The 2D Ising Model with the Checkerboard Algorithm Using NVIDIA GPUs with CUDA

António, Sebastian, Giorgos and Cyan
High-Performance Parallel Computing

23rd March 2024

Introduction

Ising Model

Note: In this study $k_b = 1$ applies! The Ising model is fundamental in statistical mechanics, representing magnetic spins on a lattice that can assume either up (+1) or down (-1) orientations. These spins interact with their neighbours, favouring either parallel or antiparallel alignment. The system is described by the following Hamiltonian

$$\mathcal{H} = -J \sum_{\langle i,j \rangle} S_i S_j - H \sum_i S_i, \tag{1}$$

where S_i stands for the spin ($S_i = \pm 1$) at lattice site i, and $\langle i, j \rangle$ denotes an interacting spin pair [2] [3]. The coefficient J is the coupling constant and H represents the external magnetic field expressed in units of energy. In this report, we set $J = 1$ and $H = 0$.

The model simulates the collective behaviour of these spins under varying temperatures. At high temperatures, thermal fluctuations cause the spins to be disordered, resulting in no net magnetisation. On the other hand, below a critical temperature, the system transitions to a magnetically ordered state, with the majority of spins aligning in the same direction.

This model is pivotal in computational physics, allowing for the simulation of complex systems through the analysis of binary spin states [2]. It represents a simple yet powerful approach to understanding phase transitions, employing high-performance computing to manage the extensive calculations involved. The primary objective of this report is the implementation of the Ising model, utilising GPU computation and CUDA for increased processing efficiency.

CUDA[7]

CUDA (Compute Unified Device Architecture) is a parallel computing platform created by NVIDIA. It allows us to use a CUDA-enabled GPU for general-purpose computing. CUDA provides a direct way to create and execute programs on GPU's parallel computing cores. With the power of GPUs on batch calculation, CUDA improves computing performance by leveraging the parallel nature of GPUs. If we call functions with simple qualifiers, the functions are executed N times in parallel by N different CUDA threads. These functions are often called kernels. Unlike sequential programming models, this approach is designed to process large blocks of data simultaneously, making it particularly effective for tasks involving mathematics, physics, and graphics. As a platform, CUDA has been widely adopted in various fields for tasks such as machine learning, scientific simulations, and 3D visualization, demonstrating its versatility and power in high-performance computing scenarios.

Ising Model Implementation

Metropolis MCMC

The project's methodology involves updating lattice point spins with a random number generator, and making acceptance decisions based on probabilities, a process characteristic of the *Metropolis Markov Chain Monte Carlo*(MCMC)[4] technique. Simulations are a good approach to this kind of many-body model as the calculation of the partition function Z is very difficult. The *Metropolis* algorithm uses the fact that we know the probability distribution ρ but without the normalization (partition function Z) [2]:

$$Z = \sum_{\omega} exp(-\beta \mathcal{H}(\omega)) \tag{2}$$

$$\rho = \frac{1}{Z} \cdot exp(-\beta \mathcal{H}) \tag{3}$$

with

$$\beta = \frac{1}{T}.$$

The usual Metropolis algorithm summarised:

- Select uniformly lattice point i, j
- Accept/Reject step:

$$\begin{cases} \Delta E < 0: & \text{flip} \\ \Delta E \geq 0: & \text{pick } r \in [0,1] \text{ and flip if } r < e^{-\Delta E \cdot \beta} \text{ otherwise, leave the spin} \end{cases}$$

where for ΔE yields

$$\Delta E = 2 \cdot J \cdot S_{i,j}[S_{i,j+1} + S_{i,j-1} + S_{i+1,j} + S_{i-1,j}] + 2 \cdot H \cdot S_{i,j}.$$

Checkerboard Metropolis

With the so-called *Checkerboard Algorithm*, the grid point to be checked is no longer selected uniformly at random; instead, we utilize the checkerboard pattern (white and black) and loop over the white and black squares independently [1]. In a parallel algorithm, the updates of one colour of the **chessboard must be synchronised** before the spins of the other colour are processed.

The checkerboard pattern in the Metropolis algorithm offers certain advantages. It allows us to separate the black lattice points from the white ones, meaning that we **only need to perform two sets of calculations**, one for each subgroup of lattice points. Since this approach results in the minimum number of divisions, the size of each lattice subgroup is maximized. Thus, this maximizes the parallel computing capabilities of the GPU, making efficient use of its architecture. As a result, the Metropolis operation will be **conducted twice**: once for the black lattice points and once for the white lattice points. As a result, each Metropolis random operation does not influence the other lattice subgroup. Figure3 summarises this.

CUDA Optimisation

To implement the *Checkerboard Algorithm*, the black and white lattices are stored in **two distinct arrays**. The lattices in each array are arranged according to their positions on the checkerboard grid. We then execute the Metropolis Monte Carlo functions to one of the arrays for instance the black array. To calculate the energy difference ΔE each grid point **requires the nearest neighbours** that are stored in the array of the other division. After updating one array, **synchronization is necessary** as the other array needs the updated values. After a certain number of iterations, the CPU calculates the total magnetisation which is the sum of all spins on the lattice. For each iteration and each lattice site, there is a need for a **uniform random number** between 0 and 1 (**Accept/ Reject step**). To achieve this **two additional kernels** were created that initialized the seed for the random flow again for each iteration. The seed was updated (++) after each iteration.

