Benchmarking Strategies for Parallel 2D Ising Model Simulation
Advanced Methods for Scientific Computing - A.Y. 2023/2024

Luigi Pagani
High-Performance Computing Engineering
Politecnico di Milano - Milan, Italy
Email: luigi2.pagani@mail.polimi.it
Student ID: 10677832

1. Introduction and project goal

The simulation of the two-dimensional (2D) Ising Model, a fundamental model in statistical mechanics, presents some computational challenges, particularly near critical temperatures where critical slowing down occurs. This project focuses on benchmarking parallelized algorithms for the 2D Ising Model simulation, aiming to enhance computational efficiency and accuracy in different high-performance computing environments.

The following parallelization strategies and algorithms are explored and analyzed:

i. **CUDA Parallelized Metropolis-Hastings Algorithm:** Implementation leveraging GPU capabilities for efficient arithmetic operations and memory management.
ii. **Wolff Algorithm:** A cluster-flipping approach with a focus on sequential implementation to address critical slowing down.
iii. **Swendsen-Wang Algorithm:** Another cluster-based method, but with a design more amenable to parallel computation, implemented using OpenMP.
iv. **Parallel Tempering with Swendsen-Wang Updates:** An advanced Monte Carlo method combining parallel tempering techniques with Swendsen-Wang cluster updates for enhanced state space exploration.

The goal is to evaluate the performance, scalability, and efficiency of these algorithms in simulating the 2D Ising Model.

2. Expected Correct Result for $J = 1$ at Temperatures Between 0 and 3 Kelvin

For the 2D Ising Model the expected behavior at temperatures between 0 and 3 Kelvin can be summarized as follows:

- **At Low Temperatures (0 K - Tc)**: In this range, the system is expected to exhibit spontaneous magnetization. At absolute zero ($T = 0$), all spins should align, resulting in maximal magnetization. As temperature increases, thermal fluctuations induce random spin flips, but the overall magnetization remains significant due to the system's tendency towards ordered states.
- **Near the Critical Temperature (Tc)**: At the critical temperature, $Tc \approx 2.269$, the system undergoes a phase transition from an ordered to a disordered state. Here, the magnetization abruptly decreases, marking the loss of spontaneous magnetization. This point is critical for observing the behavior of the Ising Model in simulations.
- **At High Temperatures (Tc - 3 K)**: Beyond the critical temperature, thermal agitation dominates, leading to a disordered state where spins are randomly oriented. In this regime, the magnetization should approach zero, indicating the lack of long-range order in the system.

This temperature-dependent behavior of magnetization is crucial for verifying the correctness of the implemented simulation algorithms. The algorithms implemented reproduce these expected results, providing confidence in their accuracy and effectiveness for simulating the 2D Ising Model.

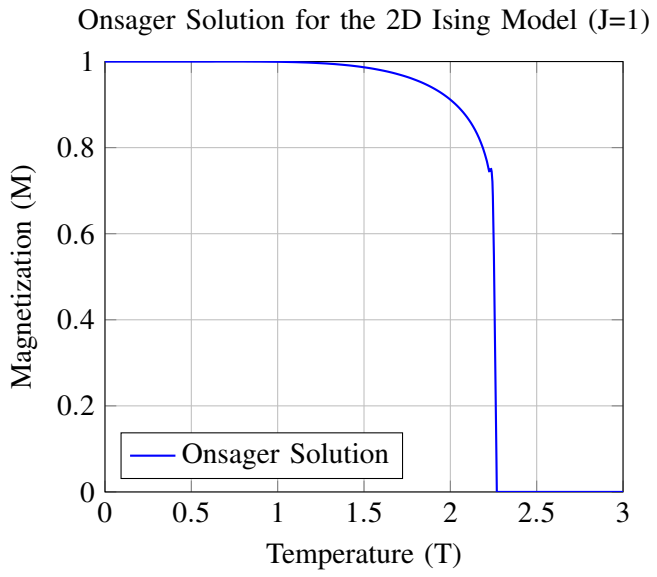

Figure 1. Theoretical magnetization curve of the 2D Ising Model as predicted by the Onsager solution.

