UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Parallelizing 2D Ising model

Different implementations and benchmarks

W. Conte

February 19, 2026

Table of Contents

1. Introduction
   - Physics
   - Hardware

2. Implementation
   - General structure
   - Serial implementation
   - Parallel implementations

3. Benchmarks
   - Physics correctness
   - Performances

Table of Contents

1. Introduction
   - Physics
   - Hardware

2. Implementation
   - General structure
   - Serial implementation
   - Parallel implementations

3. Benchmarks
   - Physics correctness
   - Performances

Ising Model (1)

Ising model is a statistical mechanics framework used to describe ferromagnetic systems.

1. The Hamiltonian (Energy)

The energy of a configuration σ is defined by:

$$\mathcal{H}(\sigma) = -J \sum_{\langle i,j \rangle} \sigma_i \sigma_j - h \sum_i \sigma_i$$

- $J > 0$: Ferromagnetic coupling constant (neighbors prefer alignment).
- $\langle i, j \rangle$: Sum over nearest neighbors (top, bottom, left, right).
- h: External magnetic field (set to 0 for spontaneous magnetization tests).

2. Order Parameter (Magnetization)

The macroscopic magnetization acts as the order parameter for the phase transition:

$$m = \frac{1}{N} \sum_{i=1}^{N} \sigma_i \quad \in [-1, 1]$$

- **High T:** $m \approx 0$ (Paramagnetic / Disordered Phase).
- **Low T:** $|m| \to 1$ (Ferromagnetic / Ordered Phase).

The Metropolis-Hastings Algorithm

Core Idea: A Markov Chain Monte Carlo (MCMC) method to sample configurations according to the Boltzmann distribution $P(\sigma) \propto e^{-\beta \mathcal{H}(\sigma)}$.

Why use it?

- Calculating the partition function Z is analytically intractable for large N.
- We need a way to simulate thermal fluctuations and reach equilibrium efficiently.

How it works (The Step):

1. Pick a random spin σ_i and propose a flip: $\sigma_i' = -\sigma_i$.
2. Calculate energy change:

$$\Delta E = 2\sigma_i \left(J \sum_{j \in \text{nn}(i)} \sigma_j + h \right)$$

3. **Acceptance Criterion:**
   - If $\Delta E \leq 0$: **Accept**
   - If $\Delta E > 0$: Accept with probability $p = e^{-\beta \Delta E}$.

Hardware specifications: the host (CPU)

Processor: Intel® Core™ i5-13420H

Hybrid Architecture Configuration:

Total: 8 Cores / 12 Threads

1. Performance Cores (P-Cores)

- **Count:** 4 Physical Cores (8 Threads via Hyper-Threading).
- **Role:** High-performance tasks, low latency, heavy single-thread workload.

2. Efficiency Cores (E-Cores)

- **Count:** 4 Physical Cores (4 Threads, NO Hyper-Threading).
- **Role:** Background tasks
- **Characteristics:** Lower clock speeds, optimized for power efficiency.

Hardware specifications: the device (GPU)

Device: NVIDIA GeForce RTX 4050 Laptop GPU

Specs:

- **Architecture:** Ada Lovelace
- **CUDA Cores:** 2560
- **VRAM:** 6 GB
- **Memory Bandwidth:** 192 GB/s
- **SM count:** 20 (128 CUDA cores each)
- **Max threads / SM:** 1536

The "Ada" Advantage:

- Unlike previous generations (Ampere), Ada Lovelace features a massive increase in on-chip memory.
- **L2 Cache Size:** 24 MB

Table of Contents

1. Introduction
   - Physics
   - Hardware

2. Implementation
   - General structure
   - Serial implementation
   - Parallel implementations

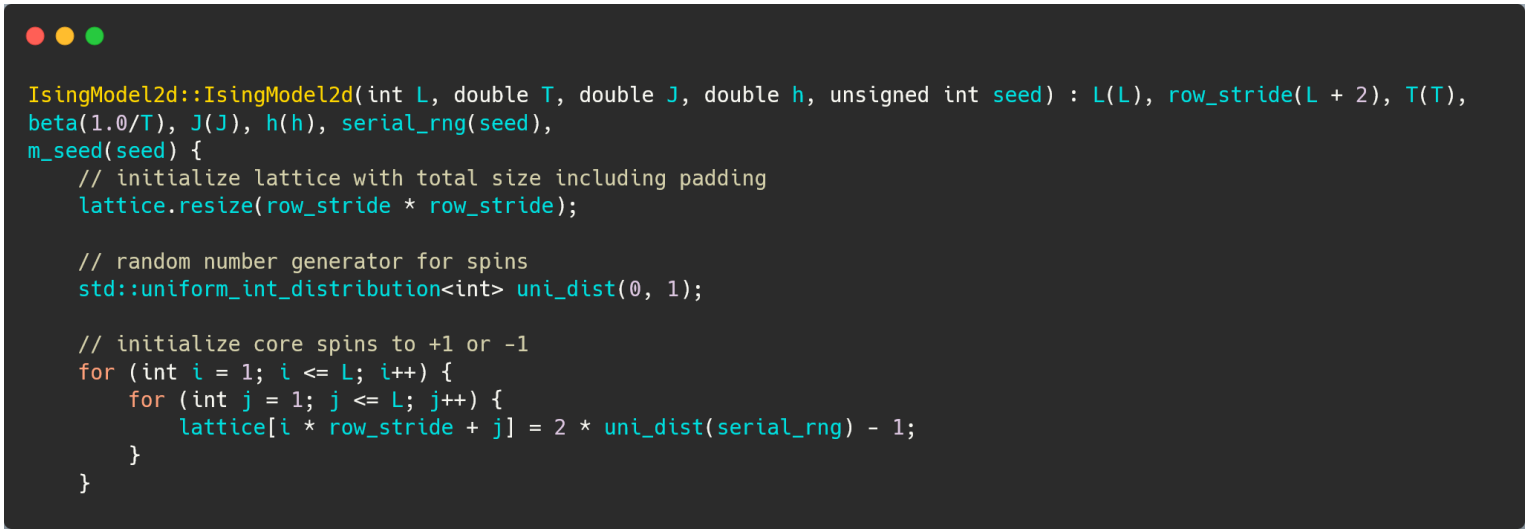
3. Benchmarks
   - Physics correctness
   - Performances

Class Structure and Wrapper

```cpp
// ...

enum class Mode {serial, openMP, cuda_global, cuda_shared};

class IsingModel2d{
    public:

        // constructor
        IsingModel2d(int L, double T, double J, double h, unsigned int seed);
        // destructor
        ~IsingModel2d();

        // default cuda block size (only one side)
        int cuda_block_size = 16;

        // ...

private:

        int L;
        int row_stride; // is equal to L + 2, because it includes the padding
        double T;
        double beta; // = 1/T
        double J = 1; // interaction term
        double h = 0; // magnetic field
        // lookup table: this will be essentialy a probability grid to
        // avoid computing exp every time
        float lookup_probs[2][5];

        // CUDA Device Pointers
        int* d_lattice = nullptr;
        float* d_lookup_probs = nullptr;
        // Pointer to a curandStande which is on the device
        void* d_states = nullptr; // void* to hide curandState to g++

        // lattice is already flattened in 1D vector for simplicity on next phases
        std::vector<int> lattice;

        // random number generators
        std::mt19937 serial_rng;
        std::vector<std::mt19937> omp_rngs; // one number per thread to avoid race conditions
        unsigned int m_seed;

        //...
```

- `enum class` for "evolution" modes (Serial, OpenMP, CUDA)
- **Public Methods** for physical observables, updates, and memory management
- **Encapsulation**: Private members handle internal logic via CUDA "wrapper functions"
- **Interoperability**: Python wrapper implemented using `PyBind11`

Constructor - generalities



```cpp
IsingModel2d::IsingModel2d(int L, double T, double J, double h, unsigned int seed) : L(L), row_stride(L + 2), T(T),
beta(1.0/T), J(J), h(h), serial_rng(seed),
m_seed(seed) {
    // initialize lattice with total size including padding
    lattice.resize(row_stride * row_stride);

    // random number generator for spins
    std::uniform_int_distribution<int> uni_dist(0, 1);

    // initialize core spins to +1 or -1
    for (int i = 1; i <= L; i++) {
        for (int j = 1; j <= L; j++) {
            lattice[i * row_stride + j] = 2 * uni_dist(serial_rng) - 1;
        }
    }
```

- Lattice mapped to 1D array (row-major) $\rightarrow$ ensures contiguous memory layout

Lookup probability table


```
// ...

// Compute lookup probabilities to determine spin flips during Metropolis updates.
    // In this way we compute only once and we do only "read" operation in the following,
    // which are more efficient than compute an exponential function at each step
    for (int s_idx = 0; s_idx < 2; ++s_idx) {
        // assign to index 0 the values of spin -1 and
        // to index 1 the values of spin +1
        double s = (s_idx == 0) ? -1.0 : 1.0;

        // for a given spin value, store the probability of spin flip given the neighbors
        for (int i = 0; i < 5; ++i) {
            // Index mapping: index 0 is relative to value of neighbors -4 (i.e., all negative)
            int physical_sum = (i * 2) - 4; // possible sum of neighbors is -4,-2,0,2,4
            // Globally we have this inidex mapping: (sum_neighbors --> index_lookup_table)
            // -4 --> 0 | -2 --> 1 | 0 --> 2 | 2 --> 3 | 4 --> 3

            // compute delta E
            double delta_E = 2.0 * s * (J * physical_sum + h);

            // assign to each position in the lookup prob the correspondent acceptance probability
            lookup_probs[s_idx][i] = (delta_E > 0) ? std::exp(-delta_E * beta) : 1.0;
            }
    }
```

- Pre-computation of Boltzmann weights for all possible ΔE configurations.
- Replaces expensive exponential calls ($e^{-\beta \Delta E}$) with simple read-only memory accesses during the simulation.