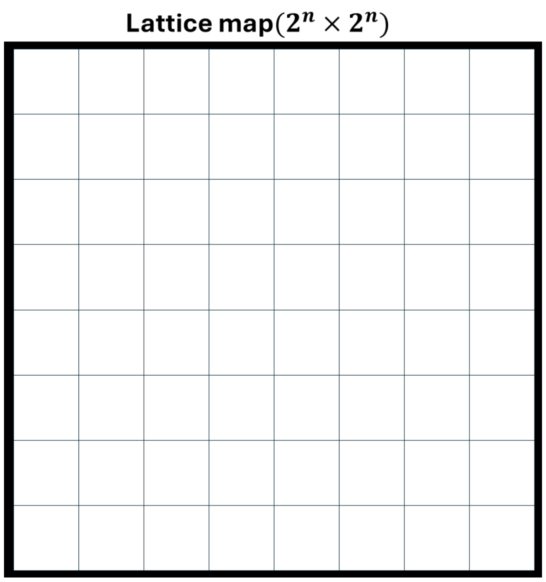

Figure 1: An empty lattice grid, which is conceptually two-dimensional, is represented as a one-dimensional array to facilitate GPU parallelization.

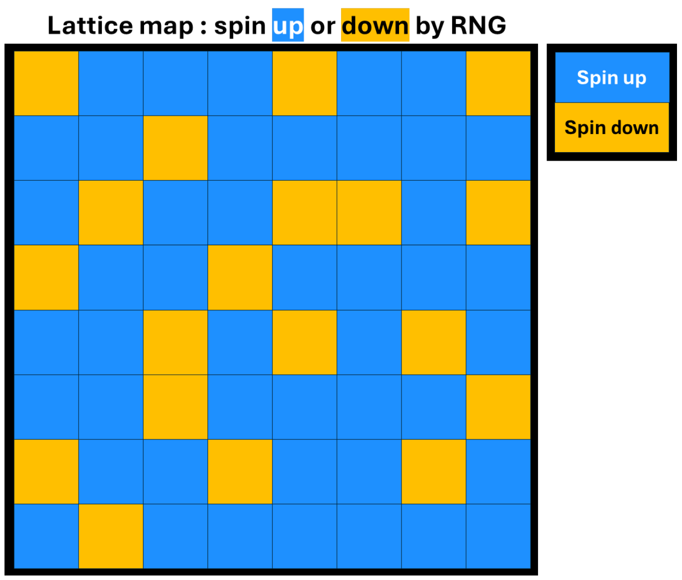

Figure 2: The figure illustrates a lattice grid following the completion of a single Metropolis Monte Carlo step. Yellow lattices denote spin-down states, and blue lattices represent spin-up states.

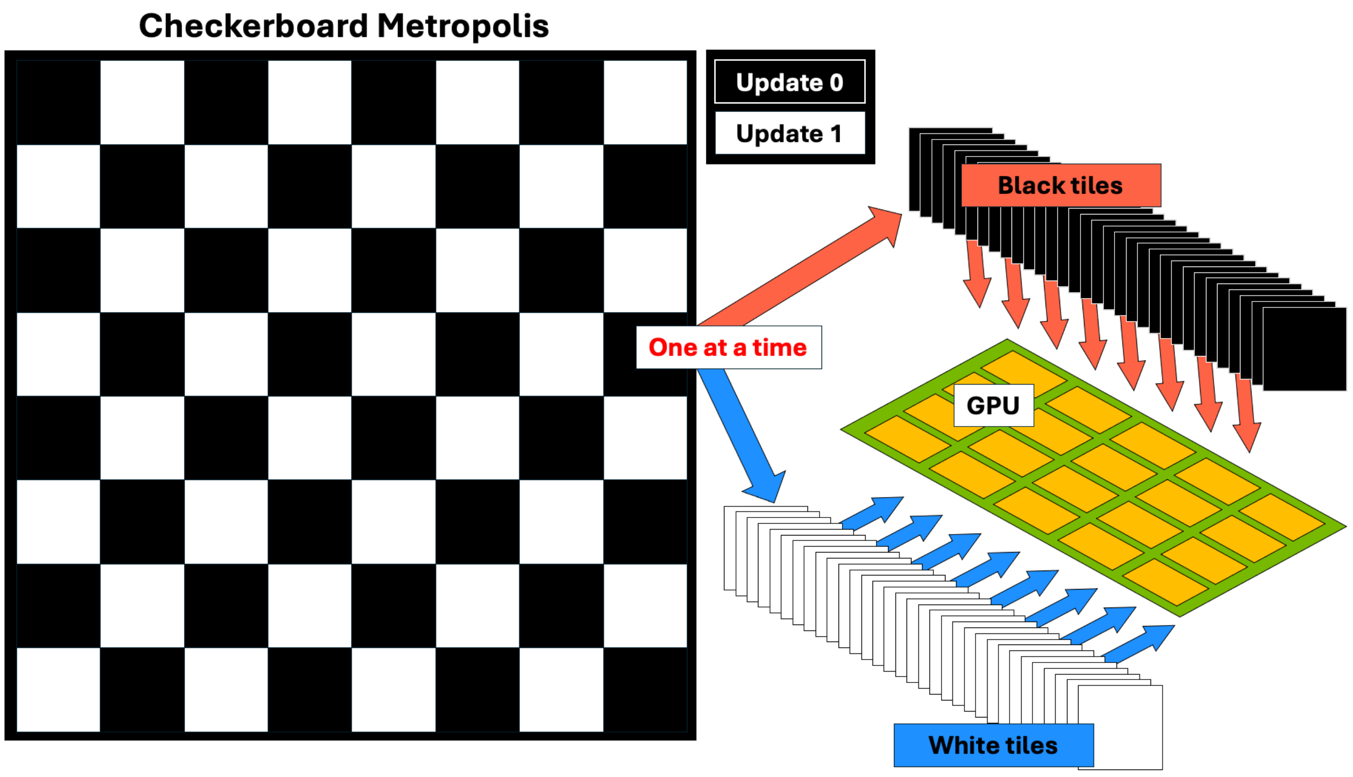

Figure 3: This figure depicts the checkerboard decomposition strategy for the Ising model simulation. The lattice is divided into black and white sublattices (chessboard pattern), which allows for the simultaneous update of non-adjacent tiles by separate GPU threads. The updates are performed in two stages: first, all black tiles are updated (Update 0), then all white tiles (Update 1). This ensures that dependencies between adjacent spins do not lead to conflicts during simultaneous updates, thereby optimizing the use of the GPU's parallel processing capabilities.

