UNIVERSITÀ DEGLI STUDI DI PADOVA

2D Ising model CUDA implementation

Modern Computing for Physics 2024-2025

Giancarlo Saran Gattorno

Overview

1. Problem outline
2. Serial algorithm
3. Parallel algorithm
4. Performance details

Ising Model

- Born to explain for **phase transitions** in ferromagnets
- Surprisingly general for large systems with short range interactions and binary microscopic states
- Has applications in **CFD, neuroscience, artificial intelligence, population modeling**

Hamiltonian $$H = -J \sum_{\langle x,y \rangle} \sigma_x \sigma_y - h \sum_x \sigma_x$$

Ising Model

- Born to explain for **phase transitions** in ferromagnets
- Surprisingly general for large systems with short range interactions and binary microscopic states
- Has applications in **CFD, neuroscience, artificial intelligence, population modeling**

Hamiltonian $$H = -J \sum_{\langle x,y \rangle} \sigma_x \sigma_y - h \sum_x \sigma_x$$

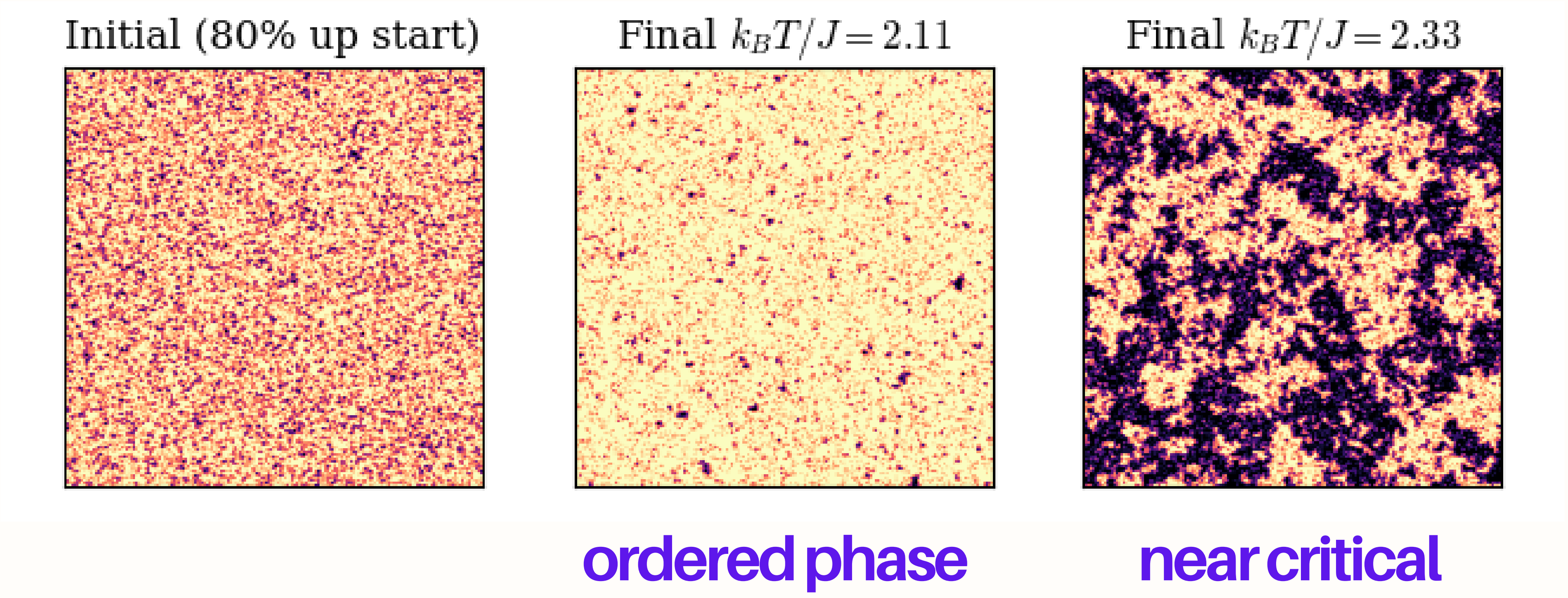
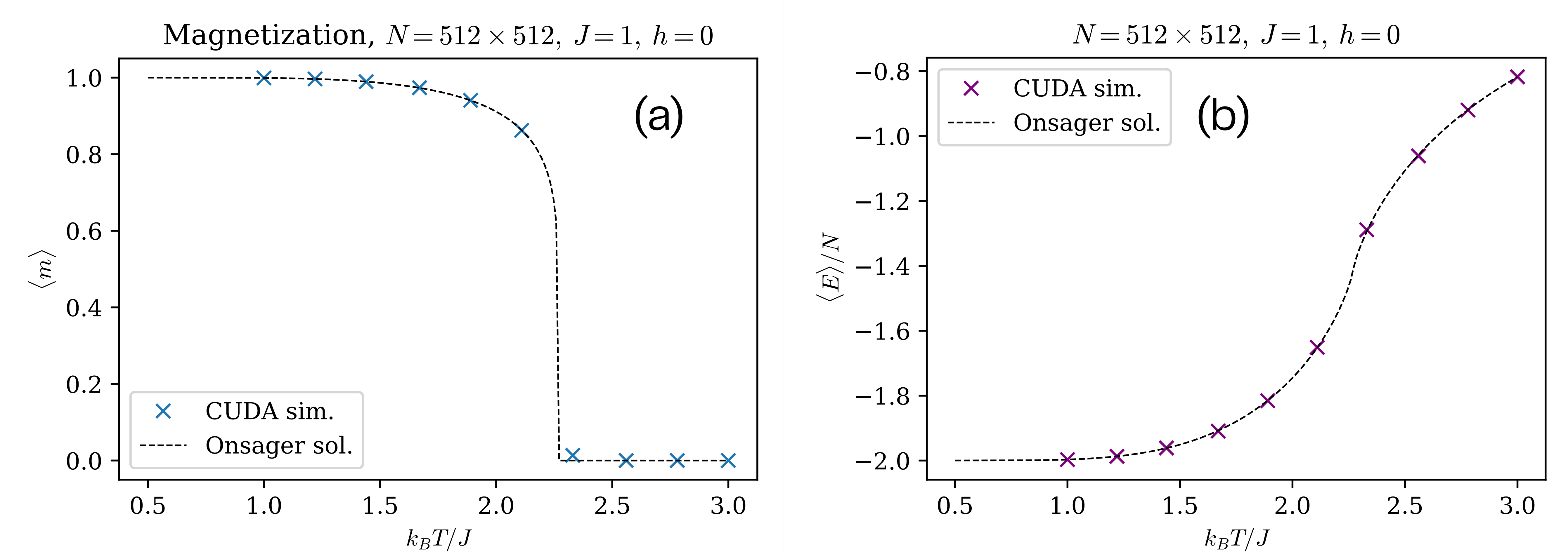


