DOI: 10.14529/jsfi210306

Efficient Implementation of Liquid Crystal Simulation Software on Modern HPC Platforms

Ilya V. Afanasyev[1,2], *Dmitry I. Lichmanov*[1,2], *Vladimir Yu. Rudyak*[3], *Vadim V. Voevodin*[1,2]

In this paper we demonstrate the process of efficient porting a software package for Markov chain Monte Carlo (MCMC) simulations on a finite cubic lattice on multiple modern architectures: Pascal, Volta and Turing NVIDIA GPUs, NEC SX-Aurora TSUBASA vector engines and Intel Xeon Gold processors. In the studied software, MCMC methodology is used for simulations of liquid crystal structures, but it can be as well employed in a wide range of problems of mathematical physics and numerical methods. The main goals of this work are to determine the best software optimization strategy for this class of algorithms and to examine the speed and the efficiency of such simulations on modern HPC platforms. We evaluate the effects of various optimizations, such as using more suitable memory access patterns, multitasking for efficient utilization of massive parallelism on the target architectures, improved cache hit-rates, parallel workload balancing, etc. We perform a detailed performance analysis for each target platform using software tools such as nvprof, Ftrace and VTune. On this basis, we evaluate and compare the efficiency of the developed computational kernels on different platforms and subsequently rank these platforms by their performance. The results show that NVIDIA GPU and NEC SX-Aurora TSUBASA platforms, although at first glance seem very different, require similar optimization approaches in many cases due to similarities in data processing principles.

Keywords: NVIDIA GPU, NEC SX-Aurora TSUBASA, liquid crystals, HPC, co-design, performance optimization, Monte Carlo, cubic lattice.

Introduction

The program under study is a software package developed specifically for meso- and macroscopic computer simulations of structure and properties of nematic and cholesteric liquid crystal droplets. Although this software is aimed to solve the specific physical problem, the computational task beyond it belongs to the one of the most important classes of computational problems. Optimization of a functional defined on a finite space cubic lattice relates to a wide range of problems from mathematics and physics to economics and operational research, which require high efficiency implementations. For example, in mathematics and natural sciences, Markov chain Monte Carlo (MCMC) simulations on cubic lattice are used in methodological studies of novel Monte Carlo techniques [13, 14, 37], spin models [9, 17, 44], quantum Monte Carlo [7, 26, 29], material design [47], bio-chemistry [31], etc. Keeping that in mind, we will focus on both the current implementation and general approach to the optimization of this type of algorithms. We believe that the optimization techniques demonstrated below are applicable to a wide range of computational problems dealing with Markov chain Monte Carlo simulations and stochastic optimization on finite space cubic lattice.

Liquid crystals (LCs) are the perfect example of soft matter materials that combine the typical properties of a crystalline solid and a viscous liquid [12]. This gives LCs unique physical properties and allows many applications of LC materials. LCs are mostly known for their use in liquid crystal displays, embedded today in almost every device. At the same time, there is a broad variety of more sophisticated applications nowadays: chemical sensors [38, 40], tunable optical

[1]Moscow Center of Fundamental and Applied Mathematics, Moscow, Russia
[2]Research Computing Center, Lomonosov Moscow State University, Moscow, Russia
[3]Faculty of Physics, Lomonosov Moscow State University, Moscow, Russia

devices [11, 22], biomedicine applications [18], and many others [21]. For the most complex applications, it is crucial to precisely understand the fundamental behaviour of LCs in various conditions, their orientational structure and properties. While there are theoretical descriptions for such problems, the analytical solutions are typically unavailable due to the complexity of the problems. At this point, computer simulation techniques are typically used to solve such problems.

The program under study implements the extended Frank elastic continuum approach to describe the energy of the system [34]. Markov chain Monte Carlo simulated annealing is used to find energy-optimal structures of droplets of liquid crystal (LC droplets) droplet structures. It previously showed good results for nematic and cholesteric LC [20, 35, 39]. The use of MCMC stochastic optimization by simulated annealing [10, 30] allows us to almost completely ignore the problem of the trapping into local minima, in contrast to the Newtonian-like iterative methods. Moreover, checkerboard decomposition algorithms [45] result in great computational efficiency and parallelizability, which is critically important for GPU implementation [5, 33]. The original software package was developed back in 2012 [34] for NVIDIA Fermi GPU architecture. Since that time, both GPU hardware and software has changed significantly, which makes it reasonable to update the computational core of the software to fit the abilities of Pascal, Volta and Turing GPUs. Moreover, it seems promising to try to port this software to vector processor architectures (VCPUs). VCPUs show significant progress in solving vectorizable problems like Markov chain Monte Carlo simulations [2, 27, 32, 41]. In particular, massively parallel NEC SX-Aurora TSUBASA vector processors equipped with high-bandwidth memory (HBM) as well as the newest Intel Xeon processors with AVX-512 vector instructions may provide significant acceleration for cubic lattice Monte Carlo problems [16].

In this paper, we use supercomputing co-design approach (i.e., porting the evaluated software to multiple target platforms with the subsequent selection of the platform which demonstrates the highest performance on the particular problem) in order to develop an efficient and high-performance implementation of the software package for simulation of LC droplets. For this purpose, three modern target platforms are investigated in this paper: NVIDIA GPUs of Volta, Pascal and Turing architectures, NEC SX-Aurora TSUBASA vector processors, and Intel Xeon Skylake multicore CPUs. These platforms belong to a significant and representative subclass of modern supercomputing architectures, which provide high-performance computational units and high-bandwidth memory. For each target architecture we describe implementation and optimization approaches, typically required to achieve high performance for the studied class of problems: selecting efficient memory access patterns, using multitasking to efficiently utilize massive parallelism, improving cache hit-rates, parallel workload balancing, and several others. Finally, we present a comparative analysis of these implementations for different target platforms and discuss the efficiency of these platforms for this class of computational problems.

The rest of the paper is organized as follows. Section 1 describes hardware features and properties of target architectures used in this work. Section 2 explains in detail what kind of calculations forms the main computationally-intensive part of the program being analyzed and why the question of its optimization is not trivial but actual. In section 3, we briefly describe a physical task solved within the entire program package as well as its implementation features. Section 4 covers a detailed description of a typical computational kernel and its contribution to a whole program package, while section 5 describes the pipeline of optimizations applied to this typical kernel on all target architectures. Conclusion summarizes the study and points out the main results of this work.

1. Target Architectures

1.1. NVIDIA GPU

Modern NVIDIA GPU consists of a set of identical Streaming Multiprocessors (SM), each of which has multiple CUDA cores, capable of performing various types of operations. CUDA (Compute Unified Device Architecture) programming model allows users to define kernels – special functions, which are executed on GPUs, and grids – configurations of these kernels, which define the number of threads used by each kernel. NVIDIA GPUs employ Single Instruction Multiple Thread (SIMT) computational model, in which threads are organized in groups of size 32 (called warps), and all threads of one warp execute the same instruction at any given moment of time. Typically each thread performs almost identical computational workflow over its own data – so called data-driven parallelism.

At the time of this research, the most recent NVIDIA GPU microarchitectures included Pascal, Volta, Turing and Ampere. For the performance evaluations in this paper, machines equipped with Pascal (P100), Volta GPUs (V100) and desktop Turing (GeForce RTX 2080 Ti) are used. Unfortunately, Ampere GPU servers were unavailable to us at the moment of this writing. An execution model is similar for all recent generations of GPUs, however, they have different hardware characteristics, the most important for our paper being theoretical peak performances and the structure of memory hierarchy. Specifications of these GPUs are shown in Tab. 1.

Table 1. Specifications of the GPU architectures used in this work

Architecture	**Pascal**	**Volta**	**Turing**
Release year	2016	2017	2018
Model	P100	V100	RTX 2080 Ti
Memory type	HBM2	HBM2	GDDR6
Memory bus, bit	4096	4096	352
Memory size, GB	16	32	11
Theoretical peak memory bandwidth, GB/s	720	900	616
L1 cache per SM, KB	64	128	64
L2 cache, KB	4096	6144	5632
Peak performance (float), TFlop/s	9.5	14.3	11.8

1.2. NEC SX-Aurora TSUBASA

The NEC SX-Aurora TSUBASA architecture with dedicated vector processors [19, 46] inherits the design concepts of a vector supercomputer and enhances its advantages to achieve higher sustained performance and higher usability. NEC SX-Aurora TSUBASA mainly consists of vector engines (VEs), equipped with a vector processor and a vector host (VH) of an x86 node. VE is used as a primary processor for executing applications, while the VH is used as a secondary processor for executing basic operating system functions that are offloaded from the

VE. VE has eight powerful vector cores with the total peak performance of 4.91 TFlop/s on single precision and 2.45 TFlop/s on double precision.