3. Physical Problem and Assumptions

Spin Interaction Assumption

In the 2D Ising model, we consider a square lattice where each site has a spin that can take a value of either +1 or -1. The key assumption is that each spin interacts only with its nearest four neighbors (left, right, above, and below). The Hamiltonian of the system is given by:

$$H = -J \sum_{\langle i,j \rangle} s_i s_j \tag{1}$$

where s_i and s_j are the spin values at sites i and j, J is the interaction strength, and the sum is over nearest neighbor pairs.

Periodic Boundary Conditions

To mimic an infinite lattice and reduce edge effects, periodic boundary conditions are applied. This means that the grid is conceptualized as a torus where the top and bottom edges are connected, as are the left and right edges. As a result, each spin has four neighbors, even at the edges of the lattice.

4. Detailed Code Analysis and Commentary

4.1. Class `AbstractLattice`

This class serves as an abstract base for lattice structure simulations. It outlines the fundamental methods required for any lattice model, ensuring a standardized approach across different implementations.

Constructor

The constructor is implicitly defined, as this is an abstract class. Derived classes must provide their own constructors to initialize specific lattice configurations.

Public Methods

- `initialize`: A pure virtual function that must be overridden to set up the lattice according to specific rules or data.
- `evaluate_energy`: Computes and returns the current energy state of the lattice. It provides a measure of the system's stability.
- `print_lattice`: Offers a visualization or textual representation of the lattice's current state, aiding in debugging and analysis.
- `get_interaction_energy`: Calculates and returns the energy due to interactions between lattice elements. Essential for understanding lattice dynamics.

Destructor

- `AbstractLattice`: A virtual destructor ensuring proper cleanup of derived class objects. It prevents memory leaks and other resource mismanagement issues in polymorphic use cases.

Usage and Extension

This class is designed to be extended by specific lattice models. By inheriting from `AbstractLattice`, derived classes must implement the pure virtual methods, thus adhering to the prescribed interface. This approach promotes consistency and reusability in lattice simulations.

Design Rationale

The use of pure virtual functions enforces a contract for derived classes, ensuring that they provide specific implementations for initialization, energy evaluation, and visualization. This design choice supports polymorphism and extensibility, key principles

in object-oriented programming, especially relevant in simulation frameworks where different models might need to be tested and compared.

4.2. Class `SquareLattice`

The `SquareLattice` class extends the 'AbstractLattice and implements a square lattice structure, primarily used in simulations. This class provides specific functionalities for initializing, visualizing, and calculating the properties of a square lattice.

Constructor

`SquareLattice`: Constructs a `SquareLattice` object with specified interaction strength and lattice size. The constructor initializes the lattice and random lattice vectors.

Parameters

- `interactionStrength`
- `latticesize`

Public Methods

- `print_lattice`: Visualizes the current state of the lattice on standard output. Spins are represented as 'o' for -1 and 'x' for 1, aiding in a quick and intuitive understanding of the lattice state.
- `evaluate_energy`: Computes the total energy of the lattice based on its current state and the interaction strength. This function is crucial for understanding the stability and dynamics of the lattice.
- `initialize`: Initializes the lattice with random spin states and calculates its initial total energy and magnetization. This method sets the starting point for simulations.
- `get_lattice`: Provides access to the lattice vector. This method allows external functions to interact with the lattice's current state.

Implementation Details

The `SquareLattice` class represents a specific type of lattice model. It uses a vector to store the lattice state, with each element representing a spin. The energy calculation takes into account the interaction between neighboring spins, following the principles of statistical mechanics.

The constructor and the `initialize` method play a crucial role in setting up the lattice for simulations. They ensure that the lattice starts from a well-defined state, either random or specified, for accurate and reproducible simulation results.

Design Rationale

This class follows the principles of object-oriented design, extending the `AbstractLattice` to provide specific functionality for square lattices. By encapsulating the lattice-related operations within this class, the design promotes modularity and reusability, allowing the `SquareLattice` to be easily integrated into larger simulation frameworks. The choice of using vectors for lattice representation is driven by the need for efficient access and manipulation of lattice states, which is critical for performance in simulations.

