Exact kinetic Monte Carlo in two dimensions on a GPU

Robert Darkins*, Dorothy M. Duffy, Ian J. Ford

Department of Physics and Astronomy, University College London, Gower Street, London, WC1E 6BT, UK

Abstract

Kinetic Monte Carlo (KMC) is widely used to study the dynamics of stochastic systems, particularly systems with two-dimensional domains like crystal surfaces. Here we present an exact algorithm that can harness the parallelism of a graphics processing unit (GPU) to accelerate KMC on two-dimensional lattices. The algorithm uses the waiting time method to perform rejection-free KMC over dynamically chosen time steps. It is parallelised using domain decomposition, and exactness is achieved by rejecting time steps that introduce errors detected through a consistency check. We benchmark our algorithm using a model of crystal growth, and show that the GPU can accelerate KMC by up to 118X relative to a serial code, although the precise speedup depends on the lattice size, the physical model and the hardware.

Keywords: Kinetic Monte Carlo, GPU, CUDA, Surface science

1. Introduction

Kinetic Monte Carlo (KMC) is a popular computational method in surface science for studying processes that occur on two-dimensional domains at scales intermediate between the atomistic and the macroscopic [1]. Its applications include crystal growth [2, 3], heterogeneous catalysis [4, 5], thin film deposition [6], surface diffusion [7], and self-assembly on substrates [8]. However, its computational cost often limits the length and time scales of simulations and consequently the range of processes and model parameters that can be explored.

Most efforts to accelerate KMC simulations using parallel schemes have focused on distributed computing systems, e.g. [9, 10, 11, 12, 13, 14, 15, 16], where some of these schemes are exact [10, 16]. A few studies have made use of the massive parallelism offered by graphics processing units (GPUs), but these GPU studies have all used approximate methods, either restricting the possible events that can occur to prevent conflicts between parallel threads [17, 18, 19, 20], or admitting errors that arise from conflicts into the dynamics [21]. Crucially, it can be challenging to quantify the errors that these methods introduce.

In this paper, we present an exact GPU-based algorithm for KMC simulation on two-dimensional lattices. We benchmark our algorithm on four different GPUs using a terrace-ledge-kink model of crystal growth as a test case.

2. Algorithm

We assume that the lattice is an $m \times n$ square grid with periodic boundaries, though these assumptions are not essential. Each cell (i, j) in the lattice has three variables associated with it:

- The state $x(i, j)$, which might represent a molecular species or a surface height.
- The time $\tau(i, j)$ at which the cell will next change state if its adjacent cells do not change state beforehand (since then $\tau(i, j)$ would be updated).
- The state vector $g(i, j)$ maintained internally by a pseudorandom number generator (PRNG). If multiple independent threads have their own copy of g, then they may generate an identical sequence of random numbers for any particular cell.

The user specifies the initial lattice state x. The (provisional) time of the next event at each cell is computed by adding an exponentially-distributed interval to the current time t,

$$\tau(i, j) = t + A(i, j)^{-1} \ln(\textsc{rand}(g(i, j))^{-1}), \quad (1)$$

*Corresponding author (r.darkins@ucl.ac.uk)

Algorithm 1 KMC multi-step integration

FB ← initial (x, τ, g) ▷ front buffer
allocate BB ▷ back buffer
$t \leftarrow 0$
$\Delta t \leftarrow$ initial time step
repeat
    ADVANCE(FB, $t + \Delta t$, BB)
    if checksums match **then**
        SWAP(FB, BB)
        $t \leftarrow t + \Delta t$
        $\Delta t \leftarrow \alpha \Delta t$ ▷ $\alpha > 1$
    else
        $\Delta t \leftarrow \beta \Delta t$ ▷ $\beta < 1$
    end if
until termination condition is met

Algorithm 2 Advance FB to time t and save to BB

procedure ADVANCE(FB, t, BB)
    for each tile T **do** ▷ parallelise over tiles
        $S \leftarrow T$ and its 8 adjacent tiles
        $S^\circ \leftarrow$ interior cells of S
        $(x', \tau', g') \leftarrow$ copy (x, τ, g) for S from FB
        loop
            $(i, j) \leftarrow$ cell in S° with smallest τ'
            if $\tau'(i, j) < t$ **then**
                update $x'(i, j)$
                update τ' for (i, j) neighbourhood
                update checksums for tile T
            else
                break
            end if
        end loop
        write (x', τ', g') for tile T to BB
    end for
end procedure

where $A(i, j)$ is the sum of the rates of all events that might occur at that cell, and RAND($g(i, j)$) generates a uniform variate between 0 and 1 from the state vector $g(i, j)$ which is uniquely seeded for each cell and which gets updated with every call to RAND. Note that $A(i, j)$ may depend on the value of x for cell (i, j) as well as the values of x for its eight adjacent cells. We refer to these nine cells centred on cell (i, j) as the neighbourhood of (i, j).