Each SX-Aurora vector core consists of three components: a scalar processing unit (SPU), a vector processing unit (VPU), and a memory subsystem. The majority of computations are performed by VPUs, while SPUs provide the functionality of a typical CPU. Since SX-Aurora is not just a typical accelerator but rather a self-sufficient processor, SPUs are designed to provide relatively high performance on scalar computations. In order to store the results of intermediate calculations, each vector core is equipped with 64 vector registers with a total register capacity equal to 128 KB. Each register is designed to store a vector of 256 double precision elements. On the memory subsystem side, six HBM modules in the vector processor can deliver up to 1.22 TB/s theoretical memory bandwidth [8] with up to 48 GB total capacity. Parallel programs for the NEC SX-Aurora TSUBASA architecture are implemented via OpenMP programming model, while vectorization is performed by NEC compiler (the user inserts specific directives, which help the compiler to perform automatic vectorization).

1.3. Intel Xeon

Intel Xeon Gold 6126 processors of Skylake microarchitecture have been used in order to evaluate the performance of modern CPUs in this paper. These 12-core processors achieve theoretical peak performance of almost 2 TFlop/s on double precision and 4 TFlop/s on single precision. Each core has a private L1 data cache of 32 KB size, a private L2 cache of 1 MB size, while all cores share a 19.25 MB non-inclusive L3 cache. Intel Xeon Gold 6126 processors support Advanced Vector Extensions 512 (AVX-512), capable of processing 16 single precision values on each cycle. Theoretical peak DRAM memory bandwidth of these processors is equal to 125 GB/s in the configuration available to us.

2. Formulation of the Problem

Since the beginning of the era of general-purpose computing on graphics accelerators, various algorithms of computational physics have been ported to NVIDIA GPUs and thoroughly optimized for this architecture [15, 28]. Mesh methods and particularly cubic lattice methods allow researchers to utilize the massive parallelizm of GPUs for the two following reasons [6, 23, 42]. First, a parallelization method is natively embedded into the lattice decomposition. Second, these methods often do not require storing and exchanging any additional data between threads (in contrast to the molecular dynamics, for example). The most common approaches used for the physical problems formulated on cubic lattice are solutions of transport equations (for example, lattice Boltzmann methods used for computational fluid dynamics, CFD [43]), Newtonian-like function optimization [10], and Monte Carlo methods (used for wide range of statistical physics problems in soft matter, bio-chemistry and quantum physics [24, 26, 31, 36, 48], and also for function optimization [30]).

In the case of the software studied in this paper, cubic lattice Monte Carlo methods are used. At the first glance, it seems that, in terms of optimization approaches, these methods should be very similar to the stencil calculations typically present in transport equations. For example, in the problem of liquid crystal structure optimizations, the free energy of the system (the value to be minimized) is a function of 3D distribution of a director vector (so called “structure”, which is varied to deliver the minimum to the free energy). The free energy in each sub-volume

of the lattice is calculated on the basis of the nearest values of director field, effectively forming $3 \times 3 \times 3$ size stencil for the calculations. And the optimization of stencil calculations for GPUs is thoroughly studied [25], consisting primarily of the usage of shared memory, utilizing registers to increase volume of cached data, block tiling, depending on the order of a numerical scheme (effectively, the size of a stencil) and the form of the used equations.

However, there are critical differences between traditional stencil calculations and Monte Carlo approach, which also makes the optimization strategies for these methods very different. Let us take a closer look at the problem to understand these differences.

First, approaches to choosing the optimal stencil size are nearly opposite. In stencil calculations, the use of numerical scheme of higher order grants higher stability of the numerical scheme and larger stencil size. In turn, using a larger stencil size increases data reuse without changing the available degree of parallelism, leading to a more efficient implementation (see Fig. 1, where stencils are shown as square brackets).

Monte Carlo calculations, for the most of physical problems, require small stencil size. Each MC step consists of three intrinsic parts: (1) trial change of the state, (2) the update of the energy function (or other function), and (3) the decision step by probabilistically accepting some trial changes (mostly favorable) and declining others (mostly unfavorable). MC steps at lattice points located at a stencil size and beyond can be performed independently in parallel. In this case, using smaller stencil size increases the available degree of parallelism without changing the data reuse (which is almost absent in this type of MC calculations), see Fig. 1. This type of space decomposition-based parallelizm is called checkerboard algorithm [45]. Practically important stochastic optimization problems often require as many MC steps as possible, which makes the checkerboard parallelization strategy one of the most useful for this type of problems [3, 5, 33]. Thus, for stencil calculations larger stencil is optimal as it allows better utilizing the limited data transfer rate, but for Monte Carlo the smaller stencil size is optimal as it allows using higher number of parallel threads.

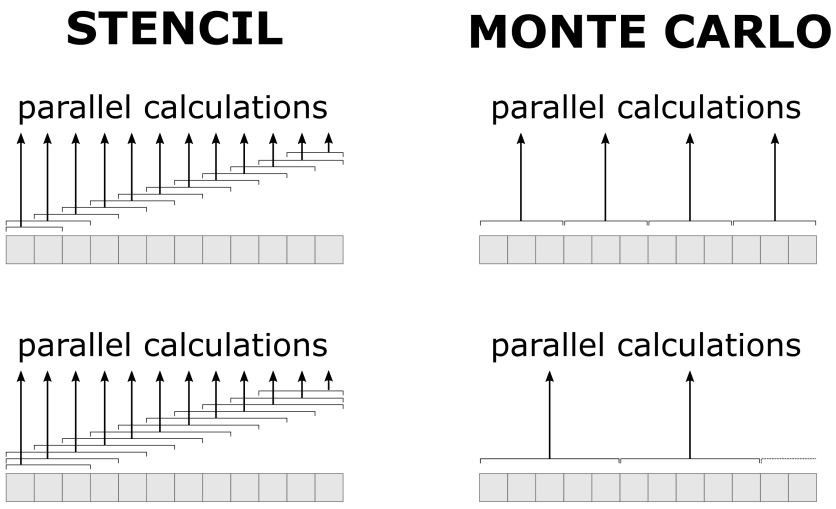


Figure 1. Simplified parallelization schemes for stencil-like calculations (left column) and Monte Carlo with checkerboard decomposition (right column) for 1D problem. Two rows demonstrate changes in parallelizm with increasing stencil size

Second difference concerns the operations with the data. In traditional stencil calculations, the data operations are pretty straightforward: the changes in the state turns to the changes in the function, which turns in the changes in the state on the next iteration. The memory

operations are similar: read the state, then write the output values after calculations. In Monte Carlo, each change will be accepted or declined with some probability. It raises the need for more memory operations: (1) each step starts with copying the current state, its energy and auxiliary parameters; (2) each step may end with restoring the state if the step is declined, which effectively is copying back all the data. It results into massive memory operations in the beginning and the end of each MC step, which is unavoidable due to the nature of Monte Carlo method.

Third, traditional stencil calculations require relatively low amount of memory per calculation and produce relatively low number of operations per step. In this case, the processing power is often limited by the memory bandwidth of the device, and the use of shared memory improves the situation significantly. In contrast, in our simulations the mathematical equations lying behind the calculations are rather complex. Calculations in each point require $3 \times 3 \times 3$ values of two 3D vector fields, large number of temporary data for the minimization of recurring calculations, and around 10 auxiliary output values to make the result not just valid but also meaningful for the end user. This forms a large amount of memory used per lattice point per step, which is far beyond the capabilities of shared memory of modern GPU devices. For example, NVIDIA V100 and RTX 2080 Ti contain only 64 KB of L1 cache per SM, while the suggested numerical scheme requires at least 100 float values per thread.

It can be seen that each iteration of MC calculations in our case requires a lot of computations as well as intensive memory usage. It should be noted, that the high memory usage for calculations is not a sequence of non-optimality of the mathematical approach or the technical implementation. For many real world problems, it turns out the same way due to the continuously increasing complexity of physical phenomena taken into account, even when the basic equations seem very simple. Clearly, it is applicable for all lattice Monte Carlo problems, not just for the topic of liquid crystals.

To sum it up, nowadays the arsenal of lattice algorithms is optimized for NVIDIA GPUs for the two major cases: stencil-type and Monte Carlo calculations with relatively low memory usage and relatively low computational intensity per step. However, many physical problems require Monte Carlo calculations with both intense memory usage and computational operations. There are serious optimization issues with applying traditional optimizations to this class of problems. Our aim is to study its optimization possibilities. So, we need to seek optimal memory access patterns inside computational kernels and efficient parallelization scheme.