Periodic Boundary Conditions - CPU

Implementation (ternary operator):

```
int i_up = (i == 1) ?  L : i - 1;
```

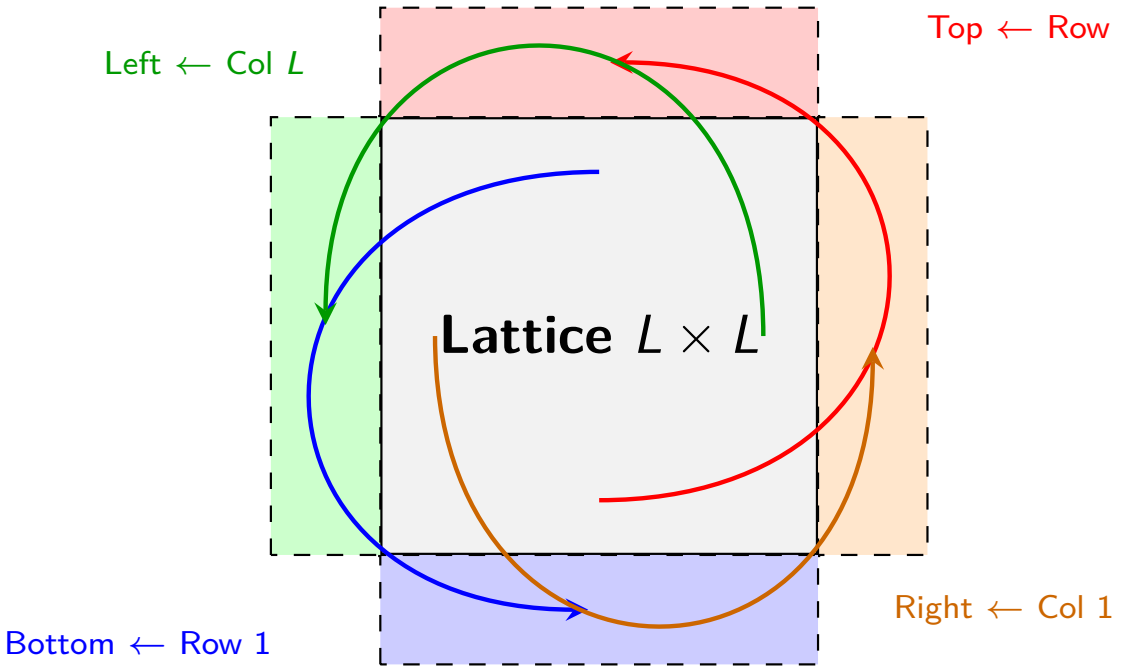
"If at top edge, go to bottom; else, go up."

Why?

- **Speed:** Calculating an index is faster than fetching data from RAM (like using padding layers).
- **Optimization:** Branch prediction unit.

Periodic Boundary Conditions - GPU

- Halo Cells (Padding): the lattice is enlarged to $(L+2) \times (L+2)$.
- Edge elements are copied into the padding rows/columns.



Why Padding for GPU?


```cpp
/********************* Sync padding ***********************/
__global__ void sync_padding_kernel(int* lattice, int L, int row_stride) {
    int k = blockIdx.x * blockDim.x + threadIdx.x + 1;
    if (k <= L) {
        // Copy top/bottom rows
        lattice[0 * row_stride + k] = lattice[L * row_stride + k];
        lattice[(L + 1) * row_stride + k] = lattice[1 * row_stride + k];
        // Copy left/right columns
        lattice[k * row_stride + 0] = lattice[k * row_stride + L];
        lattice[k * row_stride + (L + 1)] = lattice[k * row_stride + 1];
    }
}
```

- **Memory Coalescing:** threads read contiguous memory addresses.
- **Eliminates Branch Divergence:** avoids `if-else` checks for boundaries inside the kernel. All threads execute the exact same instructions (SIMT)

Energy & Magnetization


```cpp
double IsingModel2d::magnetization(Mode mode) {
    double m = 0.0;

    // Serial version
    if (mode == Mode::serial) {
        for (int i = 1; i <= L; i++) {
            for (int j = 1; j <= L; j++) {
                m += lattice[i * row_stride + j];
            }
        }
    }

  //...(OpenMP implementation with Reduction - not used)

  else if (mode == Mode::cuda_global || mode == Mode::cuda_shared){
        // this copy_to_host() is the dominant cost for large lattices
        copy_to_host();
        // compute magnetization on CPU
        return magnetization(Mode::serial);
    }

    return m / (double)(L * L);
}
```

Magnetization per spin

```cpp
double IsingModel2d::energy(Mode mode) {
    double E = 0.0;

    // Serial version
    if (mode == Mode::serial){
        for (int i = 1; i <= L; i++) {
            for (int j = 1; j <= L; j++) {
                int array_index = i * row_stride + j;

                // boundary conditions with ternary operator
                int i_up    = (i == 1) ? L : i - 1;
                int i_down  = (i == L) ? 1 : i + 1;
                int j_left  = (j == 1) ? L : j - 1;
                int j_right = (j == L) ? 1 : j + 1;

                int neighbors = lattice[i_up   * row_stride + j] + // up
                        lattice[i_down * row_stride + j] + //down
                        lattice[i * row_stride + j_left] + // left
                        lattice[i * row_stride + j_right]; // right

                // Interaction energy (halved for double counting) + field energy
                E += -0.5 * J * lattice[array_index] * neighbors - h * lattice[array_index];
            }
        }
    }

    // ... Parallel OpenMP version - NOT USED ...

  // Parallel CUDA version
    else if (mode == Mode::cuda_global || mode == Mode::cuda_shared){
        // this copy_to_host() is the dominant cost for large lattices.
        copy_to_host();

        // delegate the calculation to the CPU implementation
        return energy(Mode::serial);
    }

    return E;
}
```

Energy

SERIAL IMPLEMENTATION

Metropolis criterion

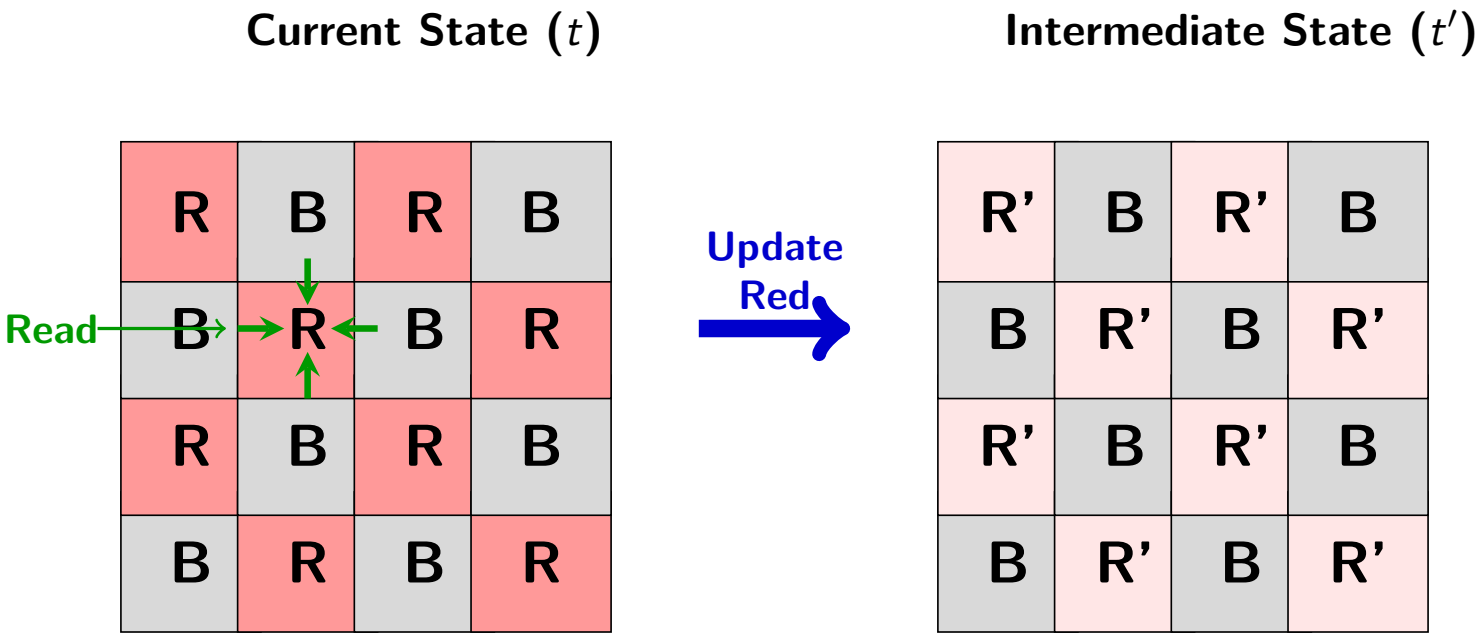

```cpp
void IsingModel2d::Metropolis_update(int i, int j, std::mt19937& rng) {

    // Identify the 1D index of the spin we are trying to flip
    int current_idx = i * row_stride + j;

    // Identify Neighbor Indices using Periodic Boundary Conditions (PBC).
    // check if we are on a boundary (1 or L) and "wrap around" explicitly.

    // if i is 1 (top row), the neighbor above is L (bottom row). Else i-1.
    int i_up    = (i == 1) ? L : i - 1;
    // if i is L (bottom row), the neighbor below is 1 (top row). Else i+1.
    int i_down  = (i == L) ? 1 : i + 1;
    // if j is 1 (left col), the neighbor left is L (right col). Else j-1.
    int j_left  = (j == 1) ? L : j - 1;
    // if j is L (right col), the neighbor right is 1 (left col). Else j+1.
    int j_right = (j == L) ? 1 : j + 1;
    // Sum the neighbors by reading directly from the real lattice locations.
    int neighbors = lattice[i_up   * row_stride + j] +
                    lattice[i_down * row_stride + j] +
                    lattice[i * row_stride + j_left] +
                    lattice[i * row_stride + j_right];

    // Calculate Energy Change (Delta E) via Lookup Table
    // map the current spin (-1 or 1) to a table index (0 or 1)
    int spin_val = lattice[current_idx];
    int spin_idx = (spin_val < 0) ? 0 : 1;

    // map the neighbor sum (-4, -2, 0, 2, 4) to a table index (0, 1, 2, 3, 4)
    int sum_idx  = (neighbors + 4) / 2;

    // use a static distribution to avoid the overhead of reconstructing
    // the object at every single function call
    static std::uniform_real_distribution<double> dist(0.0, 1.0);

    // check the probability against the pre-computed lookup table.
    if (dist(rng) < lookup_probs[spin_idx][sum_idx]) {
        lattice[current_idx] *= -1; // Flip the spin
    }
}
```