The system is integrated over time using the adaptive time step method described in Algorithm 1. The state (x, τ, g) is advanced by a small time step Δt, which may introduce an error. If an error is detected, the time step is rejected, and Δt is decreased. If no error is detected, the time step is accepted, and Δt is increased. By storing (x, τ, g) before and after the time step in separate buffers, the time step can be accepted (or rejected) by simply swapping the buffers (or not). Note that many events may occur within each time step.

A single time step iteration is described in Algorithm 2 and depicted in Figure 1. It is based on the waiting time method [22], where the next cell to perform an event is that with the smallest value of $\tau(i, j)$. For parallelisation, the lattice is divided into tiles of 8×8 cells, which requires that the grid dimensions m and n both be multiples of 8. The algorithm assigns each tile to a parallel thread, and each thread creates a copy of the state (x, τ, g) for the 3×3 tiles centered on its assigned tile, where the eight perimeter tiles are ghost tiles that will be discarded at the end of the time step. This domain of 3×3 tiles is then advanced by Δt subject to a fixed boundary condition. Every time a cell (i, j) in this domain undergoes an event, the current time t, as measured by the corresponding thread, is advanced to $\tau(i, j)$ and the value of τ is updated for cell (i, j) and its adjacent cells using equation 1. At the end of the time step, the updated state of the central tile is saved.

During a time step, each cell (i, j) is independently integrated nine times since each cell appears within the 3×3 neighbourhood of nine different tiles. Recall that the trajectory of a cell depends on its PRNG state vector $g(i, j)$ as well as the states x of itself and of its adjacent cells. All of these variables, and therefore the trajectory of each cell, will initially be the same across the nine parallel instances; however, these trajectories may eventually diverge because the domains in all nine instances are subject to distinct boundary conditions. The cells that compose the central tile of the 3×3 tiles must not be affected by the boundary condition during the time step since the central tile is retained at the end of the time step. Crucially, if the step size Δt is small enough, the effects of the boundary condition will not have enough time to propagate to the central tile.

2.1. Checksum validation

After each time step, the algorithm checks whether a fixed boundary condition has introduced an error to a central tile by checking for consistency between neighboring tiles whose shared edges must have identical histories.

