991), pp. 345–405.
- [111] D.-M. Yan, W. Wang, B. Lévy, and Y. Liu. “Efficient computation of clipped Voronoi diagram for mesh generation”. In: *Computer-Aided Design* 45.4 (2013), pp. 843–852.
- [112] D.-M. Yan, K. Wang, B. Lévy, and L. Alonso. “Computing 2D periodic centroidal Voronoi tessellation”. In: *Voronoi Diagrams in Science and Engineering (ISVD), 2011 Eighth International Symposium on*. IEEE, 2011, pp. 177–184.
- [113] M. Vořechovský, V. Sadílek, and J. Eliáš. “Application of Voronoi Weights in Monte Carlo Integration with a Given Sampling Plan”. In: *REC 2016*. 2016, pp. 441–452.
- [114] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. “The quickhull algorithm for convex hulls”. In: *ACM Transactions on Mathematical Software (TOMS)* 22.4 (1996), pp. 469–483.
- [115] B. Büeler, A. Enge, and K. Fukuda. “Exact volume computation for polytopes: a practical study”. In: *Polytopes—combinatorics and computation*. Springer, 2000, pp. 131–154.
- [116] B. Delaunay. “Sur la sphere vide”. In: *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk* 7.793-800 (1934), pp. 1–2.