PARALLEL IMPLEMENTATIONS

Checkerboard algorithm

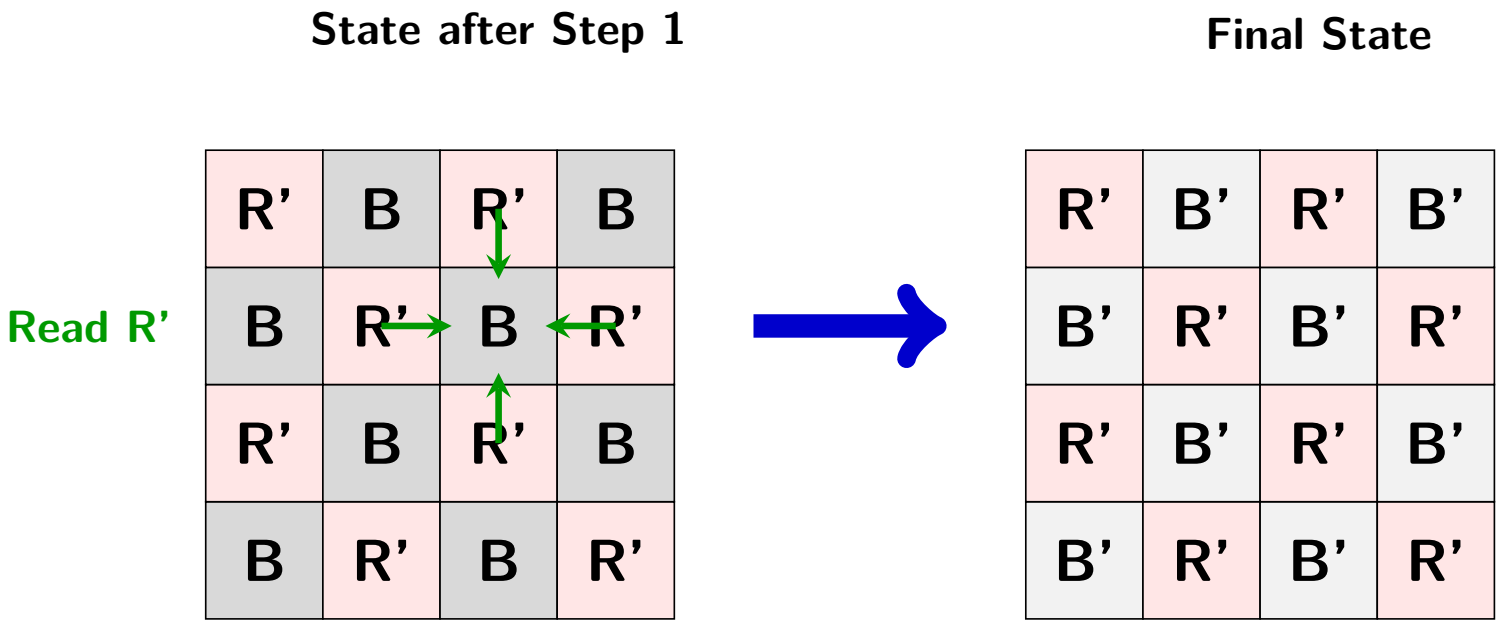
- The update of a spin requires the state of its nearest neighbors
- **Race Condition**: If neighboring spins are updated simultaneously by different threads, the read/write order is undefined
- This leads to non-deterministic results (wrong physics) or requires costly locking mechanisms (performance kill)

Checkerboard algorithm: Step 1 (Red Update)



- All Red cells are updated simultaneously.
- They only read from Black neighbors, which are constant during this step.
- **No Race Conditions**: Threads do not write to the same memory locations concurrently.

Checkerboard Algorithm: Step 2 (Black update)



This scheme will be applied both for OpenMP and CUDA

OpenMP - implementation (1)

```
void IsingModel2d::step_openmp(int steps) {

    //sync_padding();

    // Threads are created here and stay alive for all simulation
    #pragma omp parallel
    {
        // each thread gets its own unique ID and private RNG state
        int thread_id = omp_get_thread_num();
        std::mt19937& thread_rng = omp_rngs[thread_id];
// ...
```

- Start the parallel region: assign a different RNG at each thread

OpenMP - Implementation (2)

```
// ...
// time loop
        for (int s = 0; s < steps; s++) {

            // even spins
            #pragma omp for collapse(2)
            for(int i = 1; i <= L; i++){
                for(int j = 1; j <= L; j++){
                    // Checkerboard condition for even sites
                    if ((i + j) % 2 == 0) {
                        Metropolis_update(i, j, thread_rng);
                    }
                }
            }

            // odd spins
            #pragma omp for collapse(2)
            for(int i = 1; i <= L; i++){
                for(int j = 1; j <= L; j++){
                    // Checkerboard condition for odd sites
                    if ((i + j) % 2 != 0) {
                        Metropolis_update(i, j, thread_rng);
                    }
                }
            }

        } // end of time loop
    } // end of parallel region (threads are destroyed here)
}
```

1. Parallelize the nested for loop
2. Update **"Red"** cells
3. Parallelize the nested for loop
4. Update **"Black"** cells

CUDA - Implementation Strategies

- **Global Memory (Baseline)**: Standard implementation without caching optimizations
- **Shared Memory + read only cache**:
  - **Shared Memory**: Used to store and reuse the thread block's internal lattice
  - **Read-only data cache**: Used for cached, read-only access to **block neighbors**

CUDA - Global memory kernel

Università degli Studi di Padova

```cpp
__global__ void ising_step_global(int* lattice, int L, int row_stride, float* lookup_probs, int color, curandState*
states) {
    // Calculate global coordinates in the lattice (accounting for +1 padding)
    int thread_x = blockIdx.x * blockDim.x + threadIdx.x + 1;
    int thread_y = blockIdx.y * blockDim.y + threadIdx.y + 1;

    // Boundary check: ensure the thread is within the LxL lattice
    if (thread_x <= L && thread_y <= L) {

        // Checkerboard update logic
        // Only threads matching the current type are updated during this pass
        // Here we used color becuse in general there is the map:
        // even --> red | odd --> black

        if ((thread_x + thread_y) % 2 == color) {
            int global_index = thread_y * row_stride + thread_x;
            int current_spin = lattice[global_index];


            /* direct Global Memory Access:*/
            int sum_neighbors = lattice[(thread_y - 1) * row_stride + thread_x] + // Top neighbor
                                lattice[(thread_y + 1) * row_stride + thread_x] + // Bottom neighbor
                                lattice[thread_y * row_stride + (thread_x - 1)] + // Left neighbor
                                lattice[thread_y * row_stride + (thread_x + 1)]; // Right neighbor

            // Map current state and neighbor sum to the 1D lookup table index
            int row = (current_spin == -1) ? 0 : 1;
            int col = (sum_neighbors + 4) / 2;

            // Generate a random float using the pre-initialized CURAND state
            float rand_val = curand_uniform(&states[global_index]);

            // Metropolis Acceptance Criterion
            if (rand_val < lookup_probs[row * 5 + col]) {
                lattice[global_index] = -current_spin;
            }
        }
    }
}
```

CUDA - Shared memory kernel (1)


```cpp
template <int BLOCK_SIZE>
__global__ void ising_step_shared(int* lattice, int L, int row_stride, float* lookup_probs, int color, curandState*
states) {

    // local coordinates
    int thread_x = threadIdx.x;
    int thread_y = threadIdx.y;

    // global coordinates
    int x = (blockIdx.x * blockDim.x) + thread_x + 1;
    int y = (blockIdx.y * blockDim.y) + thread_y + 1;

    int global_index = y * row_stride + x;

    // shared memory
    __shared__ int my_cache[BLOCK_SIZE][BLOCK_SIZE];

    // load from Global to Shared
    if (x <= L && y <= L) {
        my_cache[thread_y][thread_x] = lattice[global_index];
    }

    // wait for all threads to load their spin
    __syncthreads();

//...
```

1. Define local (block-relative) and global (lattice-relative) indices
2. Instantiate `my_cache`. The size is determined at compile-time via the `BLOCK_SIZE` template parameter
3. Synchronously load the lattice portion from Global to Shared memory using `__syncthreads()`