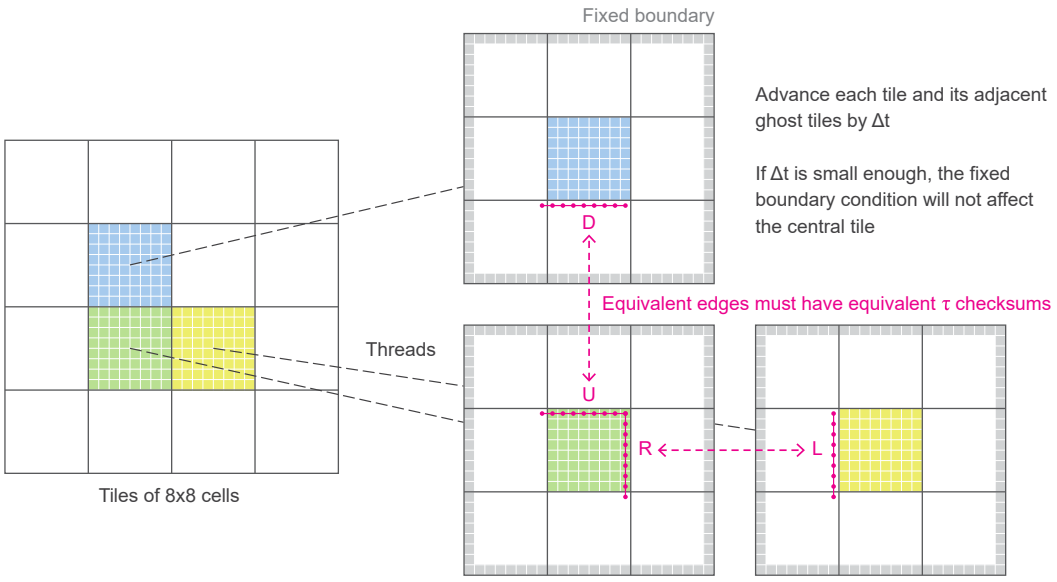

Figure 1: Summary of our algorithm. The lattice is divided into tiles of 8×8 cells. Each parallel thread picks a tile and creates a copy of the 3×3 tiles centered on that tile. These 3×3 tiles are advanced by a time step Δt, subject to a fixed boundary condition. The time step is rejected if the boundary condition affects any of the central tiles. This is detected by comparing checksums between adjacent threads along equivalent edges (shown by pink beads). For example, the U edge of the green tile must have the same checksum as the D edge of the blue tile.

Each thread records a checksum C_k for each edge k of its assigned tile T by summing the time τ of every event that occurs along edge k during the time step,

$$C_k(T) = \sum\{\tau_e \mid \text{events } e \text{ along edge } k(T)\}. \quad (2)$$

Figure 1 shows the definition of each edge, $k = L$ (left), R (right), U (up) and D (down). For consistency, if tile T_R is to the right of tile T, and tile T_U is up from tile T, then every tile T must satisfy the conditions

$$C_R(T) = C_L(T_R), \quad (3)$$
$$C_U(T) = C_D(T_U). \quad (4)$$

These conditions ensure the integrity of the entire central tile because the fixed boundary condition can only influence the central tile via its edges.

2.2. Adaptive time step

The step size Δt is initially guessed and then adapted over the course of the simulation. We choose an initial value

$$\Delta t = \min_{(i,j)} \tau(i,j) - t. \quad (5)$$

After each time step, Δt is increased if the time step was accepted ($\Delta t \leftarrow \alpha \Delta t$ where $\alpha > 1$), otherwise it is decreased ($\Delta t \leftarrow \beta \Delta t$ where $\beta < 1$).

The value of α varies during the simulation. Initially it is $\alpha = 10$ because our initial guess for Δt may be orders of magnitude smaller than the optimal value, and a large value of α allows for a rapid correction. After the first rejection of a time step, α is reduced to 1.03. Throughout the simulation, β is kept fixed at 0.5.

The step size Δt increases exponentially until it surpasses the maximum allowable value, after which it is halved (Figure 2). When using $\alpha = 1.03$ and $\beta = 0.5$, these overshoots occur once every $\log \beta / \log \alpha^{-1} \approx 23$ time steps. In our tests, we found that these values of α and β are approximately optimal but that the performance of the scheme is not highly sensitive to them. Hence, they should perform well for any physical model.

2.3. Exactness

Although the algorithm is theoretically exact, its accuracy is limited in practice by the finite precision of floating-point arithmetic. Specifically, there is an extremely low but non-zero chance that two

edges with conflicting histories will produce identical checksums. As a result, an error introduced during a time step could remain undetected.

If the checksums (and the numbers from which the checksums are computed) are represented in 64-bit precision, then the probability of a checksum collision is $\sim 2^{-53}$ per tile per time step, where 53 is the significand of the IEEE standard for floating-point arithmetic. If there are on average $\sim$10 events per tile per time step, and the time step overshoots once every $\sim$10 iterations, then the probability of introducing an error equates to $\sim 10^{-18}$ per event, which is negligible.

3. Implementation

3.1. Parallel code

We implemented our algorithm in CUDA C++ code, which we release to the public [23]. In our code, the lattice state (x, τ, g) is stored in the global memory of the GPU. During each time step, parallel threads launch from a single kernel and copy their assigned 3×3 tile neighborhood from the global memory to the registers, perform the time step calculation, and write the central tile and checksums back to the global memory. Threads invoked by a second kernel validate the checksums.

Each kernel is distributed across one block per streaming multiprocessor, with 64 threads per block. If the number of tiles exceeds the number of threads, each thread iterates over multiple tiles to ensure that all tiles are processed. The performance was not improved by assigning multiple blocks per streaming multiprocessor and processing a single tile per thread.