3. Brief Description of the Evaluated Program Package

The evaluated program package utilizes the cubic lattice approach, in which the whole 3D volume of an LC droplet is divided into cells by a cubic lattice of a predefined size. A special first-order scheme is used, thus the calculations utilize the data on the director vector value and its first spatial derivative. This scheme was optimized for unit length quasi-vector fields [34]. It helps to efficiently implement highly parallel Monte Carlo simulations with the degree of parallelism of $N_x N_y N_z/8$, where N_x, N_y and N_z are the dimensions of the lattice. The program package has been implemented in CUDA C, with CPU host used only for an initialization and input/output functions. The original version of the package from 2012 required double precision calculations for accurate results and has been briefly optimized in terms of mathematical algorithms complexity and memory consumption.

When we use the evaluated package to solve real-life physical problems, we typically produce preliminary, main and precise calculations. Preliminary calculations consist of tens to hundreds of tasks on small and medium lattices (from 16^3 to 32^3). Typically, these calculations are used to check the physical problem, task parameters and also to find basic physical regimes of the system. Main calculations consist of hundreds to thousands of tasks on medium and large lattices (32^3 to 64^3). These calculations are used to scan the physical system over parameters under study (for example, application of electric field) and find stable and metastable states of the system. Precise calculations consist of tens of tasks on large to extra large lattices (from 64^3 to 256^3). These calculations are used to re-evaluate the energy of the system with higher precision in a certain state.

It should be noted that each optimization task consists of many similar independent runs by the nature of stochastic optimization. Usually, each task requires from 4 to 10 runs; however, in some situations up to 100 runs may be required (for example, when the frustration of the system between ground and metastable states is studied).

4. Typical Computational Kernel of the Program Package

At the first stage of this research, we selected a primary computational kernel (device function) of the evaluated program, which has two important characteristics. First, this kernel performs the largest part of computations among other GPU activities in the evaluated program. Second, this kernel utilizes memory access patterns and computational workflow similar to other important kernels of the program package. Thus, it is reasonable to apply optimizations and evaluate their effects on this specific kernel, and later to generalize and apply them for the remaining kernels. This kernel will be further referred to as "typical computational kernel" (TCK).

TCK produces a trial change in the director field $\bar{n}(\bar{r})$ (2D simplification is shown as color bars in Fig. 2a), then calculates the difference in free energy of interaction between LC and external electric field ($F_{ext} = const \times \int_V \left[(\bar{n}(\bar{r}) \cdot \bar{E}(\bar{r}))\right] d\bar{r}$) between the new state (trial) and the previous one, and finally accepts or declines each of these changes (by Metropolis algorithm). Here V is the droplet volume and $E(\bar{r})$ is external electric field distribution. Green circles in Fig. 2a denote the points in which the director field is changed by the trial (so-called pivot points). This procedure seeks for the optimal distribution of director field $\bar{n}(\bar{r})$ which delivers a minimum to F_{ext}, when Monte Carlo is coupled with simulated annealing. According to the formulation of free energy, we use sparse 3D placement of pivot points to implement checkerboard decomposition technique. It allows processing these points in parallel, thus producing multiple independent local MCMC trials at one step.

Figure 2b illustrates the placement of the stored data in 2D simplification. Solid green squares represent pivot points processed simultaneously. During the next step, the set of pivot points will be shifted. Eight steps (also called ticks) cover the full 3D lattice, shown in Fig. 2c. On each step, the tick vector shows the current shift of the first pivot point from the bottom-most point of the lattice. The tick vector varies from $(0, 0, 0)$ to $(1, 1, 1)$ and cyclically increments from step to step. Thus, the most of device kernels are launched on a cubic CUDA grid of dimensions twice smaller than corresponding (physical) lattice size. A technical layer (blue squares in Fig. 2b) was introduced to make calculations on the border equivalent to the one in the inner part of the lattice. The zero contribution of technical layer to the total energy is granted by zero volumetric coefficients V_i in corresponding cells.

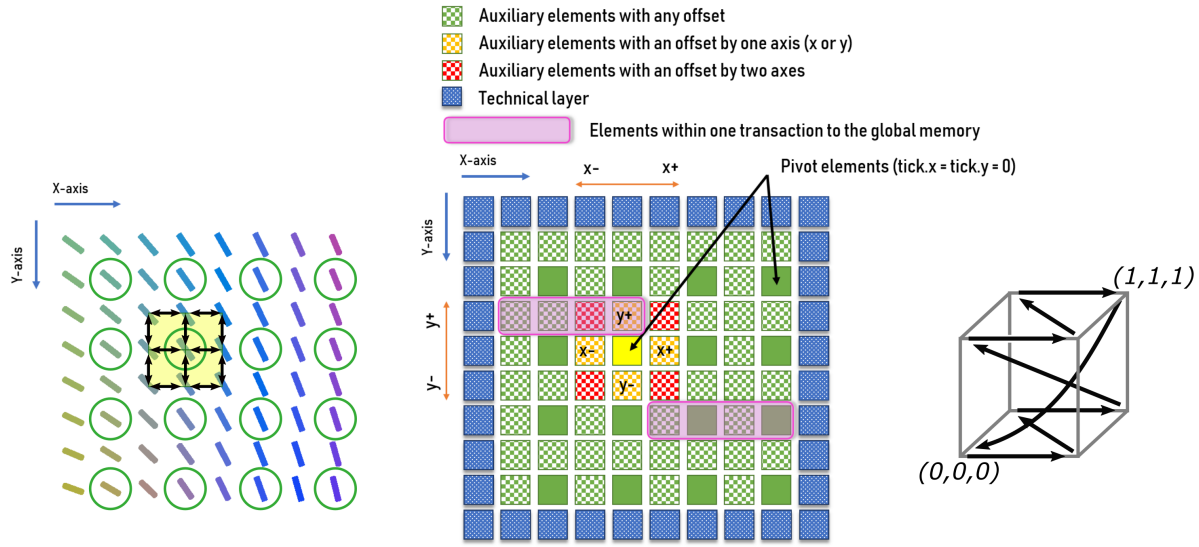


Figure 2. (a) 2D simplification of director field on a cubic lattice. Green circles show pivot points, to which trial changes of director vectors are applied in parallel. (b) Lattice configuration of the TCK (2D simplification). Pivot elements (tick.x = tick.y = 0) are solid squares on the picture, auxiliary elements (tick.x = 1 or tick.y = 1) are marked as checkered squares. Transactions to global memory from a warp of adjacent threads are shown as magenta rectangles. (c) Sequence of shifts of pivot points in 3D

In the implemented numerical scheme, the integral over droplet volume V is replaced with the sum over lattice cells (shown as yellow boxes in Fig. 2a): $F_{ext} = const \times \sum_i \left[\left\langle(\bar{n}(\bar{r}) \cdot \bar{E}(\bar{r}))\right\rangle_i \times V_i\right]$, where V_i is the volume of i-th cell, and $\langle ... \rangle_i$ is averaging over i-th cell. For better scheme convergence, the value $\left\langle(\bar{n}(\bar{r}) \cdot \bar{E}(\bar{r}))\right\rangle_i$ is averaged by 26 points per cell: cell corners (i.e., lattice points), middles of cell edges (so-called secondary lattice points), and middles of cell facets (so-called tertiary lattice points). Thus, the calculations of the energy change require not only the pivot point, but also neighbor points (illustrated by black arrows in Fig. 2a). It makes the TCK computations heavily memory-intensive.

The software implementation of the TCK consists of three major parts. In the first part of the typical kernel, each CUDA thread calculates indexes, used to access pivots and adjacent lattice elements during the following calculation. Since this kernel utilizes the lattice approach, each CUDA thread operates with a specific pivot element of the three-dimensional lattice and its neighborhood. Thus, 3^3 indexes are calculated, including the pivot (central) one. These indexes are stored on registers of streaming multiprocessors. Figure 2 shows a simplified mapping scheme of the relation between CUDA threads and data arrays processed by a kernel.

In the second part of the TCK, the data describing the state before the change in the director field is copied into separate arrays. This operation is necessary since the trial change can be declined by Metropolis algorithm, and then the previous state should be restored.

In the third part, the kernel calculates the values of F_{ext} for the trial director configuration. To finalize MCMC step, the Metropolis algorithm should be applied, and trial change of the director in each pivot point should be accepted or declined probabilistically.