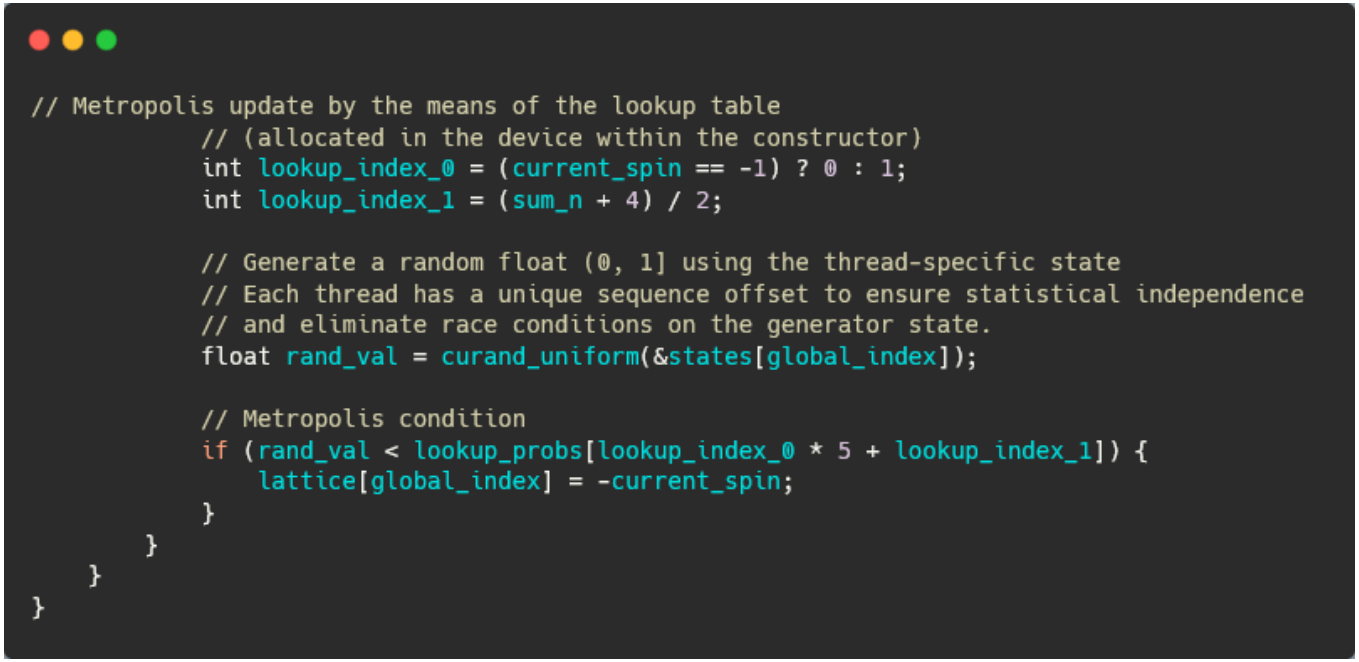
CUDA - Shared memory kernel (2)


```
// UPDATE LOGIC
    if (x <= L && y <= L) {
        if ((x + y) % 2 == color) {

            int current_spin = my_cache[thread_y][thread_x];
            int sum_n = 0;

            // left neighbor
            if (thread_x > 0) {
                sum_n += my_cache[thread_y][thread_x - 1];
            }
            else{
                sum_n += __ldg(&lattice[global_index - 1]);
            }

            // right neighbor
            if (thread_x < BLOCK_SIZE - 1){
                sum_n += my_cache[thread_y][thread_x + 1];
            }
            else {
                sum_n += __ldg(&lattice[global_index + 1]);
            }

            // up neighbor
            if (thread_y > 0){
                sum_n += my_cache[thread_y - 1][thread_x];
            }
            else{
                sum_n += __ldg(&lattice[global_index - row_stride]);
            }

            // down neighbor
            if (thread_y < BLOCK_SIZE - 1){
                sum_n += my_cache[thread_y + 1][thread_x];
            }
            else {
                sum_n += __ldg(&lattice[global_index + row_stride]);
            }
```

1. Checkerboard algorithm based on "color"
2. Sum of the neighbors: if it belongs to the bulk $\rightarrow$ Shared Memory, otherwise read it from the Texture cache.

`__ldg()` **utility:** the hardware-managed Read-Only cache provides high-bandwidth access to "halo" spins that reside in Global Memory but are constant for the duration of the kernel.

CUDA - Shared memory kernel (3)



```
// Metropolis update by the means of the lookup table
            // (allocated in the device within the constructor)
            int lookup_index_0 = (current_spin == -1) ? 0 : 1;
            int lookup_index_1 = (sum_n + 4) / 2;

            // Generate a random float (0, 1] using the thread-specific state
            // Each thread has a unique sequence offset to ensure statistical independence
            // and eliminate race conditions on the generator state.
            float rand_val = curand_uniform(&states[global_index]);

            // Metropolis condition
            if (rand_val < lookup_probs[lookup_index_0 * 5 + lookup_index_1]) {
                lattice[global_index] = -current_spin;
            }
        }
    }
}
```

- Identify the probability in the lookup table and flip the spin accordingly

Steps CUDA


```cpp
/****************** UPDATE STEP CUDA ********************/

void IsingModel2d::step_cuda_global(){
    // synchronize the padding
    launch_sync_padding_gpu(d_lattice, L, row_stride);
    // update even indices
    launch_ising_global(d_lattice, L, row_stride, d_lookup_probs, 0, d_states, cuda_block_size);

    launch_sync_padding_gpu(d_lattice, L, row_stride);
    // update odd indices
    launch_ising_global(d_lattice, L, row_stride, d_lookup_probs, 1, d_states, cuda_block_size);
}

void IsingModel2d::step_cuda_shared(){
    // synchronize the padding
    launch_sync_padding_gpu(d_lattice, L, row_stride);
    // update even indices
    launch_ising_shared(d_lattice, L, row_stride, d_lookup_probs, 0, d_states, cuda_block_size);

    launch_sync_padding_gpu(d_lattice, L, row_stride);
    // update odd indices
    launch_ising_shared(d_lattice, L, row_stride, d_lookup_probs, 1, d_states, cuda_block_size);
}
```

Synchronize the padding layers after each half-update

Table of Contents

1. Introduction
   - Physics
   - Hardware

2. Implementation
   - General structure
   - Serial implementation
   - Parallel implementations

3. Benchmarks
   - Physics correctness
   - Performances

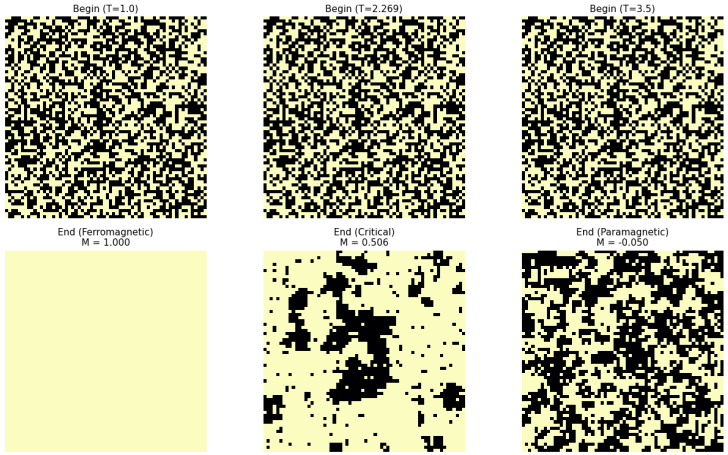
Physics analysis

To check the algorithm correctness we will do the following tests:

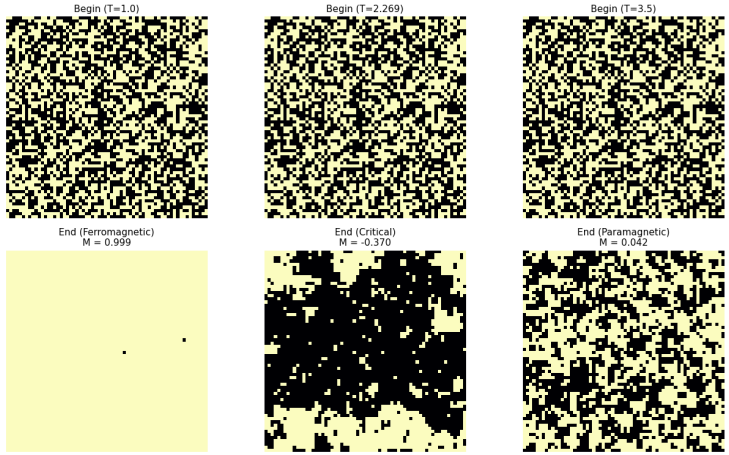
- Spontaneous magnetization ($h = 0$)
- Energy minimization during the simulation steps ($h = 0$)
- (h, T) phase diagram

Visualizations

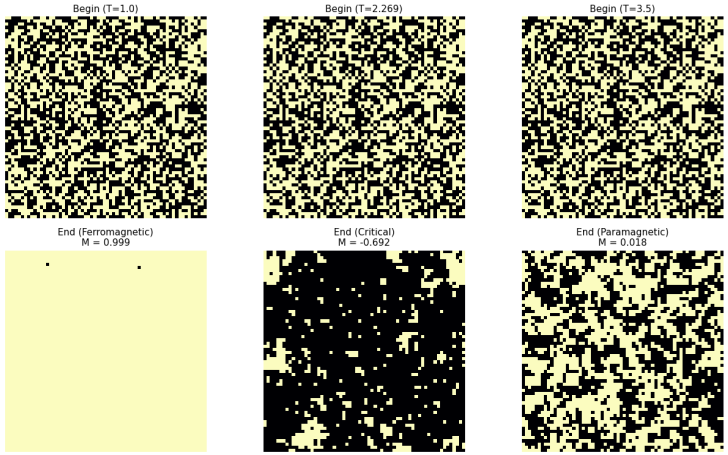
Serial (CPU Single Core)



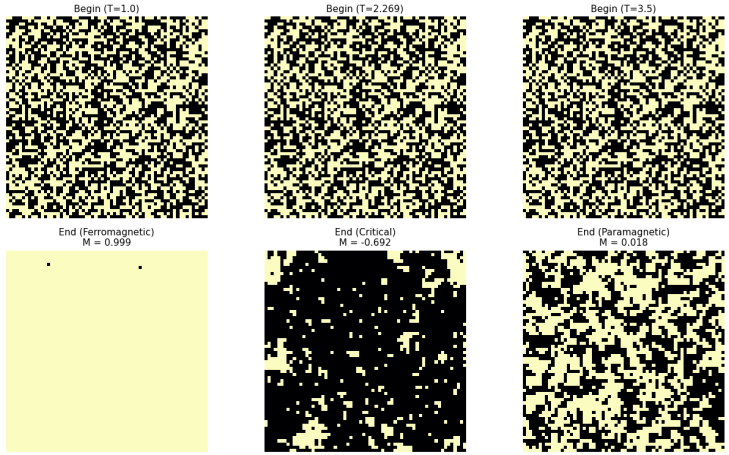
OpenMP (CPU Multi Core)