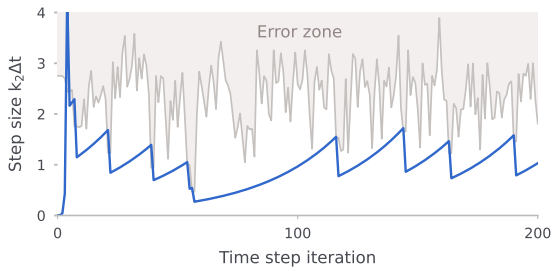

Figure 2: In our adaptive time step scheme, Δt (blue line) grows exponentially during the simulation until it is large enough to introduce an error, at which point the time step is rejected and Δt is reduced. These data were obtained from a 64×64 simulation of our crystal growth model for $\phi = 1$. The error zone was computed each time step by searching for the largest Δt that did not introduce an error.

To optimise data transfers between global memory and registers, we store each lattice variable, e.g., $x(i, j)$, as a one-dimensional array in global memory, e.g., $X(k)$, and organise it to achieve coalesced memory access. Specifically, we arrange X so that its first element corresponds to the first cell of the first tile of x, the second element of X corresponds to the first cell of the second tile of x, and so on. This tile-major ordering allows contiguous threads to access contiguous memory addresses and accelerates data transfers by an order of magnitude.

We generate 64-bit pseudorandom numbers using the Mersenne Twister algorithm implemented in the Nvidia cuRAND library.

We profiled our implementation, which revealed that the majority of time was typically spent in step 7 of algorithm 2. This step searches for the cell with the smallest value of τ, and we implemented it with a linear search through the cells. This basic method was faster than using a priority queue, likely due to thread branching.

3.2. Serial code

To serve as a benchmark for our parallel implementation of KMC, we implemented the BKL algorithm [24] in a serial C++ code. The event list was managed using a binary tree, which is a technique well-described in the literature [12].

4. Benchmark method

4.1. Physical model

To evaluate the performance of our algorithm, we tested it on a simple model of crystal growth. We designed the model to have a single parameter ϕ which controls the roughness of the crystal surface.

The model comprises an $n \times n$ grid where the state of each cell measures the local surface height, thus creating a solid-on-solid representation of a three-dimensional surface. Crystal growth occurs by the irreversible attachment of adatoms, which increases the local surface height. The rate of attachment depends on the number of newly formed bonds n_b between the adatom and its four nearest neighbors within the surface plane,

$$k(n_b) = k_2 \exp((2n_b - 4)\phi). \tag{6}$$

For example, the atom adsorbs to a terrace at a rate $k(n_b = 0) = k_2 \exp(-4\phi)$ and adsorbs to a step at a higher rate $k(n_b = 1) = k_2 \exp(-2\phi)$.