Figure: 512x512 lattices generated with the parallel CUDA Ising algorithm

Ising Model



Onsager's Solution (1944)

a) magnetization

$$m = \left[1 - \sinh^{-4}(2\beta)\right]^{1/8}$$

b) energy

$$E = -\coth(2\beta)\left[1 + \frac{2}{\pi}\left(2\tanh^2(2\beta) - 1\right)\int_0^{\pi/2} \frac{1}{\sqrt{1 - 4k(1+k)^{-2}\sin^2\theta}}\right]$$

Framework

- Code written in **CUDA-C**
- Built and tested on a **Jetson Nano** board, with nvcc v10.2
- 2GB VRAM
- 472 GFLOPS
- 25.6 GB/s bandwidth
- 128 CUDA cores, Maxwell architecture

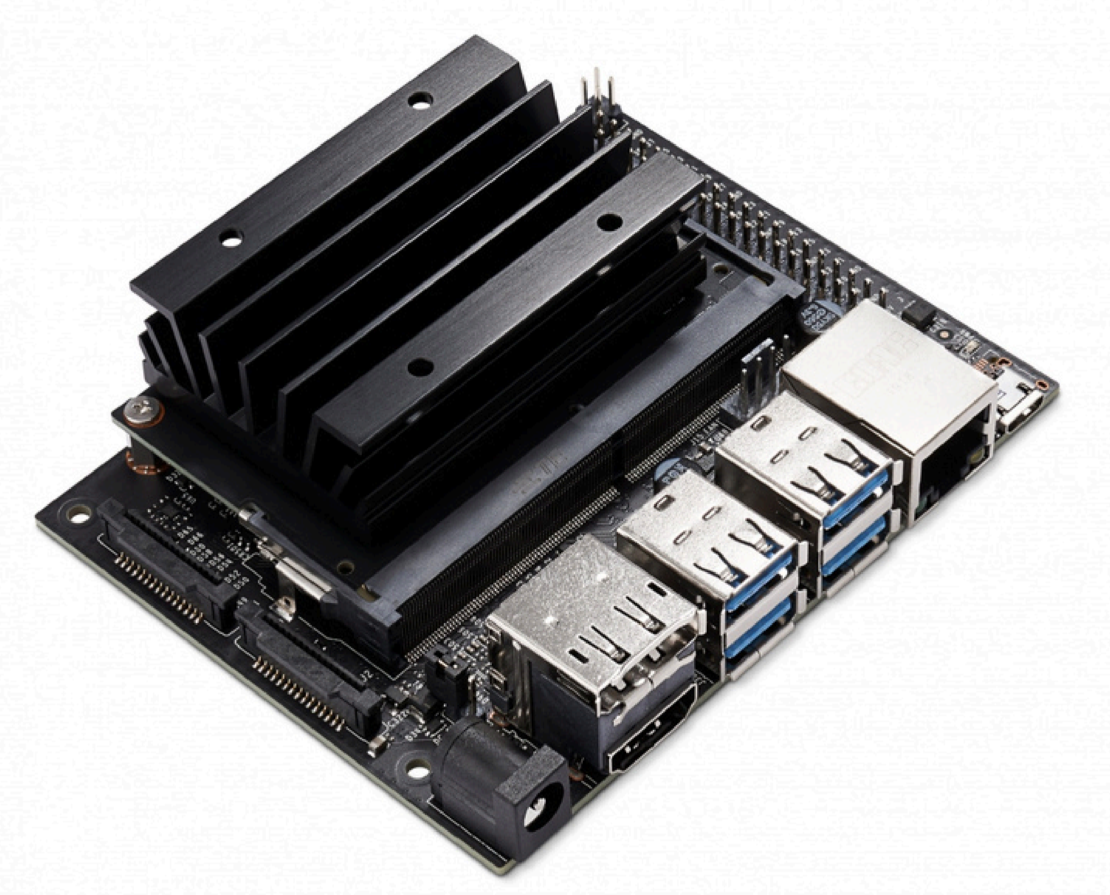


Figure: Jetson Nano board

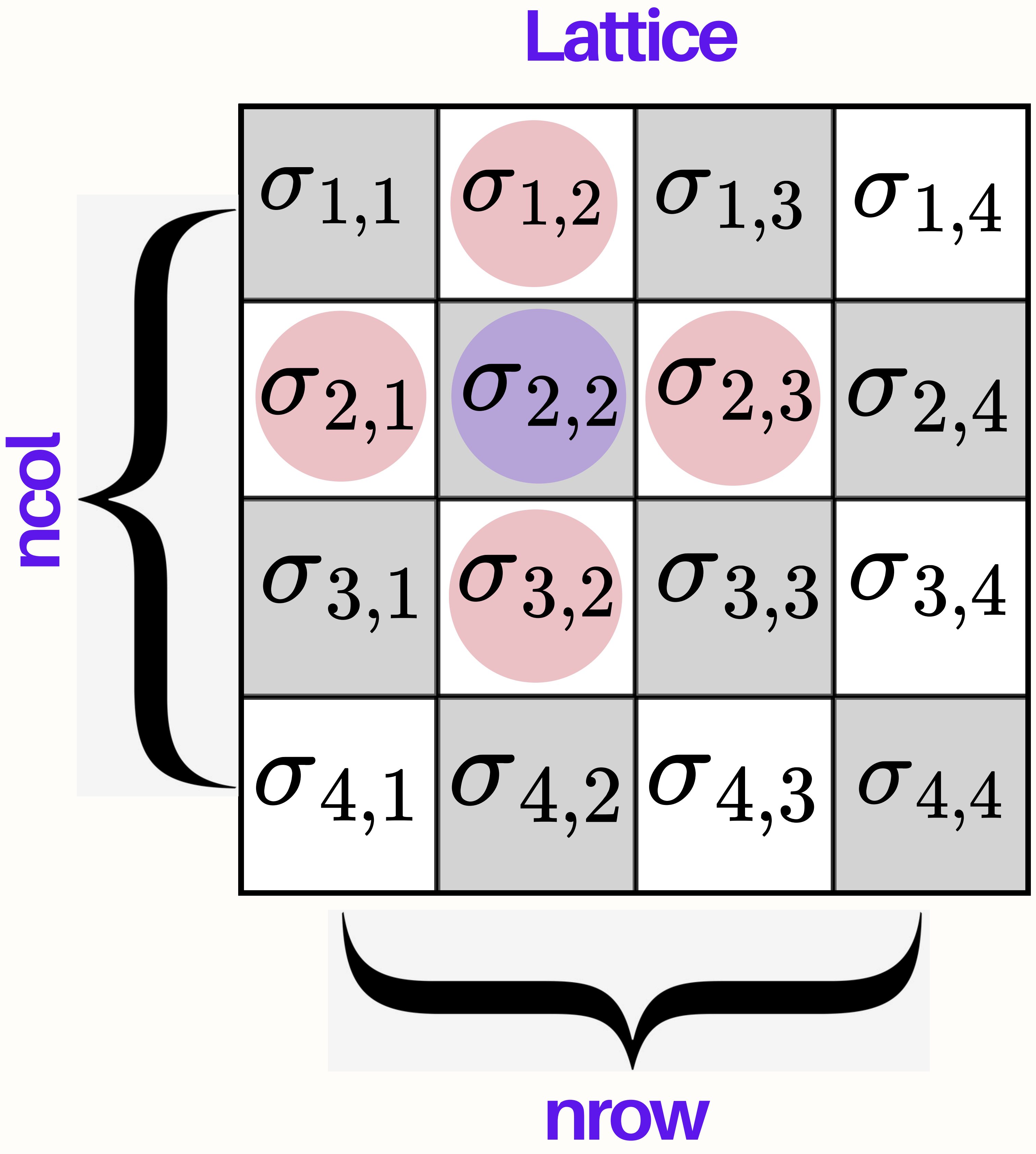
Serial algorithm

Algorithm 1 Serial Ising MCMC

Require: Spin lattice $\boldsymbol{\sigma}$, temperature T, coupling J, field h, number of steps N

for $t = 1$ **to** N **do**
    for $x = 1$ to L_x **do**
        for $y = 1$ to L_y **do**