GPU Architecture and Profiling

GPU structure

A GPU consists of distinct components each playing a vital role in its operation. The multiprocessors are at the foundation of the GPU. They are the primary computational workers of the GPU, each capable of executing numerous threads simultaneously. These multiprocessors are connected to two tiers of cache memory: the L1 Cache, which is small and fast, allocated to each multiprocessor for immediate data access; and the L2 Cache, which is large and shared amongst multiprocessors. The L2 cache acts as an intermediate buffer to speed up the retrieval of data from the device memory. The device memory, also referred to as DRAM, is a type of global memory that offers a larger capacity compared to L2 cache however, it operates at a slower speed. The shared memory within each multiprocessor provides a fast communication pathway for threads within the same block to share data quickly. Each processor within a multiprocessor has a set of registers, the fastest yet smallest memory spaces, utilized for storing immediate working data. The instruction unit manages the operation of the processors by decoding and dispatching instructions for execution. Collectively, these components constitute the GPU, often referred to as the 'device' in contrast to the CPU, which is commonly called the 'host'. This architecture is depicted in Figure4.

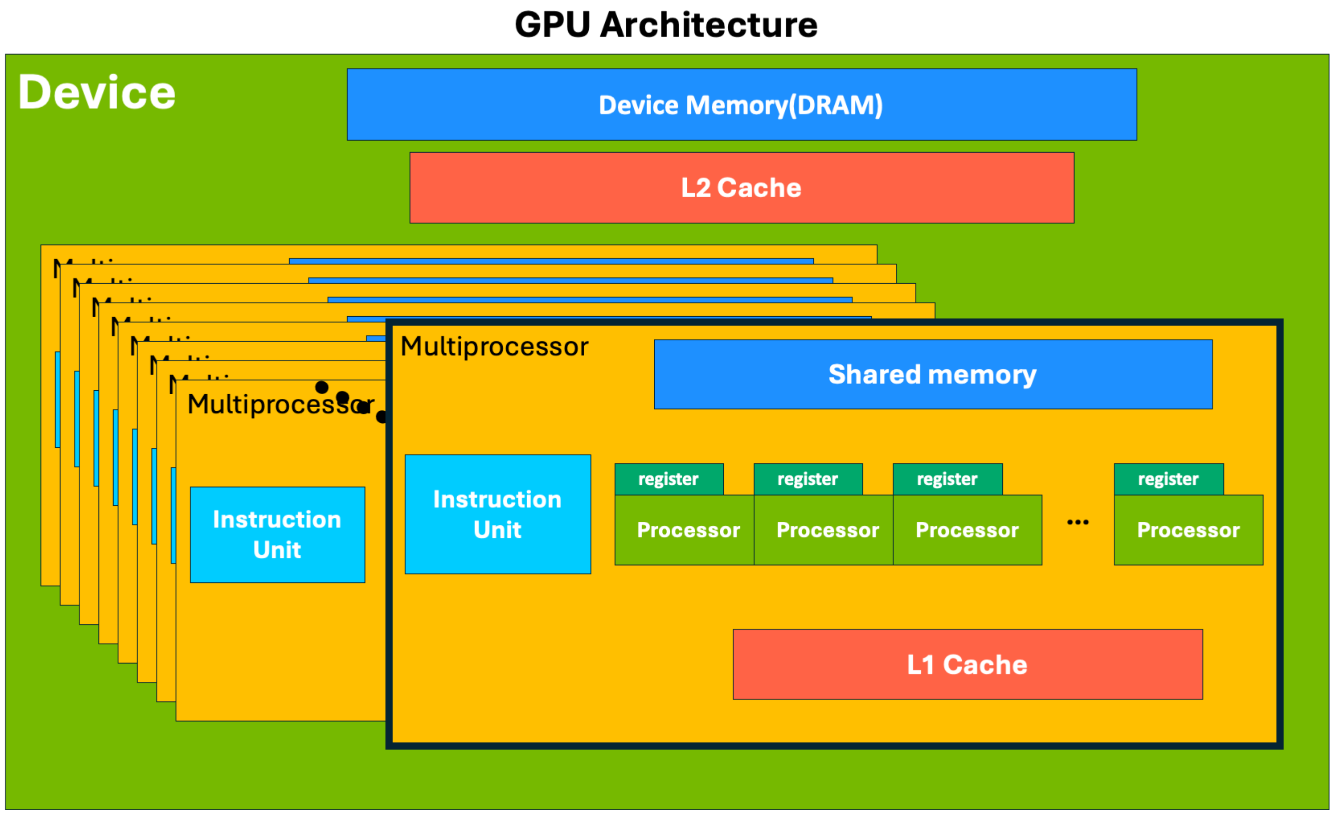

Figure 4: This figure illustrates the hierarchical structure of a GPU's architecture, highlighting the relationships between various memory and processing components. The figure is drawn based on the descriptions provided in NVIDIA's documentation hub[5].