Usage in Simulations

The `SquareLattice` class is used to create and manipulate square lattice models in simulations. Its methods provide essential capabilities like initializing the lattice, calculating its energy, and visualizing its state. These functionalities are crucial for studying phenomena like phase transitions, magnetization properties, and other physical characteristics in statistical mechanics simulations.

4.3. Class: `Wolff`

Overview

The Wolff algorithm, an alternative to the traditional Metropolis-Hastings approach, is a cluster-flipping algorithm that significantly reduces critical slowing down near phase transitions in spin systems like the Ising Model. This algorithm identifies and flips clusters of spins, effectively capturing long-range correlations that are prevalent near critical points[3].

1) **Random Seed Selection**: Begin by randomly selecting a single spin from the lattice to act as a seed for the cluster.
2) **Cluster Formation**: Grow the cluster from this seed. Neighboring spins with the same orientation as the seed spin are added to the cluster with a probability $P = 1 - e^{-2J/kT}$, where J represents the coupling constant, k is the Boltzmann constant, and T is the temperature.
3) **Cluster Flipping**: Flip all spins within the cluster simultaneously, which is more effective than in-

dividual spin flipping, particularly in the presence of long-range spin correlations.

4) **Iterations**: Repeat the process for multiple iterations, each starting with a new random seed spin, leading to the formation and flipping of a new cluster.

This algorithm is particularly advantageous near critical points, where it significantly outperforms single-spin-flip methods like Metropolis-Hastings.

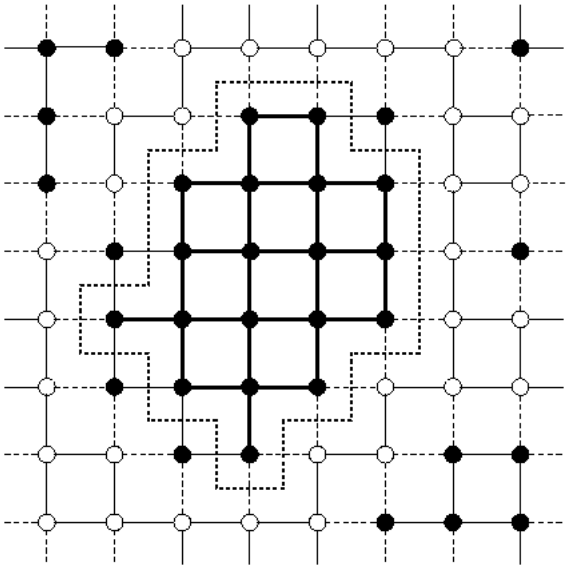

Figure 2. A lattice, showing a cluster and its boundary

Parallelization

While the current implementation of the Wolff algorithm is sequential, there were considerations for parallelization, particularly in the cluster formation and flipping processes. However, due to the inherent sequential nature of the Wolff algorithm, where the formation of a cluster depends on the dynamic addition of neighboring spins, parallelizing this process posed significant challenges. The algorithm's dependency on sequential updates can lead to complex synchronization issues in a parallel computing environment, making it less efficient for parallel adaptation, even though speedup, albeit modest, have been previously obtained [1].

In light of these challenges, I chose to focus on parallelizing the Swendsen-Wang algorithm for large-scale simulations. The Swendsen-Wang algorithm, another cluster-based approach, is more amenable to parallelization. Unlike the Wolff algorithm, which builds a single cluster starting from a random seed, the Swendsen-Wang algorithm identifies multiple clusters simultaneously. This feature naturally lends itself to parallel computation, as different processors or threads can independently identify and flip different clusters simultaneously without the need for intricate synchronization.

This class encapsulates the functionalities of the Wolff algorithm for simulating phase transitions in spin lattice systems.

Constructor

Initializes the lattice model and simulation parameters.

Parameters

- `interactionStrength` - Strength of the interaction in the lattice model.
- `latticeSize` - Size of the square lattice.
- `T_MIN` - Minimum temperature for the simulation.
- `T_MAX` - Maximum temperature for the simulation.
- `T_STEP` - Temperature step for the simulation.
- `IT` - Number of iterations for each temperature.