Contribution of the TCK into the whole program package runtime can be evaluated using nvprof profiling tool. This contribution slightly changes when processing different physical problems, for example different lattice sizes or the requirement to process the surface of the droplet.

Our measurments demonstrated that the runtime contribution of the TCK varies between 12 % and 20 % for most of the tests.

5. Co-design of the Typical Kernel of the Evaluated Software Package

5.1. Detecting Primary Optimization Techniques

At first it is necessary to highlight primary optimization techniques, which can be implemented to increase the performance of the TCK. As mentioned in the previous section, the TCK shares various computational properties and features with a vast majority of other kernels of the evaluated program package, and thus the proposed optimizations can be later easily applied to the whole program package.

First of all we optimized the TCK for the NVIDIA GPU architecture, since it provides an extremely convenient profiling toolkit, which allows easily determining performance issues and bottlenecks of CUDA programs. In this paper we used nvprof and nvvp profiling tools in order to collect various dynamic characteristics of the program, which are necessary in the process of the optimization.

The dynamic characteristics and therefore performance and efficiency of the investigated kernel significantly depend on the grid size, as shown in Tab. 2.

Table 2. Utilization of P100 GPU resources of the TCK for various problem sizes

Grid parameters	Compute utilization	Memory utilization
128^3	10 %	55 %
64^3	10 %	35 %
16^3	15 %	25 %

Table 2 shows that a significant part of kernel runtime is spent on loading information from various levels of GPU memory (device and caches), and thus memory subsystem bandwidth is the primary performance bottleneck. Kernels with memory utilization ratio higher than 50 % are usually called *memory-bound*. Since the TCK launched on the large grids (128^3) is memory bound, its interaction with memory subsystem needs to be carefully investigated and optimized.

Memory subsystem usage of any kernel highly depends on memory access patterns, which can be characterized via efficiency of GPU-transactions. GPU-transaction is a process of loading continuous chunk of data (usually 128 bytes) from memory subsystem. The efficiency of the transaction can be estimated as the amount of useful data loaded from memory divided by the transaction size. When the kernel uses a sequential memory access pattern, transactions efficiency of such kernel is equal to 100 % in the case when the initial transaction addresses are 128-byte aligned. In the worst case (random memory access pattern), each GPU warp needs to generate the amount of transactions equal to the size of warp, and the transaction efficiency for such program is roughly equal to 3 %. Figure 2b highlights two transactions for the TCK: the first one (1) loads elements within vertical offset (by Y-axis), while the second one (2) loads pivot elements and simultaneously prefetches elements within horizontal offset (by X-axis) into L1 cache.

To assist developers in calculating the transaction efficiency of the specific kernel, special gld_efficiency and gst_efficiency hardware metrics of nvprof tool can be used. Elements of the physical lattice are sparsely located in the data arrays, for example pivot elements alternate with horizontal offset elements, as shown in Fig. 2. Hence, if the warp loads pivot elements from the global memory, the transactions will contain both pivots and elements within X-axis offsets, and the latter being redundant to load, resulting in the efficiency of such transactions being twice lower compared to the case when all required elements are stored densely.

Transaction efficiency also depends on the type of the requested elements. In the original program, the director field direction in each lattice point is stored in three double precision variables. Thus unrolling arrays of structures (AoS) into structures of arrays (SoA) allows us to further increase transaction efficiency and significantly accelerate the investigated kernel.

Despite the fact that transactions to global memory in the TCK have low efficiency, memory access pattern of the kernel has an important advantage. Elements, which neighbor pivots and fall into a requested transaction when pivots are loaded (shaded purple-colored elements in Fig. 2) are prefetched into L1 GPU cache, resulting into further transactions to these elements being processed significantly faster, since they load the required data from caches instead of global memory. Due to the fact that a significant amount of elements neighboring pivots reside in L1 cache (L1 hit rate of the TCK is 72 %), only a small number of transactions will be directed to L2 cache or device memory.

Another important feature of every CUDA kernel is occupancy. Occupancy is a ratio of active warps resident on a single streaming multiprocessor to a maximum theoretical value of active warps supported by a single SM. Occupancy of the TCK launched on the smallest computational grid of size 16 is very low – only 64 blocks of size 512 are distributed over 56 SMs, while each SM allows processing up to 4 blocks of similar size simultaneously. Such small number of blocks can not fully utilize GPU resources, for example global memory bandwidth. On the other hand, when the kernel is launched on the largest grid (128^3), it fully occupies each SM with 512 blocks launched in total. Occupancy for the TCK launched on small grids can be increased using multitasking, when two or more independent CUDA-kernels are concurrently launched on a single GPU.

Occupancy of the investigated kernel is also limited by the number of registers available on a single SM. Registers are heavily used in the TCK to store both indexes and intermediate elements of physical lattice, which are accessed multiple times in different parts of the kernel. Thus, using compiler directives to limit a number of registers used by each thread is another important direction of further optimization.

5.2. Optimizations for NVIDIA GPU Architecture

Implementing a coalesced memory access pattern to global memory. Figure 2 and subsection 5.1 explain why the existing memory access pattern used in the initial version of the TCK has low efficiency of load transactions. To avoid this problem, the lattice storage configuration has been changed in the following way: the lattice was split into 8 independent parts, each corresponding to a particular shift in 3D-version of checkerboard algorithm (value of the *tick* variable), as shown in Fig. 2c. Figure 3 demonstrates a simplified 2D version of storage splitting scheme, where checkerboard algorithm supposes the use of four ticks. Here, on each tick, all pivot points are stored in the same part of lattice. This way load transactions to elements in different parts of lattice do not contain gaps filled with elements with another offset. As a

result, memory accesses to elements within different offsets will be coalesced; such optimization approximately doubles the efficiency of load and store transactions for the investigated kernel, according to the results from the Tab. 3. Since the described optimization changes the structure of the kernel significantly, all further optimizations in this paper will be applied to this kernel with this new memory access pattern.

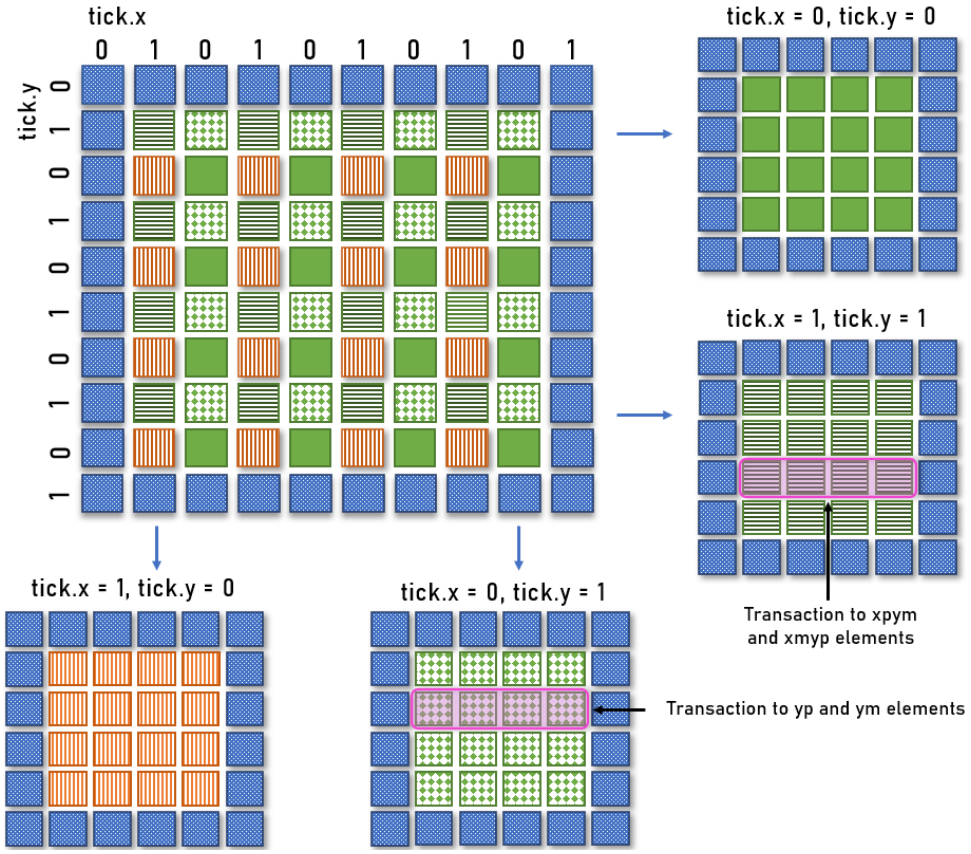


Figure 3. Lattice configuration in the investigated kernel (2D simplification). Pivot elements (tick.x=tick.y=0) are solid squares on the picture, auxiliary elements (tick.x $\neq$ 0 or tick.y $\neq$ 0) are marked as checkered squares. Transactions to global memory from a warp of adjacent threads are shown as rectangles

Table 3. The comparison of transaction efficiency for different memory access patterns

	Initial memory access pattern	**Improved memory access pattern**
Store transactions efficiency	34 %	80 %
Load transactions efficiency	43 %	70 %

Using shared memory. Shared memory of NVIDIA GPUs is a hand-driven L1 cache. Stencil kernels can benefit a lot from using this memory, since inner neighbor threads in a warp generate a transaction with highly reused elements, and with the use of shared memory we can avoid sending more than 1 transaction to global memory and do not care if required elements reside in non-programmable L1 cache. Typically each thread copies its central element from global

to shared memory, allowing other threads to access this element with lower latency and higher bandwidth.