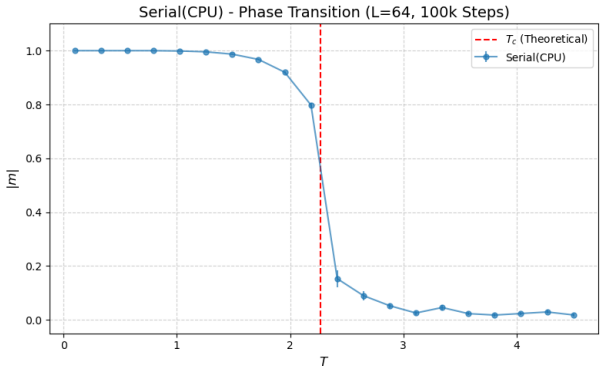
CUDA Global Memory

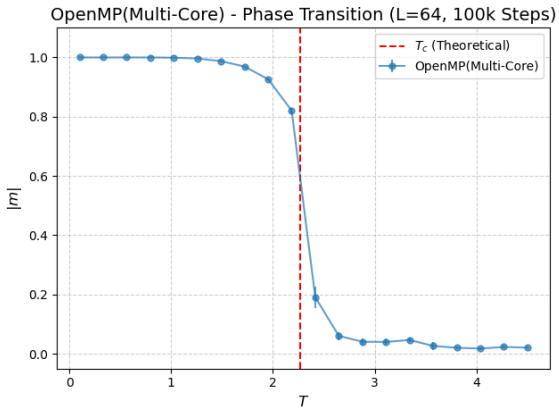


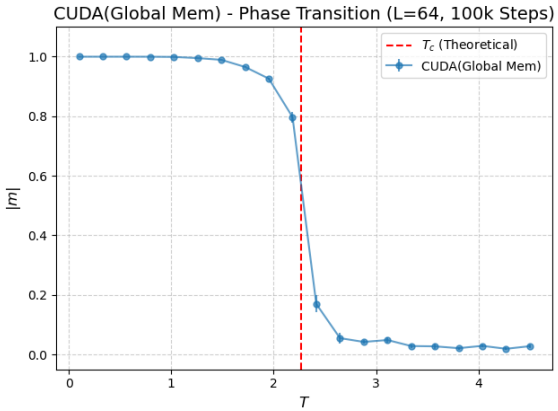
CUDA Shared Memory

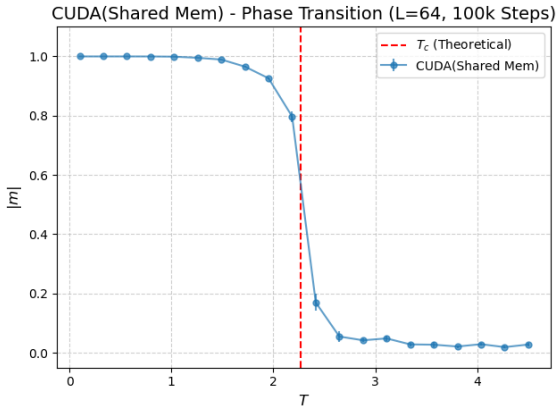


Phase transition |m| vs T



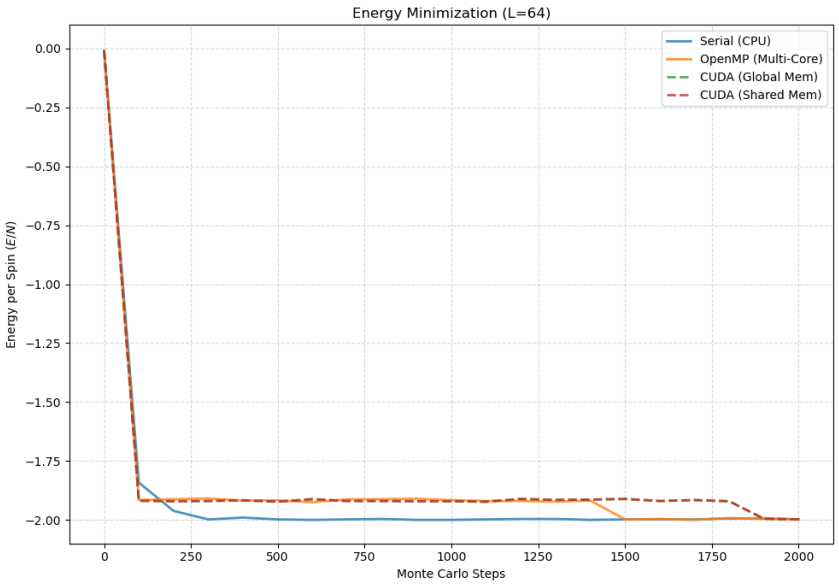






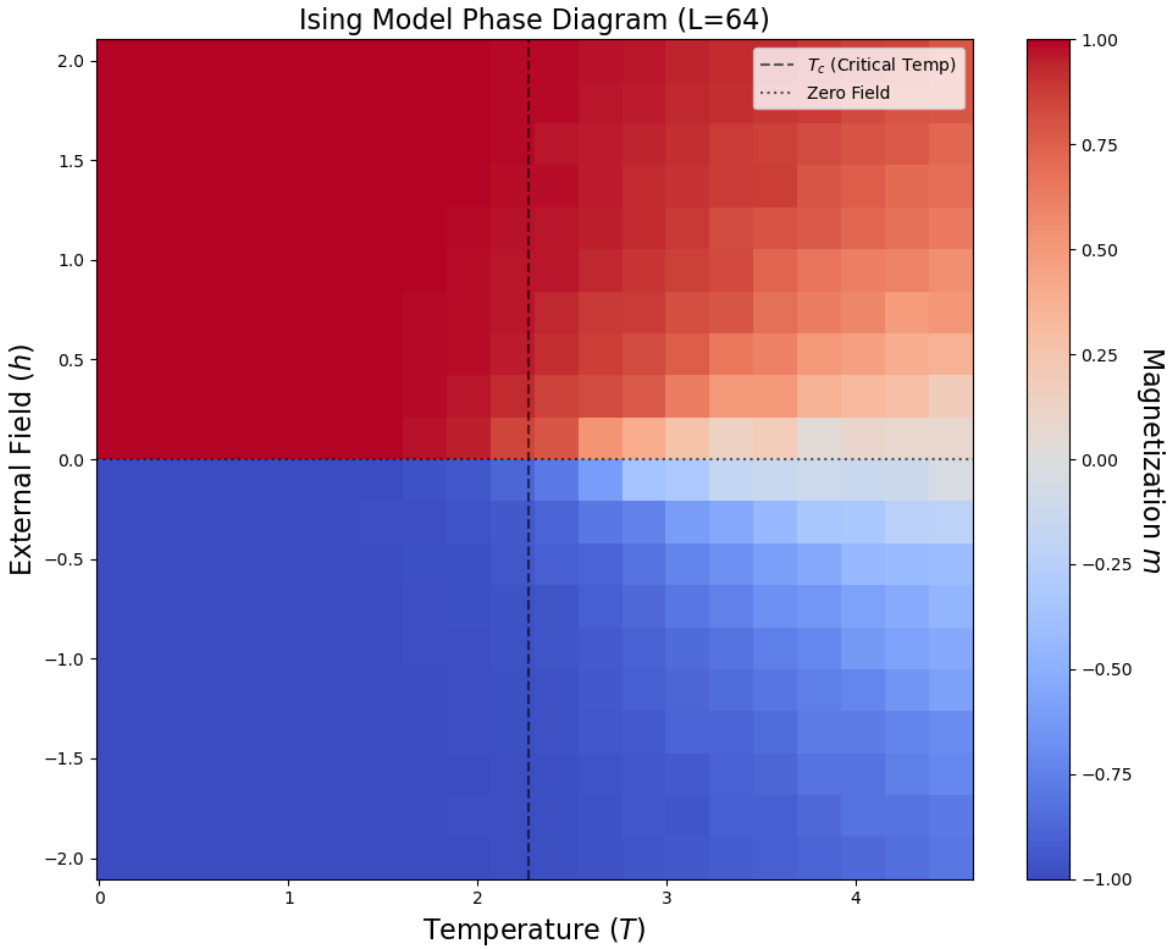
Note: The deviation from the Onsager exact solution is caused by finite size effects ($L = 64$).

Energy minimization



- All implementations successfully lead to the minimization of the system's energy.
- **Note:** The CUDAs implementations likely got trapped in a **metastable state** (local minimum).

Phase diagram (h,T)



- Critical temperature T_c raises in presence of a magnetic field

Performance analysis

We check 4 different aspects:

- OpenMP scalability
- CUDA Kernel tuning
- Execution time and throughput analysis
- Limiting scaling

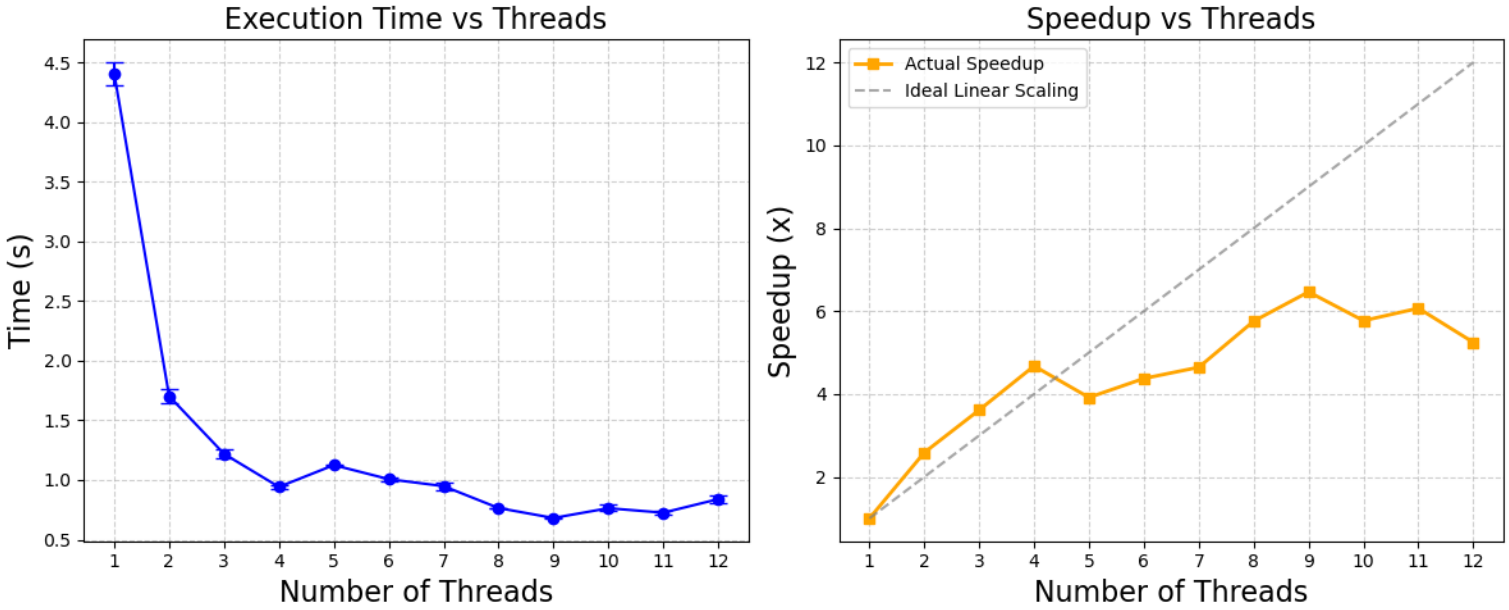
Benchmark Methodology

Time measurements were performed using the `time` module in `Python`, specifically the function `time.perf_counter()`:

- The measurement captures the total elapsed time on the Host (CPU).
- **CUDA Synchronization:** For GPU implementations, an explicit `cudaDeviceSynchronize()` is called before stopping the timer.
  - Without this, the timer would only measure the (negligible) kernel launch latency, as CUDA kernels are asynchronous.
- **Overheads:** Includes Python interpreter overhead and driver/kernel launch latencies (negligible for large L and high number of steps).

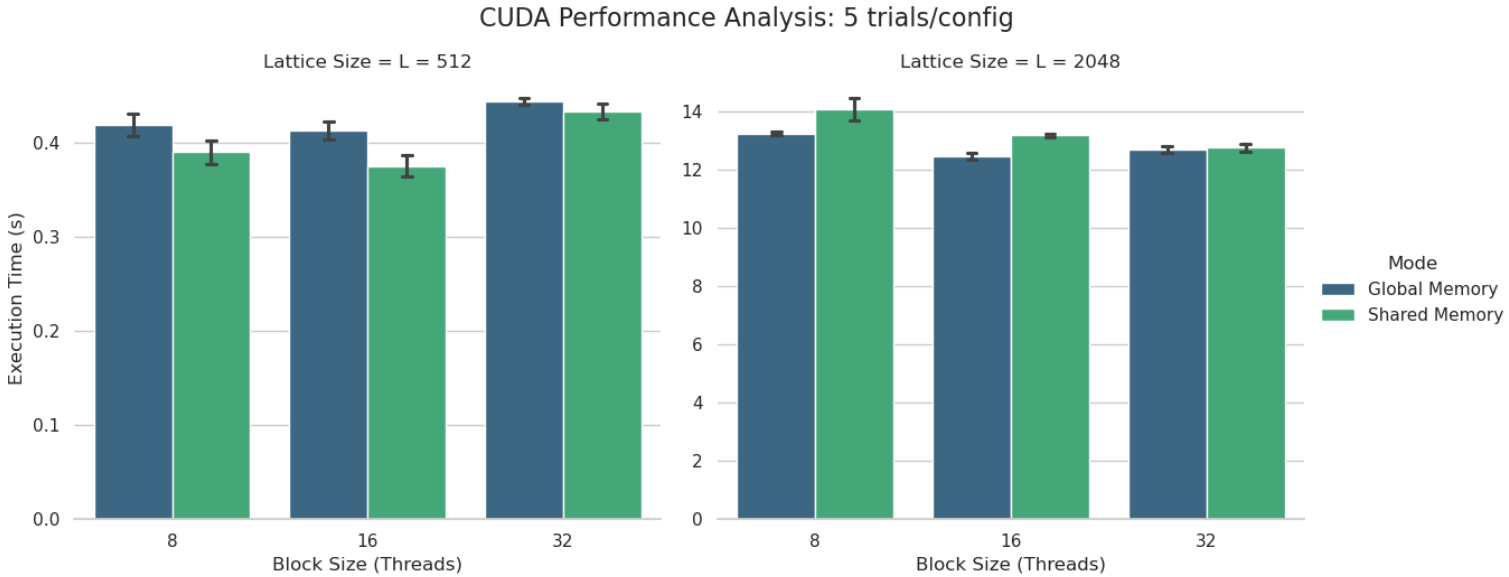
To be more precise and exclude overheads (on GPU), `cuda events` should be used instead

OpenMP vs n_threads



- In accordance with the CPU specifications (4 P-cores / 4 E-cores / 12 logical threads).
- Decrease at n_threads $= 12 \rightarrow$ (likely) context switching due to resource saturation.

CUDA vs block_size (1)



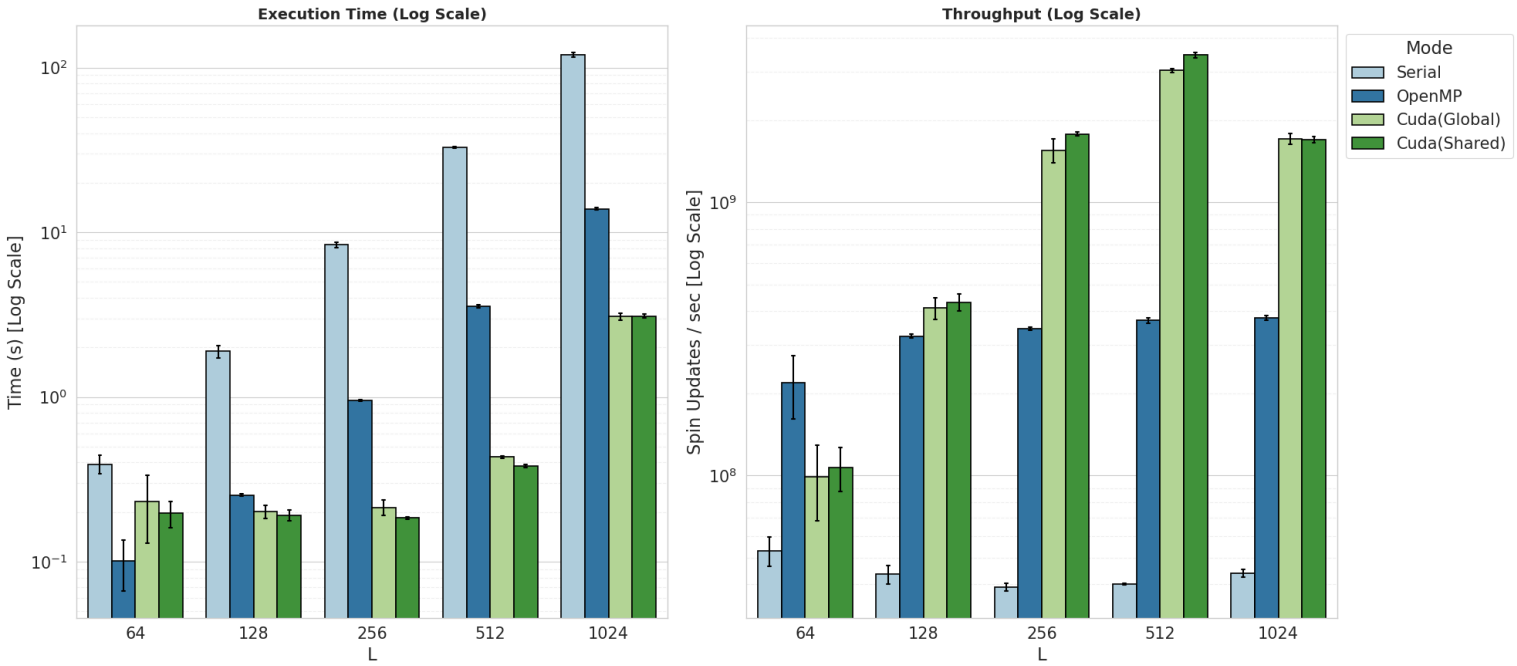
- **Latency-bound regime ($L \leq 512$):**
  The problem is accessing the single data. The dataset fits within the L2 cache. **Shared Memory** outperforms Global Memory by minimizing access latency.
- **Bandwidth-bound regime ($L \geq 512$):**
  The data size exceeds L2 capacity. Performance gap vanishes $\rightarrow$ simulation becomes limited by **VRAM bandwidth** rather than latency (see "Scaling" slide for better details).

Performance analysis: block size effects

- **Small Blocks (8×8) at $L = 2048$:**
  - **Too many borders:** We load more "ghost cells" (neighbors) relative to the actual spins we update.
  - **Management Cost:** The GPU wastes time managing thousands of tiny tasks instead of calculating.

- **Large Blocks (32×32) at $L = 512$:**
  - **Wasted Space:** The blocks are too big to fit perfectly in the hardware, leaving empty slots.
  - **Waiting Time:** At `__syncthreads()`, if one thread is slow, 1023 others must wait doing nothing.