Public Methods

- `simulate_phase_transition`: Executes the phase transition simulation over a range of temperatures using the Wolff algorithm, recording the magnetization at each step.
- `calculate_magnetization_per_site`: Computes the average magnetization per site of the lattice, given the lattice state.
- `add_to_cluster`: Adds a spin to the cluster based on a probability threshold, updating the cluster and queue of spins.
- `update`: Updates the lattice state for a given temperature using the Wolff algorithm.
- `simulate`: Conducts the simulation for a single temperature, returning the final magnetization per site.
- `create_rand_vect`: Generates a vector of random integers used in the simulation.
- `store_results_to_file`: Saves the simulation results, including magnetization and temperature data, to a file.

Private Members

- `lattice`: Represents the lattice model.

- `MagnetizationResults`: Stores the magnetization results.
- `Temperatures`: Stores the temperatures at which simulations were conducted.
- `T_STEP`, `T_MIN`, `T_MAX`, `L`, `N`, `IT`: Parameters for the simulation.

Implementation Choices

Cluster Formation Method: The Wolff algorithm uses the `add_to_cluster` function to form clusters. It adds neighboring spins to the current cluster based on a probability threshold P, which is a function of temperature and interaction strength.

Probabilistic Spin Inclusion: The algorithm calculates the probability $P = 1 - e^{-2J/T}$ for a spin to be included in the cluster.

Random Seed Selection for Cluster Initiation: The `update` function initiates the cluster growth by selecting a random seed spin.

Use of Standard Library Containers: Containers like `std::vector`, `std::unordered_set`, and `std::queue` are used for managing lattice states, clusters, and spin queues.

Spin Flip Mechanism: Once a cluster is formed, all spins within it are flipped.

Temperature-Dependent Simulation: The simulation iteratively adjusts the temperature from T_{MIN} to T_{MAX}.

Magnetization Calculation: The `calculate_magnetization_per_site` function computes the magnetization per site after each temperature step.

4.4. Class: `SwendsenWangParallel`

The Swendsen-Wang algorithm is a pivotal method for simulating spin systems, such as the Ising model, especially near critical points. Unlike the Wolff algorithm that flips clusters formed from a single seed, Swendsen-Wang constructs and flips multiple clusters in each iteration, enhancing efficiency in capturing long-range correlations.

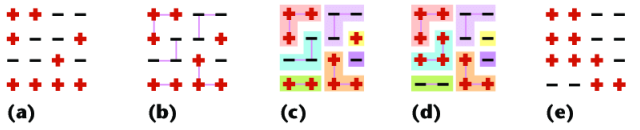

Figure 3. Illustration of the Swendsen-Wang algorithm as described in the text. (a) Original spin configuration; the "+" signs indicate spins that point upward (out of the plane) and "" signs indicate spins that point downward (into the plane). (b) Bonds (purple lines) are formed between spins of the same sign, with a probability that depends on the coupling strength. (c) All spins that are connected by bonds belong to a single cluster (each color of shading indicates a separate cluster). (d) Each cluster of spins is flipped with a probability of 50 percent. In this example, only the blue, green, and yellow clusters are flipped.(e) All bonds are erased and a new spin configuration is obtained.

1) **Bond Formation**: For each pair of neighboring spins, form a bond between them with a probability $P = 1 - e^{-2J/kT}$ if they are aligned (i.e., have the same orientation). Here, J is the coupling constant, k is the Boltzmann constant, and T is the temperature.
2) **Cluster Identification**: Identify clusters of spins where a cluster is defined as a set of spins connected by bonds. This step typically involves a graph connectivity algorithm to find all connected components (clusters) in the lattice.
3) **Cluster Flipping**: Flip each cluster of spins with a 50% probability. Unlike the Wolff algorithm, where the whole cluster identified from the seed is flipped, in Swendsen-Wang, each identified cluster has an independent chance of being flipped.
4) **Iterations**: Repeat the above steps for multiple iterations. Each iteration involves re-forming bonds and identifying new clusters, thus ensuring that the system explores a wide range of configurations.