            $\sigma'_{x,y} = -\sigma_{x,y}$
            Compute $\Delta E = E(\boldsymbol{\sigma}') - E(\boldsymbol{\sigma})$
            Compute Metropolis ratio $r = \min(1, e^{-\beta \Delta E})$
            Accept move with probability r
        end for
    end forreturn $E(\boldsymbol{\sigma}), m(\boldsymbol{\sigma})$
end for

Checkerboard algorithm



$$\Delta E_{2,2} = 2J\sigma_{2,2}(\sigma_{1,2} + \sigma_{2,1} + \sigma_{2,3} + \sigma_{3,2}) + 2h\sigma_{2,2}$$

Checkerboard algorithm

Lattice

$\sigma_{1,1}$	$\sigma_{1,2}$	$\sigma_{1,3}$	$\sigma_{1,4}$
$\sigma_{2,1}$	$\sigma_{2,2}$	$\sigma_{2,3}$	$\sigma_{2,4}$
$\sigma_{3,1}$	$\sigma_{3,2}$	$\sigma_{3,3}$	$\sigma_{3,4}$
$\sigma_{4,1}$	$\sigma_{4,2}$	$\sigma_{4,3}$	$\sigma_{4,4}$

ncol

nrow

$$\Delta E_{2,2} = 2J\sigma_{2,2}(\sigma_{1,2} + \sigma_{2,1} + \sigma_{2,3} + \sigma_{3,2}) + 2h\sigma_{2,2}$$