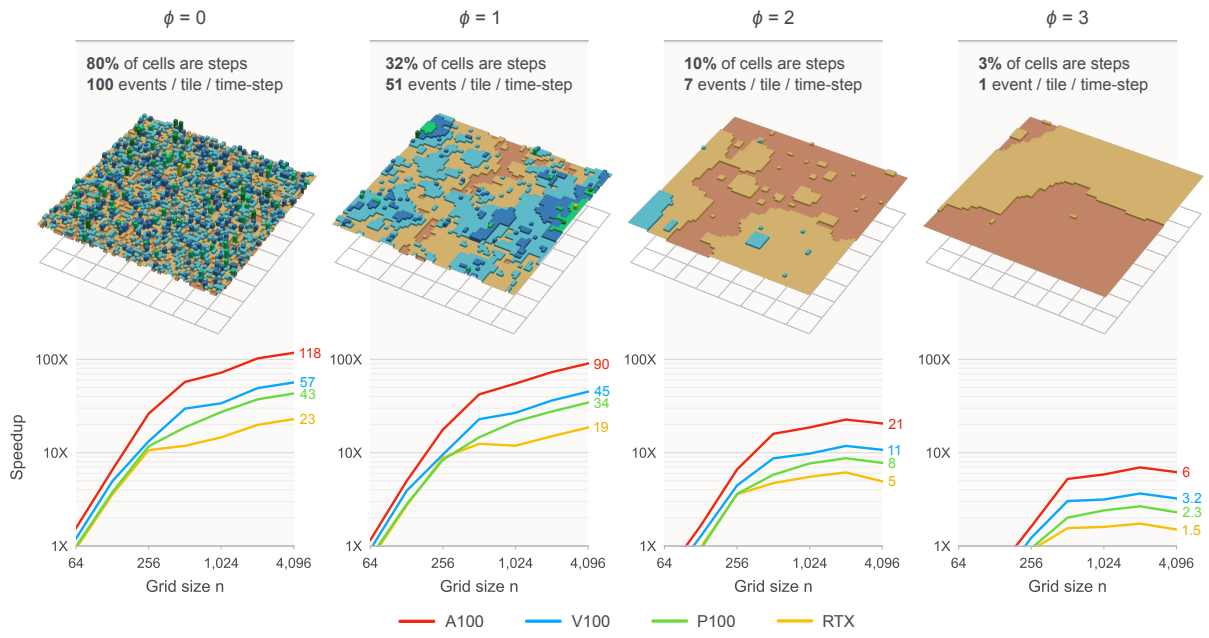

Figure 3: Benchmark results for simulations of a crystal growth model with a parameter ϕ on an $n \times n$ grid. The speedup measures the rate that the GPU performs events relative to a CPU (Intel Xeon Gold 6240 2.60 GHz). The snapshots are examples of the surface structure at different values of ϕ, with each example showing a 64×64 portion of a larger grid, and each crystal layer having a different colour.

In contriving the model to have a single parameter, we eliminated any thermodynamic measure of the driving force for growth, since the reactions are irreversible. However, the remaining parameter ϕ is analogous to the thermodynamic surface free energy in units of temperature, as the relative attachment rates of adatoms to the cells are consistent with Boltzmann statistics. Consequently, a large value of ϕ will produce surfaces that are mostly flat and featureless, whereas a small value of ϕ will produce rough surfaces, with $\phi = 0$ resulting in random deposition.

4.2. Simulations

We simulated grid sizes that were powers of two, from $n = 2^6$ to 2^{12}, and sampled the parameters $\phi = 0$, 1, 2 and 3 for each grid size.

To measure the efficiency of the parallel code on the GPU for each (n, ϕ), we started with a flat surface and relaxed it for $10^3 n^2$ events to achieve a steady growth rate. The number of events per walltime were measured over a subsequent production run of $10^3 n^2$ events.

To measure the efficiency of the serial code on the CPU, we started with the final surface configuration from a GPU production run with the same (n, ϕ) so that the surface was already relaxed. The number of events per walltime were then measured over $10^3 n^2$ events but subject to a 10 minute walltime limit.

In both the GPU and CPU cases, the production runs were sufficiently long that sampling errors were negligible.

4.3. Compilers and hardware

The parallel code was compiled using the GNU C++ 4.9.2 compiler wrapped by the CUDA 11.2.0 compiler, and its performance was measured on four Nvidia GPUs: A100, V100, P100 and GeForce RTX 3060 (RTX).

The serial code was compiled using the same GNU compiler and its performance was measured on an Intel Xeon Gold 6240 2.60GHz CPU.

5. Results and discussion

Using a one-parameter crystal growth model on an $n \times n$ grid as a test case, we found that the GPU can deliver substantial acceleration over the CPU. For example, while the CPU typically performs about 10^6 events per second, the GPU can

achieve up to 10^8 events per second. However, the degree of acceleration depends on the grid size n, the roughness of the crystal growth surface, and the type of GPU. Figure 3 provides a summary of the speedups obtained in the benchmarks, while Figure S1 contains the raw benchmark data. The fluctuations in these benchmark results were not due to sampling errors, but reflect computation complexities such as memory access patterns.

The largest speedups were observed for grid sizes greater than about 1000×1000. These large grids had a sufficient number of tiles to fully utilise the available GPU threads. The GPUs were only slower than the CPU for grid sizes smaller than about 100×100.