However, kernels of optimized package (and thus TCK) are not pure stencils and perform other operations, such as copying energies from previous step and making a Monte-Carlo decision. These parts can not benefit from using shared memory, since they are based on typical sequential memory access patterns and are memory-bound. This is the first problem of using shared-memory based optimizations in the kernels of our package: due to Amdal's law, the obtained acceleration will be much lower compared to the cases of usual stencil codes [25].

When applying shared memory optimizations to the stencil part of TCK, we face another problem: it uses a huge amount of input arrays (around 20 float arrays for TCK), required for stencil computations. Certainly, all these arrays can not be stored in shared memory due to its limited size of 64KB or 96KB per one SM. A possible solution can be the usage of registers [25], but according to the nvprof the majority of them is already used for storing other intermediate data.

A possible solution involves storing arrays in shared memory step-by-step, implementing a sort of pipeline. Unfortunately, computational-intensive functions operate only with 2 elements of stencil radius at once to calculate parts of each pivot element, which results in low data reuse. In addition, the implementation of such pipeline requires frequent thread synchronization. For these reasons shared memory optimizations can not be applied to TCK and other kernels of the package, which is confirmed by our implementation experiments.

Changing precision from *double* to *single*. Since the initial kernel launched on a large computational grid is memory-bound, it is reasonable to try to change the type of lattice elements from double to float in order to significantly reduce the amount of loaded data. We determined the critical parts of the code, where changing the precision from double to float affected the results. It corresponds to the parts where exponential functions are taken, and where reduction over large number of values is produced. It required to remain about 1 % of the variables in double precision. The other variables were switched to float. After this optimization the amount of load transactions is 1.3 times lower for single precision compared to double precision (30.7M transactions for double and 23.2M transactions for float), which is proportional to the acceleration obtained by this optimization. It is important to notice that physical results of calculations are correct when obtained via both single and double precision.

Unrolling array of structures into structure of arrays. Unrolling array of structures (AoS) into structure of arrays (SoA) allows us to further reduce the amount of memory transactions. With this optimization applied, the ratio of the requested data to the required data loaded from the global memory has doubled – from 24 % to 46 %. However, this optimization did not lead to the proportional acceleration, since this optimization also decreases the efficiency of using L1 cache: the accesses to fields of structures occur in the TCK close to each other, and thus the remaining fields are typically prefetched into L1 cache in the case of using AoS.

Decreasing the transaction size. NVIDIA GPUs allow turning off L1 cache in order to decrease the size of memory transaction and thus improving the effective bandwidth for non-coalesced memory access pattern. When L1 cache is disabled, the transaction has 4 times larger size (128 against 32 bytes). In this paper we evaluated kernel performance with L1 cache both

turned on and turned off, and the experiments demonstrated a 15 % speedup with L1 cache turned on.

Increasing the occupancy by limiting the register usage. One of the reasons why occupancy of the investigated kernel is low is that each CUDA thread uses many GPU registers – approximately 70 of them, and consequently a block of 512 threads requires 35840 registers. Each streaming multiprocessor of P100 GPU has only 32768 registers available, which does not allow running two or more blocks concurrently on the same SM. The *launch_bounds (maxThreadsPerBlock, minBlocksPerMultiprocessor)* directive can be used to limit the amount of registers utilized by a single CUDA-block, which allows running up to 4 blocks of size 512 on a single P100 SM concurrently (since there is also a hardware thread limit of 2048 threads per SM). However, applying this directive with a parameter *minBlocksPerMultiprocessor=4* turns the kernel into being more memory-bound, since the ratio of memory operations among the whole kernel increases from 35 % to 55 % and does not lead to any significant acceleration.

Applying multitasking for small grids. Occupancy values are significantly higher for large computational grids, since the amount of CUDA threads launched for such grids is also high. Insufficient number of threads launched on small grids decreases the efficiency of using available throughput of GPU memory. The achieved global memory throughput can be calculated as a sum of two nvprof metrics – gld_throughput and gst_throughput. On computational grids of size 64^3 this sum is approximately equal to 420 GB/s and remains the same for all larger grids, thus this value can be viewed as an achievable limit for the investigated kernel on a P100 GPU. This value is also relatively close to the achieved throughput of 540 GB/s on Stream benchmark [4], which also has a coalesced access pattern, but transactions of which are aligned (unlike the TCK).

For smaller grids the investigated kernel achieves throughput equal to 295 GB/s, which is 1.4 times lower compared to throughput on large grids. This gap can be eliminated by using multitasking: multiple independent instances of kernel can be launched in parallel on a single GPU, thus utilizing its hardware resources more efficiently. This way GPU occupancy is increased, which in turn allows us to hide latency. We implemented multitasking using OpenMP directives and CUDA Streams. Table 4 shows that an expected 1.4 times acceleration has been achieved.

Table 4. Theoretical acceleration which can be achieved by launching independent kernels (tasks) in parallel

	Single task	Multiple tasks	Theoretical speedup
gst+gld throughput on 64^3 lattice	975.1 GB/s	1375.82 GB/s	1.41
gst+gld throughput on 128^3 lattice	321.414 GB/s	474.97 GB/s	1.47

The overall comparison of effects from applying different types of optimizations to the TCK (launched on NVIDIA P100 GPU) is shown in Fig. 5.

5.3. Porting the Typical Kernel to NEC SX-Aurora TSUBASA

Since vector architectures and GPUs have a significant number of similar architectural and computing features [1], the TCK can be ported to the NEC SX-Aurora TSUBASA architecture

Table 5. The comparison of bandwidth (BW) and execution time values for different versions of the TCK launched on NVIDIA P100 GPU

lattice size	initial version	new pattern	AoS to SoA	register usage	changed precision	multi-tasking
64^3(BW)	319.87 GB/s	551.78 GB/s	569.05 GB/s	576.60 GB/s	409.76 GB/s	594.84 GB/s
64^3(time)	319.22 s	172.17 s	179.21 s	171.52 s	126.75 s	87.35 s
128^3(BW)	479.18 GB/s	710.67 GB/s	716.16 GB/s	558.38 GB/s	460.24 GB/s	754.21 GB/s
128^3(time)	678.1 s	457.20 s	453.7 s	581.9 s	352.98 s	339.43 s

in a relatively straightforward way. The investigated CUDA-kernel is transformed into a 3-dimensional nested loop; at each iteration of the innermost loop the same program code is executed by each of CUDA threads. Since the initial version of the kernel does not use any architecture-dependent features of the GPUs (such as shared memory, special instructions, etc.), the program code can be used on NEC SX-Aurora TSUBASA architecture without any changes. The number of iterations inside each nested loop is equal to the size of the CUDA grid in one dimension (x, y, z), and the innermost loop corresponds to X dimension of CUDA grid. This means that the innermost loop is used to process X-axis elements of lattice shown in Fig. 2. Since all iterations inside each of the nested loop are independent, computations inside the loops can be parallelized and vectorized in several different ways. NEC SX-Aurora TSUBASA architecture does not allow efficient loading information from arrays of structures. Thus, the initial version of the program was implemented with coordinate data structures unrolled into three separate arrays.

The following subsections describe the most important optimizations required to obtain a high-performance version of the TCK for NEC SX-Aurora TSUBASA vector engines. Since the investigated computational kernel is memory-bound (as demonstrated in the previous sections for GPUs), the sustained memory bandwidth values will be used during its performance evaluation. The speedup achieved by implementing each of the further discussed optimizations is listed in Tab. 7 in the end of this section.