The Swendsen-Wang algorithm is particularly advantageous in systems with strong correlations and near critical points, offering improved efficiency over single-spin-flip methods and even over some other cluster algorithms by simultaneously handling multiple clusters.

Parallelization Strategy

The Swendsen-Wang algorithm's structure supports parallelization. In this implementation, cluster identification and flipping are parallelized using OpenMP, demonstrating the use of multi-threading.

Constructor

Initializes the lattice model and simulation parameters.

Parameters

- `interactionStrength` - Strength of the interaction in the lattice model.
- `latticeSize` - Size of the square lattice.
- `T_MIN` - Minimum temperature for the simulation.
- `T_MAX` - Maximum temperature for the simulation.
- `T_STEP` - Temperature step for the simulation.
- `IT` - Number of iterations for each temperature.

This class encapsulates the functionalities of the Swendsen-Wang algorithm for simulating phase transitions in a parallel computing environment.

Public Methods

- `simulate_phase_transition`: Simulates the phase transition across a range of temperatures using the Swendsen-Wang algorithm and records the magnetization.
- `calculate_magnetization_per_site`: Calculates the magnetization per site of the lattice, returning the average magnetization.
- `find_set`: Finds the root of the set that element `x` belongs to, using path compression for efficiency.
- `union_sets`: Unions the sets containing elements `x` and `y`, using rank to keep the tree flat.
- `simulate_step`: Performs a single simulation step of the Swendsen-Wang algorithm, including forming clusters and flipping them based on a probability threshold `P`.
- `simulate`: Runs the full Swendsen-Wang simulation for a given temperature, iterating `IT` times per temperature.
- `store_results_to_file`: Stores the results of the simulation to a file, including the magnetization and temperature for each step of the simulation.

Private Members

- `lattice`: Represents the lattice model.
- `MagnetizationResults`: Stores the magnetization results.
- `Temperatures`: Stores the temperatures at which simulations were run.
- `T_STEP`, `T_MIN`, `T_MAX`, `L`, `N`, `IT`: Parameters for the simulation.
- `rngs`: A vector of random number generators, one per thread in the parallel environment.

Implementation Choices

Parallel Cluster Formation:

The process of forming clusters can be parallelized. Each thread can start building a cluster from a different site. Care must be taken to manage conflicts where two clusters might try to add the same site.

Disjoint Set Data Structure:

A disjoint set (union-find) data structure is used to efficiently manage clusters. This structure helps in quickly determining whether two sites belong to the same cluster and in merging clusters when necessary.

Use of OpenMP:

OpenMP is utilized for parallelizing the loop where clusters are formed. The `#pragma omp parallel for` directive is used to distribute the loop iterations (sites on the lattice) across multiple threads.

Thread-Local Random Number Generators:

Each thread uses its own random number generator (RNG) to decide whether to add a neighbor to a cluster. This avoids the need for synchronization when accessing a shared RNG.

Critical Section for Union Operations:

When merging clusters (in the union operation), a critical section (`#pragma omp critical`) ensures that only one thread can perform the union operation at a time, preventing data race conditions.

Single-Threaded Flip Decisions:

After clusters are formed, the decision to flip each cluster can be done in a single-threaded manner. Each cluster's flipping decision is independent and does not require synchronization.

Flipping Clusters:

The actual flipping of the clusters' spins can be done in parallel, as it involves independent write operations.

Class: `Rem`

The Replica Exchange Method, also known as Parallel Tempering, is a powerful Monte Carlo technique designed to overcome the issue of slow dynamics in simulations of systems like the 2D Ising Model, especially near critical points. The key idea is to simulate multiple replicas of the system at different

temperatures and periodically exchange configurations between these replicas. This approach allows each replica to traverse through different energy landscapes, thereby enhancing sampling efficiency and avoiding local minima.[2]