Memory representation

signed char* **d_black**

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

signed char* **d_white**

Checkerboard algorithm

```
void MCMC_step(signed char *d_black, signed char *d_white,
               int nrow, int ncol, float beta, float J, float h, bool use_lut) {

    int n_elements = nrow * ncol/2;
    int num_blocks = (n_elements + BLOCK_SIZE - 1) / BLOCK_SIZE;

    // Update black sites
    color_update<<<num_blocks, BLOCK_SIZE>>>(
        d_black, d_white, d_black_states, nrow, ncol/2, beta, J, h, use_lut, true);
    CUDA_CHECK(cudaDeviceSynchronize());

    // Update white sites
    color_update<<<num_blocks, BLOCK_SIZE>>>(
        d_white, d_black, d_white_states, nrow, ncol/2, beta, J, h, use_lut, false);
    CUDA_CHECK(cudaDeviceSynchronize());
}
```

Independent

Checkerboard algorithm

```
__global__ void color_update(signed char *color, signed char *op_color,
                             curandState *states, int nrow, int ncol,
                             float beta, float J, float h, bool use_lut, bool black_color) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    const int row = idx / ncol; //lattice row
    const int col = idx % ncol; //lattice column
    if (row >= nrow || col >= ncol) return; //avoid overflow

    signed char s_neigh = get_neighbor_spins(op_color, row, col, nrow, ncol, black_color); //S_nn
    signed char s_ij = color[idx];
    float deltaE;
    float metropolis_ratio;
    float xi = curand_uniform(&states[idx]);
    // Metropolis acceptance criterion
    // [...] -> metropolis_ratio
    if (deltaE <= 0.0f || xi < metropolis_ratio){
        //Accepted MC move, flip the spin
        color[idx] = -s_ij;
    }
}
```

Lookup Table

- Common optimization in single threaded Ising
- When computing Metropolis ratio there's an **exponential**, but the possible exponents are limited
- Precompute **table** of values to store in memory
- **Reduces FLO** per Monte Carlo step but **increases memory** reads

Lookup Table

$$LUT = e^{-2\beta(J\sigma \cdot S_{nn} + h\sigma)}$$

__device__ float* d_lut

$$\sigma \in \{-1, +1\} \left\{ \begin{array}{|c|c|c|c|c|} \hline e^{\beta(8J+h)} & e^{\beta(4J+h)} & e^{\beta h} & e^{\beta(-4J+h)} & e^{\beta(-8J+h)} \\ \hline e^{\beta(8J-h)} & e^{\beta(4J-h)} & e^{-\beta h} & e^{\beta(-4J-h)} & e^{\beta(-8J-h)} \\ \hline \end{array} \right.$$

$$\sigma \cdot S_{nn} \in \{-4, -2, 0, +2, +4\}$$

Random numbers

curandState uses XORWOW by default, all states have a global seed and an independent "channel" for each thread

```
__global__ void init_rng(curandState *states, unsigned long seed, int n, bool black_color) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned long color_seed = seed + (black_color ? 123456ULL : 0ULL); // Different seed for two colors
    if (idx < n) {
        curand_init(color_seed, idx, 0, &states[idx]);
    }
}
```

- Every thread is assigned an **independent RNG**, with a state that is continuously updated

Random numbers

curandState uses XORWOW by default, all states have a global seed and an independent "channel" for each thread