Parallelization and vectorization of the program. When porting algorithms consisting of multiple nested loops to vector architectures, the innermost loop is usually a subject to the vectorization. According to the previously discussed kernel transformation, the innermost loop processes adjacent lattice elements located in adjacent cells, which allows vector instructions to follow a relatively efficient memory access pattern. The outer loop is parallelized using OpenMP *#pragma omp parallel for* clause and *schedule (static)* mode for distributing iterations.

Improving memory access pattern. Since the investigated kernel is memory-bound, vector instructions should have a specific vector-friendly memory access pattern. For the NEC SX-Aurora TSUBASA architecture, the linear dependence between the loop iteration indexes and the indexes of the accessed arrays is the necessary condition: all arrays indexes should be represented as $i+offset$, where i is the index of the innermost (vectorized) loop. If this condition is satisfied, then vector LOAD and STORE instructions are used, otherwise – GATHER and SCATTER instructions, which are significantly less efficient. When vectorizing the initial version of the kernel, memory access indexes can be represented as $i*2+tick+1$, as illustrated in Fig. 2. This pattern leads to the usage of GATHER and SCATTER instructions which significantly

reduces the performance of the kernel. Thus, proposed in section 5.2 data-layout transformation is required for the NEC SX-Aurora TSUBASA architecture.

Collapsing nested loops. Vectorizing only the innermost loop has a significant downside: in the case of small lattices, vector instructions have relatively low length (16–32 compared to the desirable value of 256, equal to the maximum vector length of NEC SX-Aurora TSUBASA architecture), since the innermost loop does not have enough iterations to be vectorized with instructions of maximum length. However, all three-dimensional loops can be collapsed (merged) into a single linear loop, which typically has enough iterations for simultaneous vectorization and parallelization. This optimization also allowed us to achieve a significant speedup as shown in Tab. 7.

For discussed optimizations, Tab. 6 demonstrates the comparison of three important metrics: average vector length, vector operation ratio and and last level cache hit rate. These values are received with special tool for NEC Vector architectures called Ftrace.

Table 6. Performance metrics for different versions of the TCK, implemented for the NEC SX-Aurora TSUBASA architecture

metric	initial version	vectorisation and parallelisation	improved memory access pattern	loops collapsed
average vector length	1	33.9	249.6	249.6
vector operation ratio	0 %	99 %	99 %	99 %
LLC error rate	N/A	86 %	77 %	58 %

Using multitasking for small lattices. Despite applying loop collapse optimization, the performance of the developed kernel on small lattices is still limited by the insufficient amount of computational work. Thus, in such cases multitasking optimization can be implemented, similar as for NVIDIA GPUs: different kernel runs are executed on independent vector cores of SX-Aurora, while each kernel is vectorized, but not parallelized. However, the acceleration obtained by this optimization is significantly lower compared to GPUs. The number of runs in parallel affects neither the average length of a vector operation, nor the ratio of vector operations in a kernel and LLC hit rate, so obtained values by Ftrace in Tab.6 are sufficient.

Converting precision from *double* to *single*. Same as for NVIDIA GPUs, changing the kernel to operate with single precision instead of double allows halving the amount of data loaded from memory during kernel execution. However, in case of NEC SX-Aurora TSUBASA, LOAD and STORE vector instructions for double precision are capable of loading twice the amount of data compared to single precision, in approximately the same time. This causes a relatively low speedup when applying this optimization. As the number of elements and strategy of loading elements from memory remains the same as for previous optimizations, we do not need to update Tab.6 for such optimization.

The overall comparison of effects from applying different types of optimizations to the TCK (launched on NEC SX-Aurora TSUBASA) is shown in Fig. 7.

Table 7. The comparison of the sustained bandwidth(BW) and the execution time for different versions of the TCK ported to the NEC SX-Aurora TSUBASA architecture

lattice size	initial version	vectorisation and parallelisation	improved memory access pattern	loops collapsed	multi-tasking	double to float precision
32^3(BW)	0.35 GB/s	38 GB/s	63 GB/s	380 GB/s	391 GB/s	506 GB/s
32^3(time)	-	3.33 s	2.00 s	0.33 s	0.32 s	0.25 s
64^3(BW)	0.28 GB/s	53 GB/s	129 GB/s	537 GB/s	576 GB/s	638 GB/s
64^3(time)	-	18.9 s	7.8 s	1.8 s	1.7 s	1.5 s
128^3(BW)	0.28 GB/s	89 GB/s	249 GB/s	615 GB/s	625 GB/s	673 GB/s
128^3(time)	-	91.2 s	32.5 s	13.2 s	12.9 s	12.0 s
256^3(BW)	0.28 GB/s	134 GB/s	395 GB/s	657 GB/s	690 GB/s	725 GB/s
256^3(time)	-	482 s	164 s	98 s	95 s	90 s

5.4. Porting the Typical Kernel to Intel Xeon Architecture

Developing CPU version of the TCK is important, since the performance comparison between CPUs and GPUs (or NEC SX-Aurora TSUBASA) is interesting for many researches. Frequently, they need to understand if porting their programs or packages to new architectures is justified. Thus, readers of our paper may be able to obtain this knowledge, in the case when their programs are based on similar stencil schemes or mathematical algorithms, described in sections 2–4.

The performance of the developed CPU version was evaluated on the node of Lomonosov-2 supercomputer, which is equipped with Intel Xeon Gold 6126 processors.

Since OpenMP programming model can be efficiently used for modern Intel Xeon CPUs, the initial version of the typical computational kernel can be obtained in the same way as for the NEC SX-Aurora TSUBASA architecture, as described in section 5.3. Thus, when porting TCK to Intel Xeon, three-dimensional nested loop has been collapsed into one big loop, since such transformation allows achieving better parallelization and using AVX-512 vector instructions of Intel CPUs.

Next, we have studied what compiler shows better results for our application. We compared classical GNU gcc compiler to the Intel compiler (icpc), using the same flags in both cases. By choosing icpc, we managed to obtain a constant 3–4 % speedup, depending on a lattice size.

After that we studied the efficiency of OpenMP parallel instructions, applied to the investigated kernel. Using Intel VTune we have investigated that the whole CPU utilization of parallel version is 87 %, with the ratio of serial instructions below 1 %. That makes OpenMP directives quite relevant for investigated kernel.

Furthermore, we implemented a new memory access pattern, described above. The old version of a kernel had a small ratio of retired micro-ops (which shows the fraction of time processor was fully utilized by useful work), while ∼80 % of micro-ops were stalled waiting for data from DRAM (i.e., back-end bound stalls). New memory access pattern increased the ratio of retired instructions to 45 %, making the ratio of back-end bound micro-ops equal to 17 %. Finally, a vectorization of investigated kernel was studied. By placing *#pragma simd* directive and *-qoverride-limits* with *-qopt-zmm-usage=high* compiler flags, we managed to obtain 1.5 faster version of typical computational kernel.

Thus, the developed CPU version demonstrates high utilization of hardware resources, is efficiently parallelized and vectorized, and therefore in our opinion can be used for comparison with other two evaluated platforms.

6. Comparison of the Typical Kernel Performance on Different Architectures

After optimizing the TCK on different platforms, we cross-compared the behaviour of each optimization on each platform. Contributions of individual optimizations on each platform are given in Tabs. 5, 7, and sec. 5.4.

Multitasking did not lead to any significant acceleration on NEC SX-Aurora TSUBASA and Intel Xeon architectures, since they both have only a few cores (12 and 8 respectively). It can be efficiently occupied with calculations even by processing small lattices. Thus, multitasking of the studied program was found to be unnecessary on these platforms. At the same time, NVIDIA GPUs have much higher number of cores, which were hard to fully utilize without multitasking. Thus implementation of multitasking on NVIDIA GPUs resulted in a significant performance increase. Moreover, newer GPU microarchitecture demonstrated higher acceleration from multitasking optimization, since the resource of hardware parallelism is constantly growing with each generation of GPU.