Execution time



- `OpenMP` is the fastest for small lattices (**less overhead costs**)
- GPU's results are in accordance with the previous block analysis

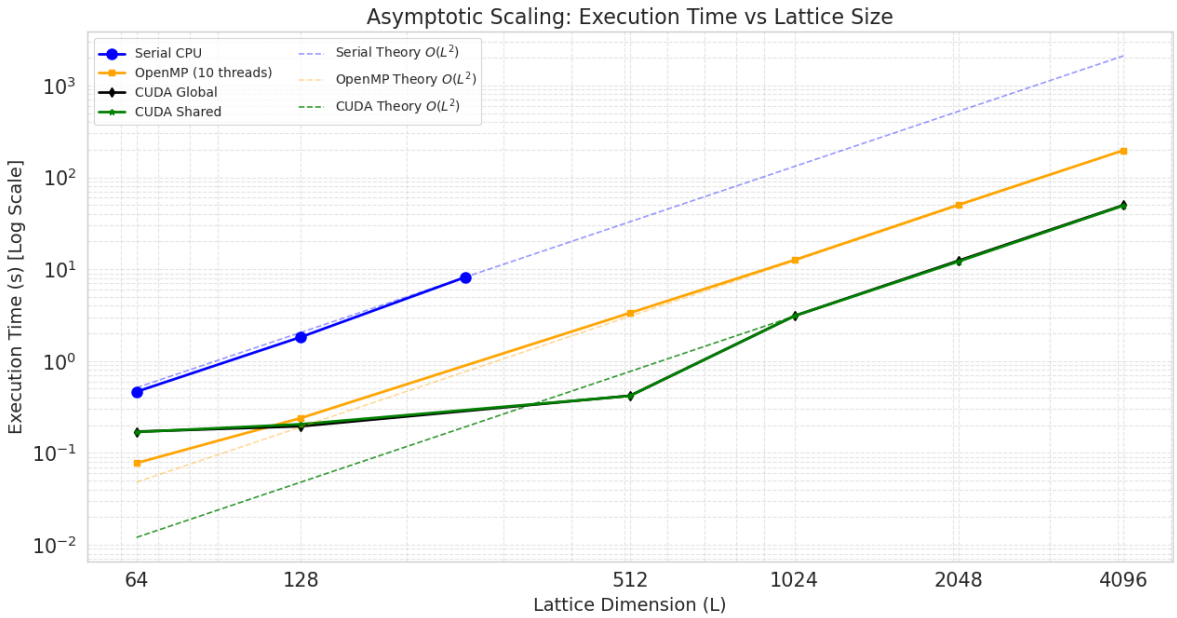
Throughput Analysis

In this analysis, the throughput is defined as the number of spin updates per second:

$$\nu_{\text{updates}} = \frac{L^2 \cdot \text{steps}}{T_{\text{execution}}}$$

- **CPU Implementation:** Throughput remains almost constant (or slightly decreasing) as L increases.
- **GPU Implementation:**
  - *Latency-bound Regime:* Throughput increases with L as hardware occupancy improves.
  - *Bandwidth-bound Regime:* Throughput reaches a plateau (becomes constant) as VRAM bandwidth saturates.

Asymptotic Scaling Analysis



- **CPU (Serial & OpenMP):** Both implementations exhibit the expected theoretical scaling of $\mathcal{O}(L^2)$ across the entire range of lattice sizes.
- **GPU (CUDA):**
  - **Small L:** dominated by constant overheads (kernel launch latency).
  - $L \geq 512$**:** the scaling converges to the asymptotic $\mathcal{O}(L^2)$ regime.

Conclusions & Future Work

- `OpenMP` is superior (or at least comparable) for small lattices ($L < 128$) where the overhead of PCIe data transfer and kernel launch outweighs the parallel benefits.
- For large systems ($L \geq 512$), the GPU implementation (Shared/Global) provides massive speedups, effectively hiding memory latency.
- The large L2 Cache on Ada Lovelace minimized the performance gap between Global and Shared Memory implementations.

Future Extensions:

- Scaling the domain decomposition using `MPI` to simulate massive lattices
- Implementing parallel reduction (e.g., Thrust) to calculate m and E directly on the GPU

Thank you for the attention

Random Number Generation - serial

- Private member `serial_rng` handles the spin flip probabilities
- It is an object of the standard library `<random>`, which:
  - Uses the **Mersenne Twister** engine (`std::mt19937`) for high-quality pseudo-random numbers
  - Provides a uniform distribution in the range $[0, 1)$ to be compared with the Metropolis acceptance probability P
  - Allows for **reproducible results** by using a fixed seed during the development and benchmarking phase
- **Note**: This generator is used for the Serial and OpenMP versions, whereas CUDA requires a different approach (`cuRAND`) to handle parallel generation

OpenMP - RNGs

- Naive parallel RNG (shared state) $\rightarrow$ **race conditions** or **serialization** (bottleneck)
- Solution: 1 RNG per thread (initialzied in the constructor)

```
// ... Constructor

// initialize RNG engines for parallel threads in OPENMP
  int max_threads = omp_get_max_threads();
  for (int i = 0; i < max_threads; i++) {
      omp_rngs.emplace_back(seed + i + 1);
  }
// ...
```

CUDA - Helper functions (1)

Wrappers

```cpp
void* gpu_alloc(size_t size) {
    void* ptr = nullptr;
    cudaMalloc(&ptr, size);
    return ptr;
}

void gpu_free(void* ptr) {
    cudaFree(ptr);
}

void gpu_memcpy_to_device(void* dest, const void* src, size_t size) {
    cudaMemcpy(dest, src, size, cudaMemcpyHostToDevice);
}

void gpu_memcpy_to_host(void* dest, const void* src, size_t size) {
    cudaMemcpy(dest, src, size, cudaMemcpyDeviceToHost);
}

void launch_cuda_sync(){
    /*Wrapper to synchronize blocks*/
    cudaDeviceSynchronize();
}
```

Class functions

```cpp
// allocate memory on device
void IsingModel2d::allocate_cuda() {
    size_t lattice_bytes = row_stride * row_stride * sizeof(int);
    size_t states_bytes = row_stride * row_stride * 70; // excess CuRand state dimension estimation

    // allocate lattice, curandstates and lookup table
    d_lattice = (int*)gpu_alloc(lattice_bytes);
    d_states = gpu_alloc(states_bytes);
    d_lookup_probs = (float*)gpu_alloc(10 * sizeof(float));
}

void IsingModel2d::copy_to_device() {
    // it copies the lattice from host to device
    gpu_memcpy_to_device(d_lattice, lattice.data(), lattice.size() * sizeof(int));
}

void IsingModel2d::copy_to_host() {
    // it copties the lattice from device to host
    gpu_memcpy_to_host(lattice.data(), d_lattice, lattice.size() * sizeof(int));
}

void IsingModel2d::deallocate_cuda() {
    // free the memory
    gpu_free(d_lattice);
    gpu_free(d_states);
    gpu_free(d_lookup_probs);
}

void IsingModel2d::upload_lookup_probs(){
    // upload to device the lookup probability table
    float host_lookup[10];
    for (int i = 0; i < 2; i++){
        for (int j = 0; j < 5; j++){
            host_lookup[i*5 + j] = lookup_probs[i][j];
        }
    }

    gpu_memcpy_to_device(d_lookup_probs,&host_lookup,sizeof(float) * 10);
}

void IsingModel2d::device_synchronize(){
    launch_cuda_sync();
}
```

CUDA - Helper functions (2)

Synchronize the padding on the device

```cpp
/******************** Sync padding **********************/
__global__ void sync_padding_kernel(int* lattice, int L, int row_stride) {
    int k = blockIdx.x * blockDim.x + threadIdx.x + 1;
    if (k <= L) {
        // Copy top/bottom rows
        lattice[0 * row_stride + k] = lattice[L * row_stride + k];
        lattice[(L + 1) * row_stride + k] = lattice[1 * row_stride + k];
        // Copy left/right columns
        lattice[k * row_stride + 0] = lattice[k * row_stride + L];
        lattice[k * row_stride + (L + 1)] = lattice[k * row_stride + 1];
    }
}
```

- Used only one index to access the interested elements

```cpp
void launch_sync_padding_gpu(int* d_lattice, int L, int row_stride) {

    // use fixed number of threads for padding
    int threads = 256;
    int grid = (L + threads - 1) / threads;

    sync_padding_kernel<<<grid, threads>>>(d_lattice, L, row_stride);
}
```

- 1D block to launch the kernel (CUDA interprets both threads and grid as a `dim3` object, even if are declared as `int`

CUDA - Random Number Generation

We implemented a dedicated kernel and a host wrapper to efficiently handle the parallel RNG initialization using the **cuRAND** library.

Initialization Kernel

```cpp
#include <cuda_runtime.h>
#include <curand.h>
#include <curand_kernel.h>
#include <iostream>
#include <cstdio>
#include "Ising_gpu_interface.h"
/******************** Setup for RNGs **********************/
__global__ void setup_rand_kernel(curandState* states, unsigned int seed, int L, int row_stride) {
    int x = blockIdx.x * blockDim.x + threadIdx.x + 1;
    int y = blockIdx.y * blockDim.y + threadIdx.y + 1;

    if (x <= L && y <= L) {
        int index = y * row_stride + x;

        // itialize each thread with a Curandstate:
        // - seed: global starting point
        // - index: sequence number to ensure independent, non-overlapping random sequences
        // - 0: initial subsequence offset
        // - states[index]: destination in global memory for this thread's state
        curand_init(seed, index, 0, &states[index]);
    }
}
```