Profiling

As we are utilizing NVIDIA GPUs for this project, we can choose from among the profiling tools that NVIDIA offers. Our preferred tool is *Nsight Compute*[6], which enables us to monitor and analyse memory usage, traffic, and communication for all the kernels in the simulation. Figure 5 shows a memory chart for the Monte Carlo step in the black tiles, it illustrates various metrics, starting with the amount of global memory requests (114.69 K) and the instruction count for accessing global memory (116.54 K). The L1 texture cache section indicates a hit rate of 24.08%, showing that roughly a quarter of all cache requests were served by the L1 texture cache, reducing the need to access the slower global memory. The L2 cache, a larger shared cache across the multiprocessors, has a hit rate of 10.45%, indicating the effectiveness of L2 in catching data requests not served by L1. The L2 cache's capacity is indicated by the current usage of 33.86 MB.

The roofline model depicted in Figure6 serves as an insightful tool for evaluating the computational efficiency of a GPU against its memory bandwidth constraints. The x-axis, displayed on a logarithmic scale, quantifies the arithmetic intensity in FLOP/byte, while the y-axis, also logarithmic, measures performance in terms of gigaFLOPS (10^9 floating-point operations per second). The characteristic 'roofline' shape represents the peak performance capabilities of the GPU, with the plateau indicating the maximum computational performance and the angled section representing the performance limitations due to memory bandwidth. There are two different roofs in the figure, the lower is for single-precision operation and the other for double-precision.

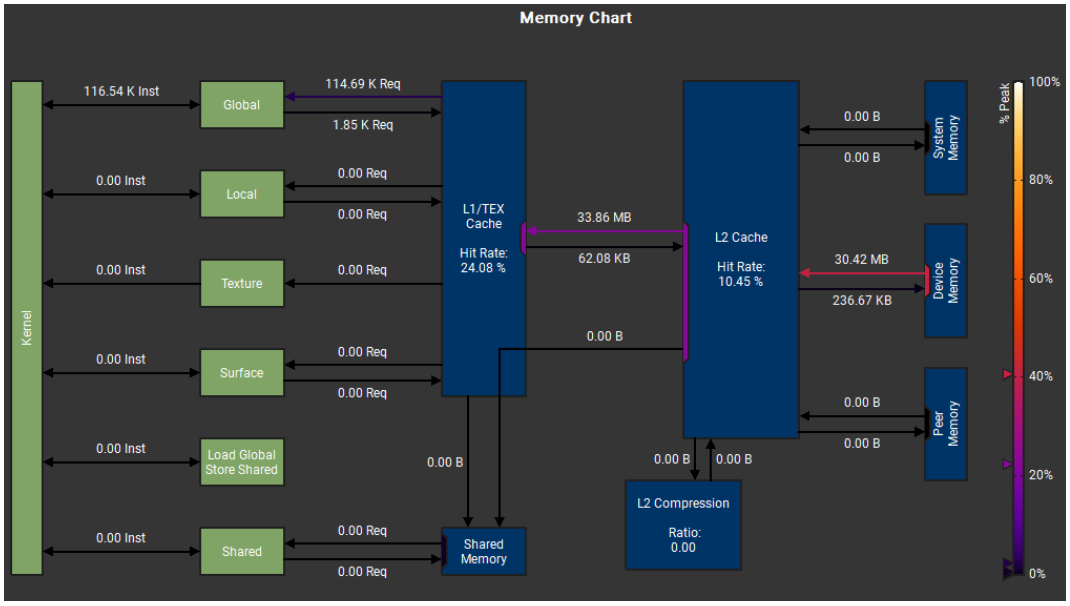

Figure 5: GPU Profiling memory chart: In this figure, the memory workload analysis for 32 threads per block and a lattice size of $(L_x, L_y) = (1024, 1024)$ is shown while executing the kernel. Ideally, we should have more memory transfers on the left side of the chart

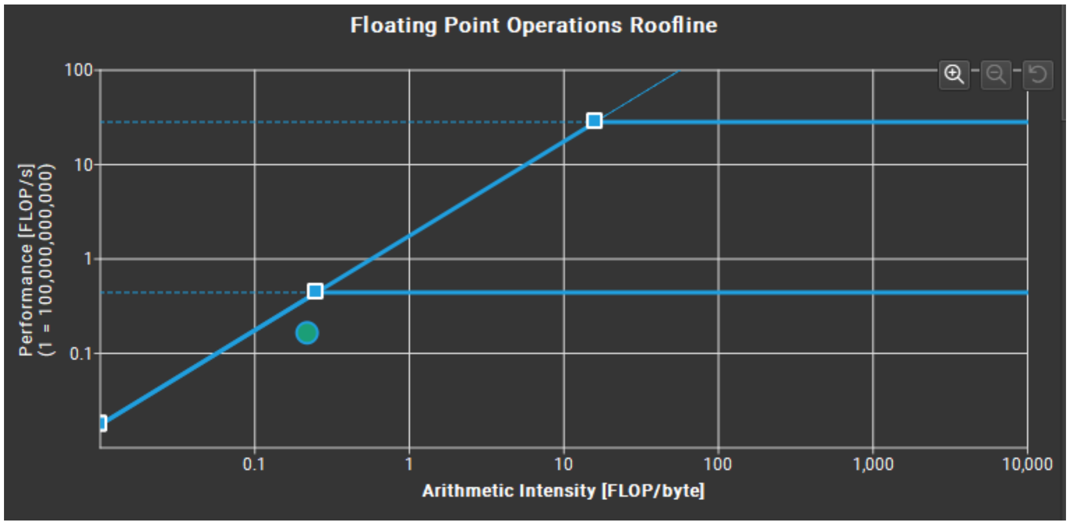

Figure 6: The roofline model of GPU performance evaluating computational efficiency and memory bandwidth constraints. We can observe two different roof lines, one for single precision and the other for double precision. The blue dot represents the performance of our benchmarking of the *monteCarlo_black* kernel for 32 threads per block and a lattice size of $(L_x, L_y) = (1024, 1024)$.