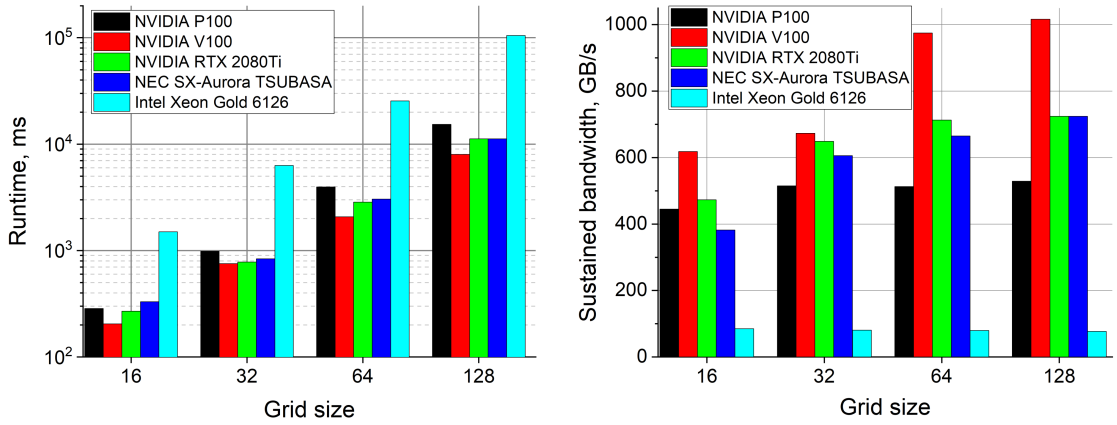
Improving memory access pattern (eliminating arrays of structures and splitting the lattice) was found to be absolutely crucial for the NEC SX-Aurora TSUBASA architecture, since using LOAD and STORE instructions leads to 3–4 times acceleration. The same optimization is not determinative for NVIDIA GPUs, because the initial memory access pattern provides higher utilization of L1 cache. Still, the new memory access pattern reduces runtime of the TCK by 1.5–2 times. For Intel Xeon processors, the new memory access pattern increases the number of retired micro-ops and thus reduced a kernel runtime by 1.5 times.

Changing precision demonstrates the highest acceleration (1.35 times) on NVIDIA GPUs, while practically does not speed up calculations on NEC SX-Aurora TSUBASA architecture due to the implementation details of LOAD instructions in it.

Figure 4 shows the overall comparison of the TCK performance on target platforms. These data provide the runtime and the sustained bandwidth values for most optimized versions of the TCK on each architecture. The sustained bandwidth values for all target architectures are proportional to the theoretical peak bandwidth values. The highest bandwidth (and therefore lower runtime) is achieved on NVIDIA V100 GPU architecture. NVIDIA RTX 2080 Ti GPU performs slightly better than NEC on small and medium grids, but on large grids their bandwidths and runtimes are equal. NVIDIA P100 GPU demonstrated results 15 %–40 % below NVIDIA RTX 2080 Ti GPU and NEC SX-Aurora TSUBASA. In comparison to these architectures, Intel Xeon Gold 6126 bandwidth are strikingly lower, and runtime is larger, accordingly.

Conclusions

In this paper, we investigated and improved the computational efficiency of simulations software for liquid crystals on various modern supercomputing architectures. The computational problem in this software belongs to the class of stochastic optimization of a functional defined on finite space cubic lattice. Namely, the solver is based on Markov chain Monte Carlo with Metropolis algorithm, paralleled by sparse checkerboard decomposition. The following plat-



(a) (b)

Figure 4. The comparison of (a) runtime (in logarithmic scale) and (b) sustained bandwidth values of the TCK on different platforms

forms were used: NVIDIA GPU (Pascal, Volta and Turing microarchitectures), NEC SX-Aurora TSUBASA vector engines, and Intel Xeon Gold 6126 processors.

We studied and compared the efficiency of multiple optimization strategies for this software on each platform. These included the usage of more suitable memory access patterns, implementation of multitasking for efficient utilization of massive parallelism of the platforms, improving cache hit-rates, parallel workload balancing and several others.

As a result of the provided research, evaluated platforms can be ranked by the ability to efficiently solve this discussed class of problems as follows: (1) NVIDIA V100 GPUs, (2) NEC SX-Aurora TSUBASA, (3) NVIDIA RTX 2080 Ti GPUs, (4) NVIDIA P100 GPUs, (5) Intel Xeon Gold 6126 CPUs. It should be noted that NVIDIA GPUs and NEC SX-Aurora TSUBASA showed roughly equal efficiency and performance for this type of computational problems, while Intel Xeon processors demonstrated significantly lower performance for it.

The optimization techniques demonstrated above are useful not only to the particular program of liquid crystals simulations software. Instead, we believe it can be applied to a wide range of computational problems dealing with Markov chain Monte Carlo simulations on finite space cubic lattice, including ensemble simulations, aim search, stochastic optimization and other techniques, aimed to solve problems in mathematics, computational physics, chemistry and biology, economics and multidisciplinary studies.

Acknowledgments

The reported study was funded by the Russian Foundation for Basic Research, project number 20-37-70036. The work presented in section 5.3 is supported by the Russian Ministry of Science and Higher Education, agreement No. 075-15-2019-1621. The research is carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University.

References

1. Afanasyev, I.V., Voevodin, V.V., Voevodin, V.V., et al.: Analysis of Relationship Between SIMD-Processing Features Used in NVIDIA GPUs and NEC SX-Aurora TSUBASA Vector Processors. In: Parallel Computing Technologies - 15th International Conference, PaCT 2019, Proceedings. Lecture Notes in Computer Science, vol. 11657, pp. 125–139. Springer (2019). https://doi.org/10.1007/978-3-030-25636-4_10

2. Barbosa, C.H., Kunstmann, L.N., Silva, R.M., et al.: A workflow for seismic imaging with quantified uncertainty. Computers & Geosciences 145, 104615 (2020). https://doi.org/10.1016/j.cageo.2020.104615

3. Baxter, R.: The inversion relation method for some two-dimensional exactly solved models in lattice statistics. Journal of Statistical Physics 28(1), 1–41 (1982). https://doi.org/10.1007/BF01011621

4. Bergstrom, L.: Measuring NUMA effects with the STREAM benchmark. CoRR abs/1103.3225 (2011), http://arxiv.org/abs/1103.3225

5. Block, B., Virnau, P., Preis, T.: Multi-GPU accelerated multi-spin Monte Carlo simulations of the 2D Ising model. Computer Physics Communications 181(9), 1549–1556 (2010). https://doi.org/10.1016/j.cpc.2010.05.005

6. Boroni, G., Dottori, J., Rinaldi, P.: Full GPU implementation of Lattice-Boltzmann methods with Immersed Boundary Conditions for Fast Fluid Simulations. The International Journal of Multiphysics 11(1), 1–14 (2017). https://doi.org/10.21152/1750-9548.11.1.1

7. Dudzin-acuteski, M., Sznajd, J.: Suzuki-Trotter decomposition and renormalization of a transverse-field Ising model in two dimensions. Phys. Rev. B 55(22), 14948–14952 (1997). https://doi.org/10.1103/PhysRevB.55.14948

8. Egawa, R., Komatsu, K., Momose, S., et al.: Potential of a modern vector supercomputer for practical applications: performance evaluation of SX-ACE. The Journal of Supercomputing 73(9), 3948–3976 (2017). https://doi.org/10.1007/s11227-017-1993-y

9. Fang, Y., Feng, S., Tam, K.M., et al.: Parallel tempering simulation of the three-dimensional Edwards–Anderson model with compact asynchronous multispin coding on GPU. Computer Physics Communications 185(10), 2467–2478 (2014). https://doi.org/10.1016/j.cpc.2014.05.020

10. Floudas, C.A., Pardalos, P.M.: Encyclopedia of Optimization. Springer Science+Buisiness Media, LLC. (2009), https://www.springer.com/gp/book/9780387747583

11. Geng, Y., Noh, J., Drevensek-Olenik, I., et al.: High-fidelity spherical cholesteric liquid crystal Bragg reflectors generating unclonable patterns for secure authentication. Scientific Reports 6(1) (2016). https://doi.org/10.1038/srep26840

12. Goodby, J.W., Tschierske, C., Raynes, P., et al. (eds.): Handbook of Liquid Crystals. Wiley-VCH Verlag GmbH & Co. KGaA (2014). https://doi.org/10.1002/9783527671403

13. Gourgoulias, K., Katsoulakis, M.A., Rey-Bellet, L.: Information criteria for quantifying loss of reversibility in parallelized KMC. Journal of Computational Physics 328, 438–454 (2017). https://doi.org/10.1016/j.jcp.2016.10.031

14. Gourgoulias, K., Katsoulakis, M.A., Rey-Bellet, L.: Information metrics for long-time errors in splitting schemes for stochastic dynamics and parallel kinetic Monte Carlo. SIAM Journal on Scientific Computing 38(6), A3808–A3832 (2016). https://doi.org/10.1137/15m1047271

15. Harju, A., Siro, T., Canova, F.F., et al.: Computational physics on graphics processing units. In: Applied Parallel and Scientific Computing - 11th International Conference, PARA 2012, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7782, pp. 3–26. Springer (2012). https://doi.org/10.1007/978-3-642-36803-5_1