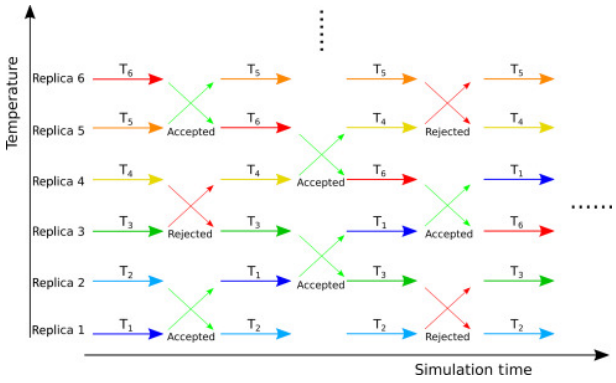

Figure 4. Schematic of the replica exchange. A number of MD simulations called replicas (six shown here) are performed in parallel and with different temperatures. At recurring intervals, temperatures, or coordinates of neighboring replicas are exchanged with a probability that meets the Metropolis criterion.

Mechanism of Replica Exchange

The mechanism of replica exchange involves replicating the system multiple times, each at a different temperature, and these replicas are simulated independently most of the time. At regular intervals, a pair of replicas, typically at adjacent temperatures, are chosen, and an exchange of their configurations is proposed. The exchange of configurations between replicas is accepted based on the Metropolis criterion, which depends on the difference in their energies and temperatures. This method allows replicas at lower temperatures to escape local minima by periodically 'visiting' higher energy states facilitated by the higher temperature replicas.

This class implements the Replica Exchange Monte Carlo (REM) algorithm for simulating phase transitions in spin lattice systems.

Constructor

Initializes the lattice model and simulation parameters.

Parameters

- `interactionStrength` - Strength of the interaction in the lattice model.
- `latticeSize` - Size of the square lattice.
- `T_MIN` - Minimum temperature for the simulation.
- `T_MAX` - Maximum temperature for the simulation.
- `T_STEP` - Temperature step for the simulation.
- `IT` - Number of iterations for each temperature.

Public Methods

- `simulate_phase_transition`: Executes the phase transition simulation across a range of temperatures using the REM method in parallel. vbnet Copy code
- `evaluate`: Calculates the energy of a given lattice configuration.
- `calculate_magnetization_per_site`: Computes the average magnetization per site of the lattice.
- `find_set`: Finds the root of the set an element belongs to, using path compression for efficiency.
- `union_sets`: Unions two sets containing elements, using rank to keep the tree flat.
- `simulate_step`: Performs a single simulation step of the Monte Carlo algorithm.
- `simulate`: Runs the full Monte Carlo simulation for a given temperature.
- `attempt_replica_exchange`: Attempts to exchange lattice configurations with a neighboring replica.
- `store_results_to_file`: Saves the simulation results, including magnetization and temperature data, to a file.

Private Members

- `lattice`: Represents the lattice model.
- `MagnetizationResults`: Stores the magnetization results.
- `Temperatures`: Stores the temperatures at which simulations were conducted.
- `T_STEP`, `T_MIN`, `T_MAX`, `L`, `N`, `IT`: Parameters for the simulation.

Implementation Choices

Parallel Simulation: The REM algorithm divides the temperature range among MPI processes and performs simulations in parallel, enhancing computational efficiency and scalability.

Energy Evaluation: The `evaluate` method calculates the energy of a lattice configuration, crucial for understanding the system's state and the effectiveness of phase transitions.

Magnetization Calculation: The `calculate_magnetization_per_site` function computes the average magnetization per site, providing insight into the magnetic properties of the lattice at different temperatures.

Disjoint Set Operations: The `find_set` and `union_sets` methods implement efficient disjoint set operations using path compression and union by rank, optimizing the cluster formation process in the Monte Carlo simulation.

Monte Carlo Simulation Steps: The `simulate_step` function encapsulates the core steps of the Monte Carlo method, including cluster formation and spin flipping, tailored for the REM algorithm.

Temperature-Dependent Simulation: The `simulate` function conducts a detailed Monte Carlo simulation at a given temperature, integral for studying the behavior of the system across the temperature spectrum.

Replica Exchange Mechanism: The `attempt_replica_exchange` method is pivotal in the REM approach, allowing for the exchange of lattice configurations between replicas at different temperatures, based on the Metropolis criterion.