From Figure 6., we observe that we are close to the roofline, where the memory and compute-bound regions intersect. The fact that we are that close to the roofline suggests that our implementation is great. We make good, but not perfect, use of our resources and achieve really good performance. To achieve an even better performance we should make an even better use of our resources and reduce data tranfers as much as possible.

Benchmarking: Strong and Weak Scaling

We get the performance of around 54 million lattice updates per second.

Specs of the machine used

The specifications of the GPUs utilized for the project are detailed in Table 1. The benchmarking results presented were obtained using Giorgos's laptop.

	Giorgos' Laptop	ERDA
Model	NVIDIA GeForce RTX 3050 Ti Laptop GPU	NVIDIA A30 MIG 1g.6gb
Total Global Memory	4,294,443,008 bytes	6,241,124,352 bytes
Total Shared Memory per Block	49,152 bytes	49,152 bytes
Total registers per block	65,536	65,536
Warp size	32	32
Maximum Memory pitch	2,147,483,647 bytes	2,147,483,647 bytes
Maximum threads per Block	1,024	1,024
Maximum dimension 0 of Block	1,024	1,024
Maximum dimension 1 of Block	1,024	1,024
Maximum dimension 2 of Block	64	64
Maximum dimension 0 of Grid	2,147,483,647	2,147,483,647
Maximum dimension 1 of Grid	65,535	65,535
Maximum dimension 2 of Grid	65,535	65,535
Clock rate	1,035,000 kHz	1,440,000 kHz
Total constant Memory	65,536 bytes	65,536 bytes
Texture alignment	512 bytes	512 bytes
Concurrent copy and execution	Yes	Yes
Number of Multiprocessors	20	14
Kernel execution timeout	Yes	No

Table 1: Comparison of GPU specifications for benchmarking: Giorgos's laptop and the ERDA environment.

Strong and weak scaling

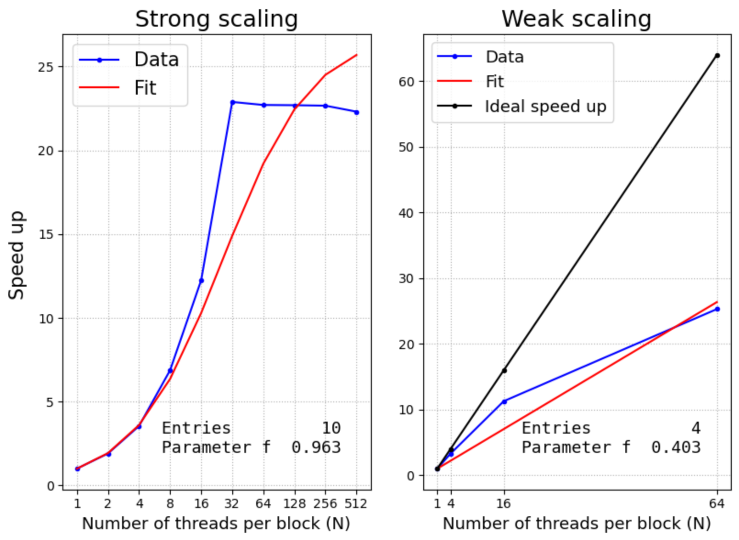

Figure 7: Speed up as a function of the number of threads per block. For the strong scaling (left), the workload $L_x \cdot L_y$ was fixed and set to $(L_x, L_y) = (1024, 1024)$, while for the weak scaling (right), the workload $L_x \cdot L_y$ (left) was proportional to the number of threads per block (N) with $L_x \cdot L_y = 16 * 16 * N$ (linear scaling). Due to the requirement of a square lattice, we sampled less data for the weak scaling plot. The number of iterations was set to 1024 for all. (Note: The number of blocks is not constant, but varying as $N_{blocks} = (\frac{L_x}{2*N_x}, \frac{L_y}{N_y}, 1)$ as can be seen in the code, where N_x and N_y denote the number of threads per block in the x and y dimensions.)

Having calculated the speedup S by dividing the elapsed time[1] of the sequential version (1 thread per block) by that of the parallelized version (N threads per block), we plotted the graphs above and fitted Ahmdahl's (left) and Gustafson's (right) law in the strong and weak scaling context respectively.

Ahmdahl's and Gustafson's laws are given by the following formulas

$$S_{Ahm.}(N) = \frac{1}{1 - f + \frac{f}{N}} \tag{4}$$

$$S_{Gus.}(N) = 1 + (N-1)\,f \tag{5}$$

Normally, N is the number of processors, but since we are dealing with GPU we do not control the number of processors. So we decided to perform this analysis by looking at the number of threads used per block. By fitting Equations 4 and 5, we were able to compute the parallel fractions f in each context, which were found to be $f = 0.963$ from Ahmdahl's law and $f = 0.403$ from Gustafson's law. This means that are code is 96.3% and 40.3% parallelizable in each context.

As far as strong scaling is concerned, for $N \leq 32$, our speedup is monotonously increasing. For $N > 32$, on the other hand, the speedup plateaus. This means that for $N = 32$ there are fewer idle cycles on the streaming multiprocessor(SM), as there are more active warps (and therefore, threads) ready to execute instructions. Consequently, the GPU's computational resources, such as execution units and registers, are optimally utilized for this N, leading to increased throughput. For $N > 32$, the speedup plateaus, meaning that adding more resources does not bring about a difference. This is expected due to the existence of the sequential part and the fact that it stays the same no matter how many resources we utilize. To substantiate this, we profiled the code using *NVIDIA's Nsight Compute* and we will restrict our discussion to the first launch of the *monteCarloStep_black* kernel. In Table2, we present a table showing that even though we add more resources, the threads need to wait even longer to execute an instruction. We chose the *Achieved Occupancy* and the *Stall Long Scoreboard* as suitable indicators of the aforementioned behavior.