The performance of the GPU was affected by the physical model through its impact on the load balance of the threads. Specifically, in the crystal growth model, events mainly occurred along step edges on the crystal surface. When the model parameter ϕ was large, resulting in a low step density, there was thread redundancy due to the limited number of concurrent events. In our benchmark simulations, we observed the following statistics for values $\phi = 0$, 1, 2, and 3: the percentage of cells that represented steps (highly reactive sites) was 80%, 32%, 10%, and 3%, respectively; the average number of events per tile per time step was approximately 100, 51, 7, and 1; and the maximum speedup was 118X, 90X, 21X, and 6X. If another physical model were to exhibit a reaction density similar to any of our benchmark cases, it would be expected to experience a similar speedup on a GPU.

The four GPUs ranked consistently from fastest to slowest as follows: A100 (up to 118X), V100 (up to 57X), P100 (up to 43X), and RTX (up to 23X). Note that the RTX is a consumer graphics card while the other three GPUs are specialised for scientific computing.

Our algorithm could be expanded in various ways. For example, the grid cells do not need to be square, nor must the grid have periodic boundaries. The interaction range of cells could also be extended beyond just the adjacent cells, although this would require the fixed boundary layers to be thicker, resulting in smaller time steps and a reduced speedup. Unfortunately, we found that our algorithm provides almost no speedup when generalised to three spatial dimensions.

6. Conclusions

Our algorithm for parallel kinetic Monte Carlo in two dimensions achieves a speedup of one to two orders of magnitude on a GPU relative to a serial code. Importantly, our approach does not require a compromise between accuracy and speedup. This accelerated capability will enable the simulation of surfaces over extended length and time scales, providing better statistical sampling and access to processes that are otherwise too slow to simulate.

Funding

This work was supported by an Engineering and Physical Sciences Research Council (EPSRC) Programme Grant (EP/R018820/1).

References

[1] M. Andersen, C. Panosetti, K. Reuter, A practical guide to surface kinetic Monte Carlo simulations, Frontiers in chemistry 7 (2019) 202. doi:10.3389/fchem.2019.00202.

[2] W. Ma, J. F. Lutsko, J. D. Rimer, P. G. Vekilov, Antagonistic cooperativity between crystal growth modifiers, Nature 577 (7791) (2020) 497–501. doi:10.1038/s41586-019-1918-4.

[3] J. F. Lutsko, A. E. S. Van Driessche, M. A. Durán-Olivencia, D. Maes, M. Sleutel, Step crowding effects dampen the stochasticity of crystal growth kinetics, Physical review letters 116 (1) (2016) 015501. doi:10.1103/PhysRevLett.116.015501.

[4] R. Salazar, A. P. J. Jansen, V. N. Kuzovkov, Synchronization of surface reactions via Turing-like structures, Physical Review E 69 (3) (2004) 031604. doi:10.1103/PhysRevE.69.031604.

[5] M. Pineda, M. Stamatakis, Kinetic Monte Carlo simulations for heterogeneous catalysis: Fundamentals, current status, and challenges, The Journal of Chemical Physics 156 (12) (2022) 120902. doi:10.1063/5.0083251.

[6] P. Ghosh, N. Gupta, M. Dhankhar, M. Ranganathan, Kinetic Monte Carlo simulations of self-organization of Ge islands on Si (001), Physical Chemistry Chemical Physics 23 (34) (2021) 19022–19031. doi:10.1039/D1CP00069A.

[7] G. M. Akselrod, F. Prins, L. V. Poulikakos, E. M. Y. Lee, M. C. Weidman, A. J. Mork, A. P. Willard, V. Bulovic, W. A. Tisdale, Subdiffusive exciton transport in quantum dot solids, Nano letters 14 (6) (2014) 3556–3562. doi:10.1021/nl501190s.

[8] F. Silly, U. K. Weber, A. Q. Shaw, V. M. Burlakov, M. R. Castell, G. A. D. Briggs, D. G. Pettifor, Deriving molecular bonding from a macromolecular self-assembly using kinetic Monte Carlo simulations, Physical Review B 77 (20) (2008) 201408. doi:10.1103/PhysRevB.77.201408.

[9] Y. Shim, J. G. Amar, Semirigorous synchronous sublattice algorithm for parallel kinetic Monte Carlo simulations of thin film growth, Physical Review B 71 (12) (2005) 125432. doi:10.1103/PhysRevB.71.125432.

[10] Y. Shim, J. G. Amar, Rigorous synchronous relaxation algorithm for parallel kinetic monte carlo simulations of thin film growth, Physical Review B 71 (11) (2005) 115436. doi:10.1103/PhysRevB.71.115436.