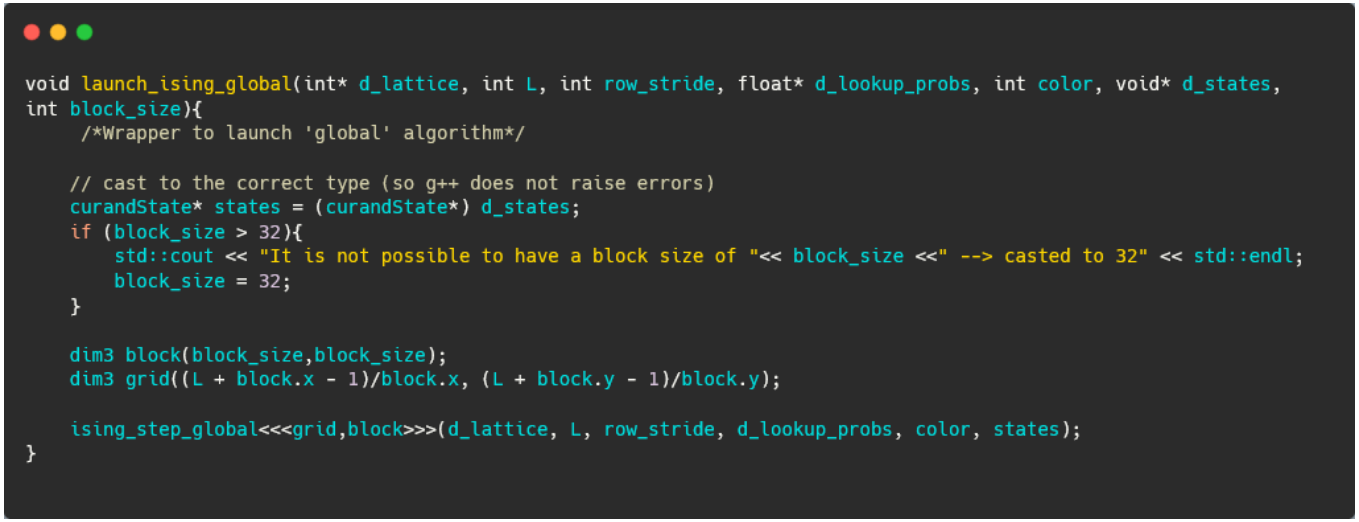
Host Wrapper

```cpp
// ========================================
// WRAPPERS (KERNEL LAUNCH)
// ========================================

void launch_setup_rng(void* d_states, unsigned int seed, int L, int row_stride) {
    curandState* states = (curandState*)d_states;

    // fixed block size here
    dim3 block(16, 16);
    dim3 grid((L + block.x - 1) / block.x, (L + block.y - 1) / block.y);

    setup_rand_kernel<<<grid, block>>>(states, seed, L, row_stride);

    cudaDeviceSynchronize();
}
```

The `curandState` structure holds the generator's internal state (seed, sequence, offset) for each thread.

- `curand_init` is computationally expensive → execute it **only once** at the start.
- The states are stored in Global Memory and reused by the Metropolis kernels at each simulation step.

CUDA - Global memory kernel (wrapper)



```
void launch_ising_global(int* d_lattice, int L, int row_stride, float* d_lookup_probs, int color, void* d_states,
int block_size){
     /*Wrapper to launch 'global' algorithm*/

    // cast to the correct type (so g++ does not raise errors)
    curandState* states = (curandState*) d_states;
    if (block_size > 32){
        std::cout << "It is not possible to have a block size of "<< block_size <<" --> casted to 32" << std::endl;
        block_size = 32;
    }

    dim3 block(block_size,block_size);
    dim3 grid((L + block.x - 1)/block.x, (L + block.y - 1)/block.y);

    ising_step_global<<<grid,block>>>(d_lattice, L, row_stride, d_lookup_probs, color, states);
}
```

- Control condition regarding the limit of 1024 threads in a single block.

CUDA - Shared memory wrapper

UNIVERSITÀ DEGLI STUDI DI PADOVA

```cpp
void launch_ising_shared(int* d_lattice, int L, int row_stride, float* d_lookup_probs, int color, void* d_states,
int block_size) {
    // cast to the correct type (so g++ does not raise errors)
    curandState* states = (curandState*)d_states;

    dim3 block(block_size, block_size);
    dim3 grid((L + block.x - 1) / block.x, (L + block.y - 1) / block.y);

    // Choose the correct template based on runtime block_size
    switch(block_size) {
        case 8:
            ising_step_shared<8><<<grid, block>>>(d_lattice, L, row_stride, d_lookup_probs, color, states);
            break;
        case 16:
            ising_step_shared<16><<<grid, block>>>(d_lattice, L, row_stride, d_lookup_probs, color, states);
            break;
        case 32:
            ising_step_shared<32><<<grid, block>>>(d_lattice, L, row_stride, d_lookup_probs, color, states);
            break;
        default:
            std::cerr << "ERROR: Block size " << block_size << " not supported! (Use 8, 16, or 32)" << std::endl;

    }
}
```

- Switch to define block size (both for blocks and shared memory)

Update method

Serial

```cpp
/******* UPDATE STEP ********/

void IsingModel2d::update(Mode mode, int steps) {
    switch(mode) {

        /****** SERIAL CPU EXECUTION ******/
        case Mode::serial:
            for (int i = 0; i < steps; i++) {
                // The serial step already includes sync_padding() internally
                step_serial();
            }
            break;
```

OpenMP

```cpp
//...
/****** PARALLEL OPENMP EXECUTION *****/

        case Mode::openMP:
            step_openmp(steps);
            break;
//...
```

CUDA

```cpp
/****** PARALLEL CUDA EXECUTION *****/
        case Mode::cuda_global:

            // d_lattice must be already allocated and populated on the GPU.
            // if this is the very first run, ensure copy_to_device() was called before.

            for (int i = 0; i < steps; i++) {
                // perform one full Monte Carlo step (Padding + Red + Black)
                step_cuda_global();
            }

            // wrapper for the blocks synchronization
            launch_cuda_sync();

            // note: We do NOT copy data back to Host here.

            break;

        case Mode::cuda_shared:

            // d_lattice must be already allocated and populated on the GPU.
            // if this is the very first run, ensure copy_to_device() was called before.

            for (int i = 0; i < steps; i++) {
                // perform one full Monte Carlo step (Padding + Red + Black)
                step_cuda_shared();
            }

            // wrapper for the blocks synchronization
            launch_cuda_sync();

            // note: We do NOT copy data back to Host here.

            break;
    }
}
```

Speedup for L = 512 (5k steps)

Mode	**Time (s)**	**Speedup (vs Serial)**
Serial	32.7749	1.00x
OpenMP	3.5483	9.24x
CUDA (Global)	0.4315	75.96x
CUDA (Shared)	0.3793	86.42x

Extra: Simulated annealing (1)

Simulated Annealing is an optimization method that adapts the system's temperature according to a specific cooling schedule during the simulation.
The decay formula is the following:

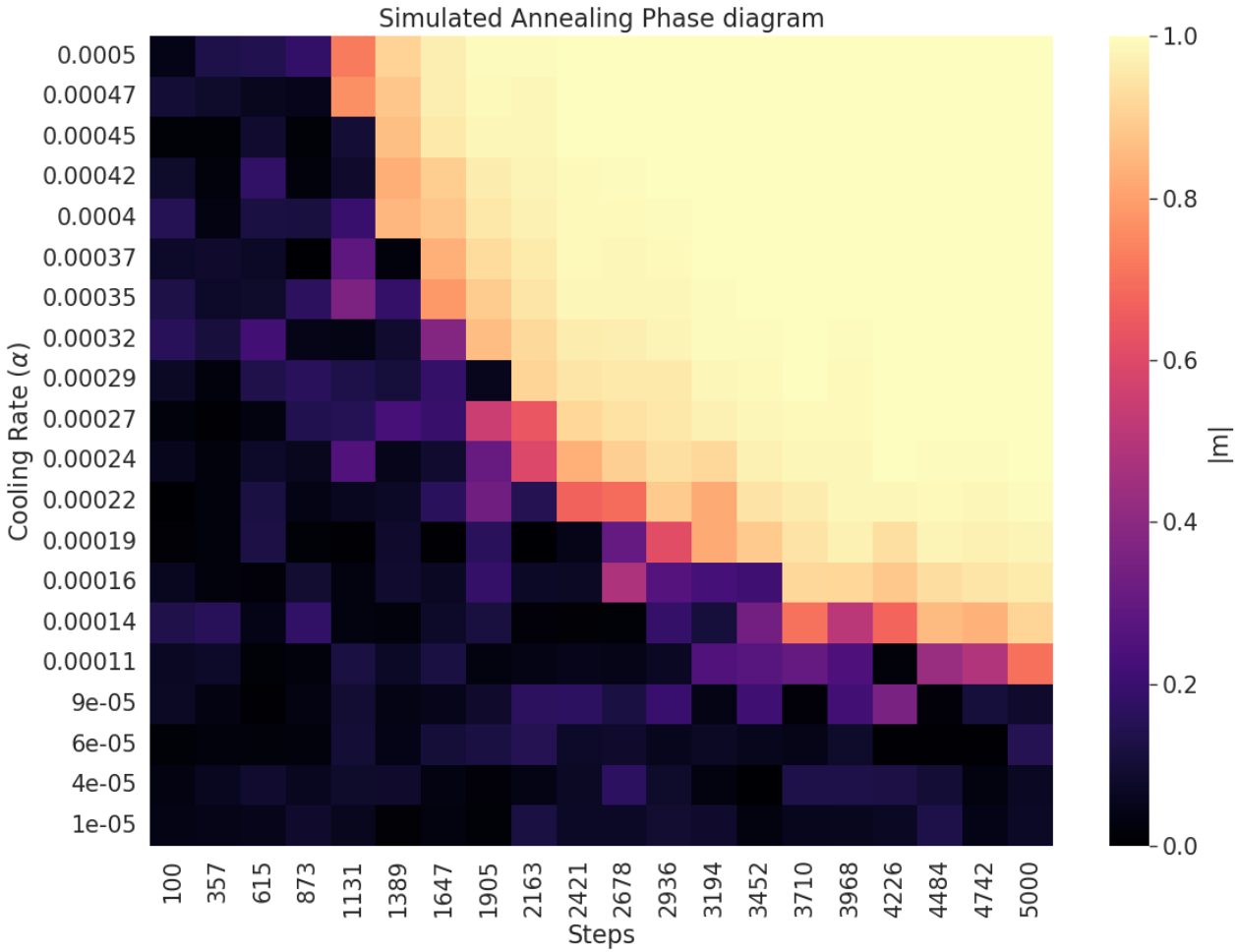
$$T(t) = T_0 \cdot \exp(-\alpha t)$$

- T_0: Initial temperature
- α: Decay rate (cooling speed)
- t: Simulation step

Extra: Simulated Annealing (2)

Parameter Space Analysis (α vs Steps)

(Starting temperature $T_0 = 4$)



Extra: Simulated Annealing (3)

- **Low α (Slow cooling):** The temperature decays too slowly. Within the limited steps, the system remains at high T and remains disordered due to **thermal fluctuations**.
- **Increasing α:** The cooling rate is sufficient to reach the ordered phase (Ground State), allowing the system to **escape local minima** and settle before freezing.