[1]For the benchmarking, we had turned off all I/O's. *cudaMalloc* and *cudaMemcpy* are not included in the elapsed time as well.

Number of threads per block	Achieved Occupancy (%)	Stall Long Scoreboard (instructions per cycle)
32	30.93	18
64	62.54	39

Table 2: Table presenting the achieved occupancy (Ratio of active warps per multiprocessor to the maximum number of possible active warps) in % and the *Stall Long Scoreboard* (Stalling caused by the warps waiting for data to execute an instruction) in *instructions per cycle*.

As far as weak scaling is concerned (right subplot of Figure 7.), we notice that our implementation does not perform that well (moderate scalability), especially for $N \geq 16$, since our curve is straying more and more from the ideal one.

To have an ideal speedup, we need to keep the time of the parallelizable versions the same when increasing the number of threads per block and the workload. The main reasons why the ideal speedup cannot be reached are the overhead of the launching of the global kernels (*monteCarloStep_black* and *monteCarloStep_white*) and that of the random number generator since they are all being launched/called in every iteration. The synchronization of the threads is also not taken into account by Gustafson's law. All of the aforementioned are independent of N and therefore, do not scale linearly with N, leading to poorer performance for increasing N.

Physical Analysis

Theoretically, it can be shown that the critical temperature of the 2D Ising Model yields the following relation [3]:

$$\frac{T_c}{J} = \frac{2}{\ln(1+\sqrt{2})} \approx 2.269. \tag{6}$$

This critical behaviour is visible in plots showing magnetization per site M and susceptibility per site χ, where they are calculated as:

$$M = \frac{1}{L^2}\sum_i S_i \tag{7}$$

$$\chi = \beta\sqrt{\frac{1}{L^2}\sum_i (S_i - \langle S_i \rangle)^2}, \tag{8}$$

where L^2 is the System size. These equations can be derived as a result of the canonical ensemble [2]. Figures 8 and 9 show the model output of the *CUDA* version of the simulation.

Here, we plot the **absolute value of the magnetization** with periodic boundary conditions applied, where we have no preferred spin direction. In the **low-temperature region**, all spins align in the same direction, yielding a magnetization per site of 1.0. **With higher temperatures**, thermal fluctuations increase, causing the magnetization to decrease. At a **critical point**, a peak occurs (dashed). This would be a **singularity in the susceptibility** for infinitely large lattice sizes, as it is formally the derivative of the magnetization where the magnetization has an infinite slope at the inflexion point. Thus, the susceptibility exhibits a discontinuity, and as a response function (the second derivative of a thermodynamic potential), it indicates a **second-order phase transition**.

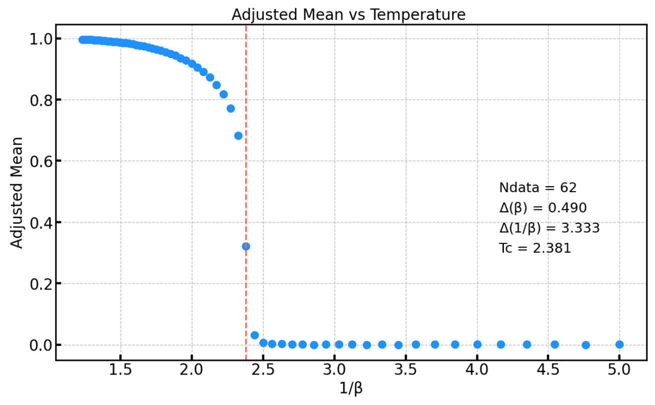

Figure 8: Absolute value of the magnetization per site plotted against the temperature $\frac{1}{\beta}$ for a system of size 1024×1024. The y-axis is dimensionless.

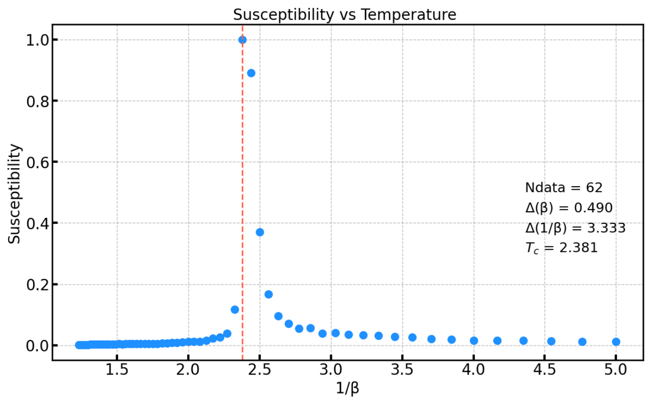

Figure 9: Susceptibility per site (normalized on the peak value) plotted against the temperature $\frac{1}{\beta}$ for a system of size 1024×1024. The y-axis is dimensionless.

The dashed line in the plots indicates a rough **estimate of the critical temperature**, which here is $T_c = 2.381$. This approach to estimating the critical temperature is just a quick test. Usually, so-called **Binder cumulants** are used to compute the critical point very efficiently. The thermodynamic quantities like magnetisation and susceptibility are **limited in usefulness** for this purpose [2]. All in all, the plots and results agree with the theoretical expectations.