4.5. CUDA Parallelized Metropolis-Hastings Algorithm

Overview

The Metropolis-Hastings algorithm for the 2D Ising model can be summarized in the following steps:

1) Initialize the lattice with spins either randomly or in an ordered state.
2) Randomly select a spin from the lattice.
3) Calculate the energy change ΔE that would result if this spin were flipped. This can be computed based on the interaction with its nearest neighbors.
4) Decide whether to flip the spin. If $\Delta E \leq 0$, flip the spin. If $\Delta E > 0$, flip the spin with a probability of $e^{-\Delta E/kT}$, where k is the Boltzmann constant and T represents the temperature.
5) Repeat steps 2 to 4 for a large number of iterations to ensure the system reaches equilibrium.

These steps allow the system to explore different configurations and approach the Boltzmann distribution at a given temperature, facilitating the study of various thermodynamic properties in the Ising model.

Parallelization Strategy

The parallelization strategy in this CUDA implementation is centered around maximizing the use of GPU resources. A checkerboard updating scheme is employed to efficiently parallelize the simulation of the Ising Model. This scheme is based on dividing the lattice into two interlocking checkerboard patterns, typically referred to as black and white squares. 5

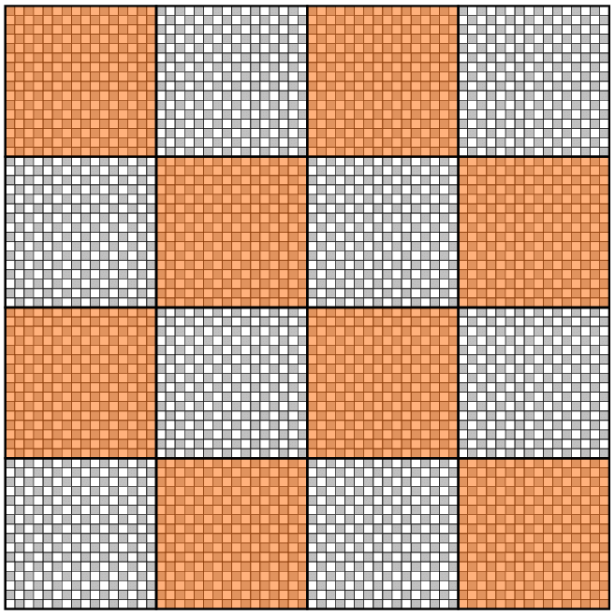

Figure 5. A double 2D checkerboard of $D \times D = 4 \times 4$ tiles, each one containing $T \times T = 16 \times 16$ spins.

In each Monte Carlo step, all the lattice sites corresponding to one color (black or white) are updated simultaneously. Since the nearest neighbors of any site in this pattern are of the opposite color, updating all sites of the same color at once avoids conflicts where a site and its neighbor might try to update simultaneously based on outdated information. This strategy allows for significant parallelization, as each lattice site is updated by a separate thread.

Operations such as energy calculation and spin flipping are then performed in parallel across the entire lattice, with two distinct kernel launches per Monte Carlo step - one for black sites and one for white sites. This checkerboard approach significantly reduces the overall simulation time and enhances the efficiency of the simulation, making it ideal for studying the properties of the Ising Model across a wide range of temperatures.

The structure of this CUDA program is specifically designed for development on cloud-based platforms like Google Colab and Kaggle. The streamlined code structure, avoiding traditional .cuh and .cu files and Makefile, is more practical for these environments, focusing on ease of use and efficiency in building and executing the program.

Functions

- `main`: Initializes CUDA variables, sets up random number generation, and controls the temperature loop for simulations.
- `__device__ int get_index`: Calculates the index in the lattice based on row and column.
- `__device__ int delta_energy`: Computes the change in energy for a given lattice spin.
- `__global__ void flip_spins`: Manages the flipping of spins and updating energy in the lattice.
- `__global__ void initialize_lattice_kernel`: Initializes the lattice with random spin values.
- `__global__ void setup_rand_kernel`: Sets up the random number generation for the simulation.
- `__global__ void calculate_magnetization_kernel`: Calculates the magnetization of the lattice.