16. Huth, B., Meyer, N., Wettig, T.: Lattice QCD on a novel vector architecture. CoRR abs/2001.07557 (2020), https://arxiv.org/abs/2001.07557

17. Kapitan, V.Y., Nefedev, K.V.: High performance calculation of magnetic properties and simulation of nonequilibrium phenomena in nanofilms. In: Modeling, Simulation and Optimization of Complex Processes - HPSC 2012, pp. 95–107. Springer (2014). https://doi.org/10.1007/978-3-319-09063-4_8

18. Khan, M., Li, W., Mao, S., et al.: Real-time imaging of ammonia release from single live cells via liquid crystal droplets immobilized on the cell membrane. Advanced Science 6(20), 1900778 (2019). https://doi.org/10.1002/advs.201900778

19. Komatsu, K., Momose, S., Isobe, Y., et al.: Performance evaluation of a vector supercomputer SX-Aurora TSUBASA. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis. pp. 54:1–54:12. SC '18, IEEE, Piscataway, NJ, USA (2018). https://doi.org/10.1109/SC.2018.00057

20. Krakhalev, M.N., Rudyak, V.Yu., Prishchepa, O.O., et al.: Orientational structures in cholesteric droplets with homeotropic surface anchoring. Soft Matter 15(28), 5554–5561 (2019). https://doi.org/10.1039/c9sm00384c

21. Lagerwall, J.P., Scalia, G.: A new era for liquid crystal research: Applications of liquid crystals in soft matter nano-, bio- and microtechnology. Current Applied Physics 12(6), 1387–1412 (2012). https://doi.org/10.1016/j.cap.2012.03.019

22. Larsen, T., Bjarklev, A., Hermann, D., Broeng, J.: Optical devices based on liquid crystal photonic bandgap fibres. Optics Express 11(20), 2589 (2003). https://doi.org/10.1364/oe.11.002589

23. Li, W., Fan, Z., Wei, X., Kaufman, A.: GPU-Based flow simulation with complex boundaries. Tech. rep. (2003)

24. Maltseva, D., Zablotskiy, S., Martemyanova, J., et al.: Diagrams of states of single flexible-semiflexible multi-block copolymer chains: A flat-histogram Monte Carlo study. Polymers 11(5) (2019). https://doi.org/10.3390/polym11050757

25. Maruyama, N., Nomura, T., Sato, K., Matsuoka, S.: Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In: Conference on High Performance Computing Networking, Storage and Analysis, SC 2011. pp. 11:1–11:12. ACM (2011). https://doi.org/10.1145/2063384.2063398

26. Otsuka, Y., Seo, H., Motome, Y., Kato, T.: Finite-temperature phase diagram of quasi-one-dimensional molecular conductors: Quantum Monte Carlo study. Journal of the Physical Society of Japan 77(11), 113705 (2008). https://doi.org/10.1143/jpsj.77.113705

27. Peng, B., Li, J., Akkas, S., et al.: Rank position forecasting in car racing. In: 35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17-21, 2021. pp. 724–733. IEEE (2021). https://doi.org/10.1109/IPDPS49936.2021.00082

28. Phillips, J.C., Hardy, D.J., Maia, J.D.C., et al.: Scalable molecular dynamics on CPU and GPU architectures with NAMD. The Journal of Chemical Physics 153(4), 044130 (2020). https://doi.org/10.1063/5.0014475

29. Raedt, H.D., Lagendijk, A.: Monte Carlo simulation of quantum statistical lattice models. Physics Reports 127(4), 233–307 (1985). https://doi.org/10.1016/0370-1573(85)90044-4

30. Rao, S.S.: Engineering Optimization: Theory and Practice. John Wiley & Sons, Inc. (2009), https://www.wiley.com/en-us/Engineering+Optimization%3A+Theory+and+Practice%2C+5th+Edition-p-9781119454793

31. Rathore, N., de Pablo, J.J.: Monte Carlo simulation of proteins through a random walk in energy space. The Journal of Chemical Physics 116(16), 7225–7230 (2002). https://doi.org/10.1063/1.1463059

32. Resch, M.M., Kovalenko, Y., Bez, W., et al. (eds.): Sustained Simulation Performance 2018 and 2019. Springer (2020). https://doi.org/10.1007/978-3-030-39181-2

33. Romero, J., Bisson, M., Fatica, M., Bernaschi, M.: High performance implementations of the 2D Ising model on GPUs. Computer Physics Communications 256, 107473 (2020). https://doi.org/10.1016/j.cpc.2020.107473

34. Rudyak, V.Yu., Emelyanenko, A.V., Loiko, V.A.: Structure transitions in oblate nematic droplets. Physical Review E 88(5) (2013). https://doi.org/10.1103/physreve.88.052501

35. Rudyak, V.Y., Krakhalev, M.N., Sutormin, V.S., et al.: Electrically induced structure transition in nematic liquid crystal droplets with conical boundary conditions. Physical Review E 96(5) (2017). https://doi.org/10.1103/physreve.96.052701

36. Shakirov, T., Zablotskiy, S., Boeker, A., et al.: Comparison of Boltzmann and Gibbs entropies for the analysis of single-chain phase transitions. The European Physical Journal H 226, 705–723 (2017). https://doi.org/10.1140/epjst/e2016-60326-1

37. Shao, W., Guo, G.: Multiple-try simulated annealing algorithm for global optimization. Mathematical Problems in Engineering 2018, 1–11 (2018). https://doi.org/10.1155/2018/9248318

38. Shvetsov, S.A., Emelyanenko, A.V., Boiko, N.I., et al.: Communication: Orientational structure manipulation in nematic liquid crystal droplets induced by light excitation of azodendrimer dopant. The Journal of Chemical Physics 146(21), 211104 (2017). https://doi.org/10.1063/1.4984984

39. Shvetsov, S.A., Rudyak, V.Yu., Emelyanenko, A.V., et al.: Photoinduced orientational structures of nematic liquid crystal droplets in contact with polyimide coated surface. Journal of Molecular Liquids 267, 222–228 (2018). https://doi.org/10.1016/j.molliq.2018.01.054

40. Sivakumar, S., Wark, K.L., Gupta, J.K., et al.: Liquid crystal emulsions as the basis of biological sensors for the optical detection of bacteria and viruses. Advanced Functional Materials 19(14), 2260–2265 (2009). https://doi.org/10.1002/adfm.200900399

41. Tian, Z., Yokoyama, H., Araki, T.: Parallel latent dirichlet allocation using vector processors. In: 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS). pp. 1548–1555. IEEE (2019). https://doi.org/10.1109/hpcc/smartcity/dss.2019.00213

42. Tran, N.P., Lee, M., Hong, S.: Performance optimization of 3D lattice Boltzmann flow solver on a GPU. Scientific Programming 2017, 1–16 (2017). https://doi.org/10.1155/2017/1205892

43. Tu, J., Yeoh, G.H., Liu, C.: Chapter 9 - some advanced topics in CFD. In: Tu, J., Yeoh, G.H., Liu, C. (eds.) Computational Fluid Dynamics (Third Edition), pp. 369–417. Butterworth-Heinemann, third edition edn. (2018). https://doi.org/10.1016/B978-0-08-101127-0.00009-X

44. Weigel, M.: Simulating spin models on GPU. Computer Physics Communications 182(9), 1833–1836 (2011). https://doi.org/10.1016/j.cpc.2010.10.031

45. Weigel, M.: Monte Carlo methods for massively parallel computers. In: Order, Disorder and Criticality, pp. 271–340. World Scientific (2017). https://doi.org/10.1142/9789813232105_0006

46. Yamada, Y., Momose, S.: Vector engine processor of NEC Brand-New supercomputer SX-Aurora TSUBASA. In: Intenational symposium on High Performance Chips (Hot Chips2018) (2018)

47. Yusoff, M.N.S., Jaafar, M.S.: Performance of CUDA GPU in Monte Carlo simulation of light-skin diffuse reflectance spectra. In: 2012 IEEE-EMBS Conference on Biomedical Engineering and Sciences. pp. 264–269. IEEE (2012). https://doi.org/10.1109/iecbes.2012.6498056

48. Zablotskiy, S.V., Martemyanova, J.A., Ivanov, V.A., Paul, W.: Diagram of states and morphologies of flexible-semiflexible copolymer chains: A Monte Carlo simulation. Journal of Chemical Physics 144(24), 244903 (2016). https://doi.org/10.1063/1.4946035