Future Work

Using Shared Memory

One huge improvement would have been to use shared memory. Shared memory is located on-chip, which makes it much faster to access compared to global memory, which resides off-chip and is shared among all threads in the grid. To incorporate shared memory into our implementation, we would need to make sure that every block has the required data to perform all calculations and for that, we needed to restructure the algorithm. Using another memory shown in figure 5 could give us some improvements but nothing very noticeable compared with the shared memory.

Summary and Conclusion

In conclusion, the utilization of NVIDIA GPUs with CUDA in implementing the Checkerboard Algorithm for the 2D Ising Model has yielded promising results. The primary objective of this report was to develop an efficient CUDA code capable of reproducing essential properties of the 2D Ising Model accurately. Through the checkerboard algorithm design and optimization, we have successfully achieved this goal. Yet, there are possibilities to improve the efficiency of the code further and dive more into the physical analysis of the phase transition.

Central to our approach was the adoption of the *Checkerboard Algorithm*, a clever strategy that maximizes parallel computing capabilities by partitioning the lattice into black and white subgroups. This algorithmic technique allowed us to exploit the high potential of GPU architectures, significantly accelerating the simulation process while maintaining fidelity to the underlying physical model.

The culmination of our efforts is a highly efficient CUDA C code capable of faithfully reproducing key properties of the 2D Ising Model, including magnetization, susceptibility, and critical behaviour.

References

[1] Kherzie Andal. The checkerboard metropolis algorithm explained. `https://medium.com/@kherzieandal/the-checkerboard-metropolis-algorithm-explained-1ecdd301d17d`, 2023. Accessed from Kherzie Andal's blog: `https://kherzieandal.github.io/blog`.

[2] K. Binder and D.W. Heermann. *Monte Carlo Simulation in Statistical Physics*. Springer, 3 edition, 1997.

[3] Marek Biskup et al. *Methods of Contemporary Mathematical Statistical Physics*. Springer, 2009.

[4] N. Metropolis, A. Rosenbluth, M. Rosenbluth, M. Teller, and E. Teller. Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[5] NVIDIA. Gpu performance background user's guide. NVIDIA Deep Learning AI, 2024. Accessed: 2024-03-23.

[6] NVIDIA. NVIDIA Nsight Compute. `https://developer.nvidia.com/nsight-compute`, 2024. Accessed: 2024-03-25.

[7] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda. `https://developer.nvidia.com/cuda-toolkit`. Accessed: [Insert access date here].

Codes

Cuda Version (Metropolis Checkerboard Algorithm)