Implementation Choices

Streamlined Structure for Cloud-Based Platforms: The program avoids traditional .cuh and .cu files and Make, simplifying its use in cloud-based environments.

Optimization for GPU Performance: The choice of NTHREADS and lattice size is crucial for maximizing GPU efficiency. The program ensures optimal performance by aligning the number of threads with the lattice structure.

Energy and Spin Dynamics: The `delta_energy` and `flip_spins` functions accurately calculate the energy changes and manage spin orientations, which are central to studying phase transitions.

Random Initialization: The `initialize_lattice_kernel` and `setup_rand_kernel` functions provide a robust random initialization of the lattice and random number generators, essential for the stochastic nature of Monte Carlo simulations.

Temperature-Dependent Simulation: The main loop varies the temperature, allowing the study of phase transition dynamics over a range of temperatures.

5. Results and performance

CPU Specifications

- **Model**: Intel Core i7-9750H CPU
- **Architecture**: x86_64
- **CPU Op-Mode(s)**: 32-bit, 64-bit
- **Byte Order**: Little Endian
- **CPU(s)**: 6
- **On-line CPU(s) List**: 0-5
- **Threads per Core**: 2
- **Cores per Socket**: 6
- **Socket(s)**: 1
- **NUMA Node(s)**: 1

GPU Specifications

- **Model**: NVIDIA Quadro P600
- **Series**: Quadro Series
- **CUDA Parallel Processing Cores**: 96
- **Frame Buffer Memory**: 1 GB DDR3
- **Memory Interface**: 128-bit
- **Memory Bandwidth**: 25.6 GB/s

5.1. Performance of the implementations

Algorithm	Time Required
Wolff Algorithm	***568.959s***
Swendsen-Wang Algorithm (Single-threaded)	***5.824s***
Swendsen-Wang Algorithm (6-threaded)	***12.267s***
CUDA Metropolis Algorithm	***421.364s***

TABLE 1. COMPARISON OF TIME REQUIRED FOR COMPLETE PHASE TRANSITION (0.2 K-2.8 K) OF A LATTICE OF SIZE $N = 256 \times 256$.

The comparison of time required to simulate a complete phase transition involves three distinct algorithms: the Wolff Algorithm, the Swendsen-Wang algorithm (both in single-threaded and 30-threaded executions using OpenMP), and the CUDA Metropolis algorithm. It is important to note that this comparison excludes the Replica Exchange method. The primary function of the Replica Exchange method is to enhance

convergence efficiency near the critical temperature by reducing the number of iterations required. However, it does not directly contribute to speed enhancements, as it can be thought as a ”different form” of parallelism. Furthermore, accurately quantifying the reduction in iterations necessitated by the Replica Exchange method is not straightforward, and thus it is omitted from this comparative analysis.

The parallel implementation of the Swendsen-Wang algorithm is slower than the serial version in all tests, even when using processors with up to 30 physical cores. The only way to accelerate the parallel algorithm was by removing the `#pragma omp critical` section from the cluster formation phase. While incorrect from an algorithmic perspective, this modification did not compromise convergence over more than 30 empirical tests. However, we do not include the performance of this modified parallel approach given it deviates from a valid implementation.

References

[1] J. Kaupu žs, J. Rim š āns, and R. V. N. Melnik. “Parallelization of the Wolff single-cluster algorithm”. In: *Phys. Rev. E* 81 (2 Feb. 2010), p. 026701. DOI: 10.1103/PhysRevE.81.026701. URL: https://link.aps.org/doi/10.1103/PhysRevE.81.026701.

[2] E Marinari and G Parisi. “Simulated Tempering: A New Monte Carlo Scheme”. In: *Europhysics Letters (EPL)* 19.6 (July 1992), pp. 451–458. ISSN: 1286-4854. DOI: 10.1209/0295-5075/19/6/002. URL: http://dx.doi.org/10.1209/0295-5075/19/6/002.

[3] M. E. J. Newman and G. T. Barkema. *Monte Carlo methods in statistical physics*. Oxford: Clarendon Press, 1999.