[11] J. A. Mitchell, F. Abdeljawad, C. Battaile, C. Garcia-Cardona, E. A. Holm, E. R. Homer, J. Madison, T. M. Rodgers, A. P. Thompson, V. Tikare, E. Webb, S. J. Plimpton, Parallel simulation via SPPARKS of on-lattice kinetic and Metropolis Monte Carlo models for materials processing, Modelling and Simulation in Materials Science and Engineering 31 (5) (2023) 055001. doi:10.1088/1361-651X/accc4b.

[12] S. Plimpton, C. Battaile, M. Chandross, L. Holm, A. Thompson, V. Tikare, G. Wagner, E. Webb, X. Zhou, C. G. Cardona, A. Slepoy, Crossing the mesoscale no-man's land via parallel kinetic Monte Carlo, Sandia Report SAND2009-6226 1 (2009). doi:10.2172/966942.

[13] S. Ravipati, G. D. Savva, I. A. Christidi, R. Guichard, J. Nielsen, R. Réocreux, M. Stamatakis, Coupling the time-warp algorithm with the graph-theoretical kinetic Monte Carlo framework for distributed simulations of heterogeneous catalysts, Computer Physics Communications 270 (2022) 108148. doi:10.1016/j.cpc.2021.108148.

[14] P. Heidelberger, D. M. Nicol, Conservative parallel simulation of continuous time Markov chains using uniformization, IEEE Transactions on Parallel and Distributed Systems 4 (8) (1993) 906–921. doi:10.1109/71.238625.

[15] G. Korniss, M. A. Novotny, P. A. Rikvold, Parallelization of a dynamic Monte Carlo algorithm: a partially rejection-free conservative approach, Journal of Computational Physics 153 (2) (1999) 488–508. doi:10.1006/jcph.1999.6291.

[16] G. D. Savva, R. L. Benson, I. A. Christidi, M. Stamatakis, Exact distributed kinetic monte carlo simulations for on-lattice chemical kinetics: lessons learnt from medium-and large-scale benchmarks, Philosophical Transactions of the Royal Society A 381 (2250) (2023) 20220235. doi:10.1098/rsta.2022.0235.

[17] N. J. van der Kaap, L. J. A. Koster, Massively parallel kinetic Monte Carlo simulations of charge carrier transport in organic semiconductors, Journal of Computational Physics 307 (2016) 321–332. doi:10.1016/j.jcp.2015.12.001.

[18] J. Kelling, G. Ódor, M. F. Nagy, H. Schulz, K. H. Heinig, Comparison of different parallel implementations of the 2+1-dimensional KPZ model and the 3-dimensional KMC model, The European Physical Journal Special Topics 210 (1) (2012) 175–187. doi:10.1140/epjst/e2012-01645-8.

[19] G. Arampatzis, M. A. Katsoulakis, P. Plecháč, M. Taufer, L. Xu, Hierarchical fractional-step approximations and parallel kinetic Monte Carlo algorithms, Journal of Computational Physics 231 (23) (2012) 7795–7814. doi:10.1016/j.jcp.2012.07.017.

[20] J. Kelling, G. Ódor, Extremely large-scale simulation of a Kardar-Parisi-Zhang model using graphics cards, Physical Review E 84 (6) (2011) 061150. doi:10.1103/PhysRevE.84.061150.

[21] F. Jiménez, C. J. Ortiz, A GPU-based parallel object kinetic Monte Carlo algorithm for the evolution of defects in irradiated materials, Computational Materials Science 113 (2016) 178–186. doi:10.1016/j.commatsci.2015.11.011.

[22] J. Dall, P. Sibani, Faster monte carlo simulations at low temperatures. the waiting time method, Computer Physics Communications 141 (2) (2001) 260–267. doi:10.1016/S0010-4655(01)00412-X.

[23] R. Darkins, Exact on-lattice KMC on a GPU, `https://github.com/RDarkins/kmc-gpu`, GitHub Repository (2023).

[24] A. B. Bortz, M. H. Kalos, J. L. Lebowitz, A new algorithm for monte carlo simulation of ising spin systems, Journal of Computational Physics 17 (1) (1975) 10–18. doi:10.1016/0021-9991(75)90060-1.