Code Listing 1: Checkerboard Algorithm

```
#include <stdio.h>

#include <chrono>
#include <cmath>
#include <cstddef>
#include <cstdlib>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <random>
#include <sstream>
#include <vector>
// #include "omp.h"
#include <curand.h>
#include <curand_kernel.h>
#include <math.h>

#include <fstream>
#include <sstream>
#include <type_traits>
#include <vector>

#include "cuda_runtime.h"
#include "device_launch_parameters.h"

static void HandleError(cudaError_t err,
                        const char *file,
                        int line) {
    if (err != cudaSuccess) {
        printf("%s in %s at line %d\n", cudaGetErrorString(err),
               file, line);
        exit(EXIT_FAILURE);
    }
}
#define HANDLE_ERROR(err) (HandleError(err, __FILE__, __LINE__))

/*
specifically designed for lattice_white and lattice_black
*/

const int LX = 1024;  // Size of the lattice
const int LY = 1024;
const int L = 1024;

const int ITERATIONS = 3 * 1024;
// const int MCSteps = 10000;  // Number of Monte Carlo steps
int savestep = 3 * 1024 + 1;
const double J = 1.0;  // Coupling constant

__device__ __constant__ int d_J = 1.0f;
__device__ __constant__ int d_LX = LX;
__device__ __constant__ int d_LY = LY;
__device__ __constant__ int d_SubL = LX / 2;

__device__ float d_beta = 0.2f;

/* host side arrays*/
int *h_l;
int *h_lb;
int *h_lw;

/* device side arrays*/
int *d_l;
int *d_lb;
int *d_lw;

curandState *devStates_b, *devStates_w;

void initialize(int *h_l);
void matrix_decomposition(int *lattice, int *even_elements, int *odd_elements, int
    size);

__global__ void monteCarloStep_white(int *lattice_black, int *lattice_white,
    curandState *devStates_w);
__global__ void monteCarloStep_black(int *lattice_black, int *lattice_white,
    curandState *devStates_b);
__global__ void setup_kernel_b(curandState *state_b, unsigned long long seed1);
__global__ void setup_kernel_w(curandState *state_w, unsigned long long seed2);
__global__ void set_d_beta(float beta);

__global__ void set_d_beta(float beta) {
    // Ensure only one thread (the first thread) performs the action
    if (threadIdx.x == 0 && blockIdx.x == 0) {
        d_beta = beta;
    }
}

__global__ void unique(int *input) {
    int threadid = threadIdx.x;
    printf("threeadIdx : %d, value : %d \n", threadid, input[threadid]);
}

int magnetization(int *d_lb, int *d_lw) {
    int sum = 0;
    for (int i = 0; i < L * L / 2; i++) {
        sum += (d_lw[i] + d_lb[i]);
    }
    return sum;
}

int main() {
    size_t size = LX * LY;

    h_l = (int *)malloc(size * sizeof(int));
    h_lb = (int *)malloc(size / 2 * sizeof(int));
    h_lw = (int *)malloc(size / 2 * sizeof(int));

    HANDLE_ERROR(cudaMalloc(&d_l, size * sizeof(int)));
    HANDLE_ERROR(cudaMalloc(&d_lb, size * sizeof(int) / 2));
    HANDLE_ERROR(cudaMalloc(&d_lw, size * sizeof(int) / 2));

    // −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
    // Allocate memory for device states
    cudaMalloc((void **)&devStates_b, size / 2 * sizeof(curandState));
    cudaMalloc((void **)&devStates_w, size / 2 * sizeof(curandState));
    // −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

    initialize(h_l);
    matrix_decomposition(h_l, h_lb, h_lw, size);

    HANDLE_ERROR(cudaMemcpy(d_l, h_l, size * sizeof(int), cudaMemcpyHostToDevice));
    HANDLE_ERROR(cudaMemcpy(d_lb, h_lb, size * sizeof(int) / 2,
       cudaMemcpyHostToDevice));
    HANDLE_ERROR(cudaMemcpy(d_lw, h_lw, size * sizeof(int) / 2,
       cudaMemcpyHostToDevice));

    unsigned long long seed1 = 1234;
    unsigned long long seed2 = 1011;

    dim3 block(128, 2, 1);

    dim3 grid(LX / 2 / block.x, LY / block.y, 1);
    // std::ostringstream filename;
    // filename << "mag_results/magnetization" << std::fixed << std::setprecision(0)
        << (beta * 100) << "txt";

    // std::ofstream outfile(filename.str(), std::ios::app);
    // if (!outfile.is_open()) {
    //     std::cerr << "Unable to open file for writing." << std::endl;
    //     return EXIT_FAILURE;
    // }

    auto tstart = std::chrono::high_resolution_clock::now();

    for (int t = 1; t <= ITERATIONS; t++) {
        // −−−−−−−−−−−−−−−−−−−−
        // Setup kernel for initializing device states
        setup_kernel_b<<<grid, block>>>(devStates_b, seed1);
        setup_kernel_w<<<grid, block>>>(devStates_w, seed2);
        seed1++;
        seed2++;
        // −−−−−−−−−−−−−−−−−−−−

        HANDLE_ERROR(cudaDeviceSynchronize());
        monteCarloStep_black<<<grid, block>>>(d_lb, d_lw, devStates_b);
        HANDLE_ERROR(cudaDeviceSynchronize());
        monteCarloStep_white<<<grid, block>>>(d_lb, d_lw, devStates_w);
        HANDLE_ERROR(cudaDeviceSynchronize());

        // Inside the if block where savestep condition is checked
    }

    auto tend = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(tend −
       tstart).count();
    float MLUPS = (ITERATIONS * static_cast<double>(LX * LY) / (duration /
        1000000.0)) / 1000000.0;

    printf("performance : %f MLUPS \n", MLUPS);

    // Free host memory
    free(h_l);
    free(h_lb);
    free(h_lw);

    // Free device memory
    cudaFree(d_l);
    cudaFree(d_lb);
    cudaFree(d_lw);

    // Free device memory for device states
    cudaFree(devStates_b);
    cudaFree(devStates_w);
    std::cout << "Elapsed time:" << std::setw(9) << std::setprecision(4)
              << (tend - tstart).count() * 1e-9 << "\n";

    return 0;
}

void initialize(int *h_l) {
    for (int i = 0; i < LX; i++) {
        for (int j = 0; j < LY; j++) {
            h_l[i + j * LX] = 1;
        }
    }
}

__global__ void monteCarloStep_black(int *lattice_black, int *lattice_white,
    curandState *devStates_b) {
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int ID = x + y * d_SubL;

    int xp = (x + 1 + d_SubL) % d_SubL;

    int yp = (y + 1 + d_SubL) % d_SubL;
    int ym = (y - 1 + d_SubL) % d_SubL;

    curandState localState = devStates_b[ID];

    int deltaE = 2 * lattice_black[ID] * (lattice_white[ID] + lattice_white[xp + y *
        d_SubL] + lattice_white[x + yp * d_SubL] + lattice_white[x + ym * d_SubL]);

    float randnum = curand_uniform(&localState);

    if (deltaE <= 0 || randnum < expf(-d_beta * deltaE)) {
        lattice_black[ID] *= -1;  // Accept the spin flip
    }
}
__global__ void monteCarloStep_white(int *lattice_black, int *lattice_white,
    curandState *state_w) {
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int ID = x + y * d_SubL;

    // int xp = (x + 1 + d_SubL) % d_SubL;
    int xm = (x - 1 + d_SubL) % d_SubL;

    int yp = (y + 1 + d_SubL) % d_SubL;
    int ym = (y - 1 + d_SubL) % d_SubL;

    curandState localState = state_w[ID];
    float randnum = curand_uniform(&localState);

    int deltaE = 2 * d_J * lattice_white[ID] * (lattice_black[ID] + lattice_black[xm
        + y * d_SubL] + lattice_black[x + yp * d_SubL] + lattice_black[x + ym *
       d_SubL]);

    if (deltaE <= 0 || randnum < expf(-d_beta * deltaE)) {
        lattice_white[ID] *= -1;  // Accept the spin flip
    }
}

void matrix_decomposition(int *h_l, int *h_lb, int *h_lw, int size) {
    int even_count = 0, odd_count = 0;
    for (int i = 0; i < L; ++i) {
        for (int j = 0; j < L; ++j) {
            if ((i + j) % 2 == 0) {
                h_lb[even_count++] = h_l[j + i * L];
            } else {
                h_lw[odd_count++] = h_l[j + i * L];
            }
        }
    }
}

// --------------------------------------------------------
__global__ void setup_kernel_b(curandState *state_b, unsigned long long seed1) {
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int ID = x + y * d_SubL;

    curand_init(seed1, ID, 0, &state_b[ID]);
}

__global__ void setup_kernel_w(curandState *state_w, unsigned long long seed2) {
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int ID = x + y * d_SubL;

    curand_init(seed2, ID, 0, &state_w[ID]);
}
```