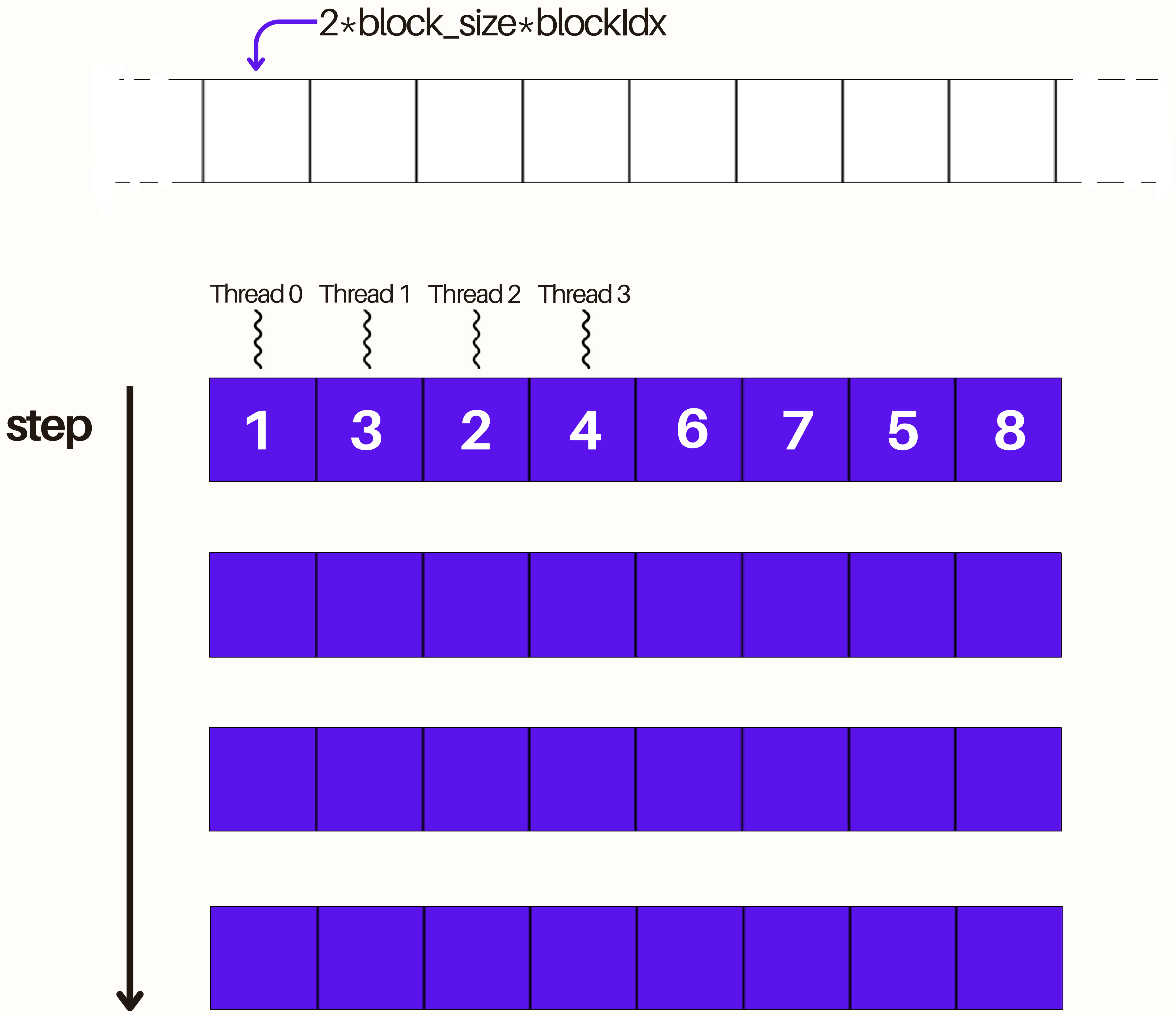
```
__global__ void init_rng(curandState *states, unsigned long seed, int n, bool black_color) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned long color_seed = seed + (black_color ? 123456ULL : 0ULL); // Different seed for two colors
    if (idx < n) {
        curand_init(color_seed, idx, 0, &states[idx]);
    }
}
```

- Every thread is assigned an **independent RNG**, with a state that is continuously updated

```
__global__ void init_lattice(signed char *color, curandState *globalStates, float up_prob, int nrow, int ncol){
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int row = idx / ncol;
    int col = idx % ncol;
    if (row >= nrow || col >= ncol) return;

    float xi = curand_uniform(&globalStates[idx]);
    color[idx] = (xi < up_prob) ? 1 : -1;
}
```

Parallel Reduction



__device__ signed char* **d_white**

$$F(\sigma_{idx}^{(color)}) =$$

magnetization

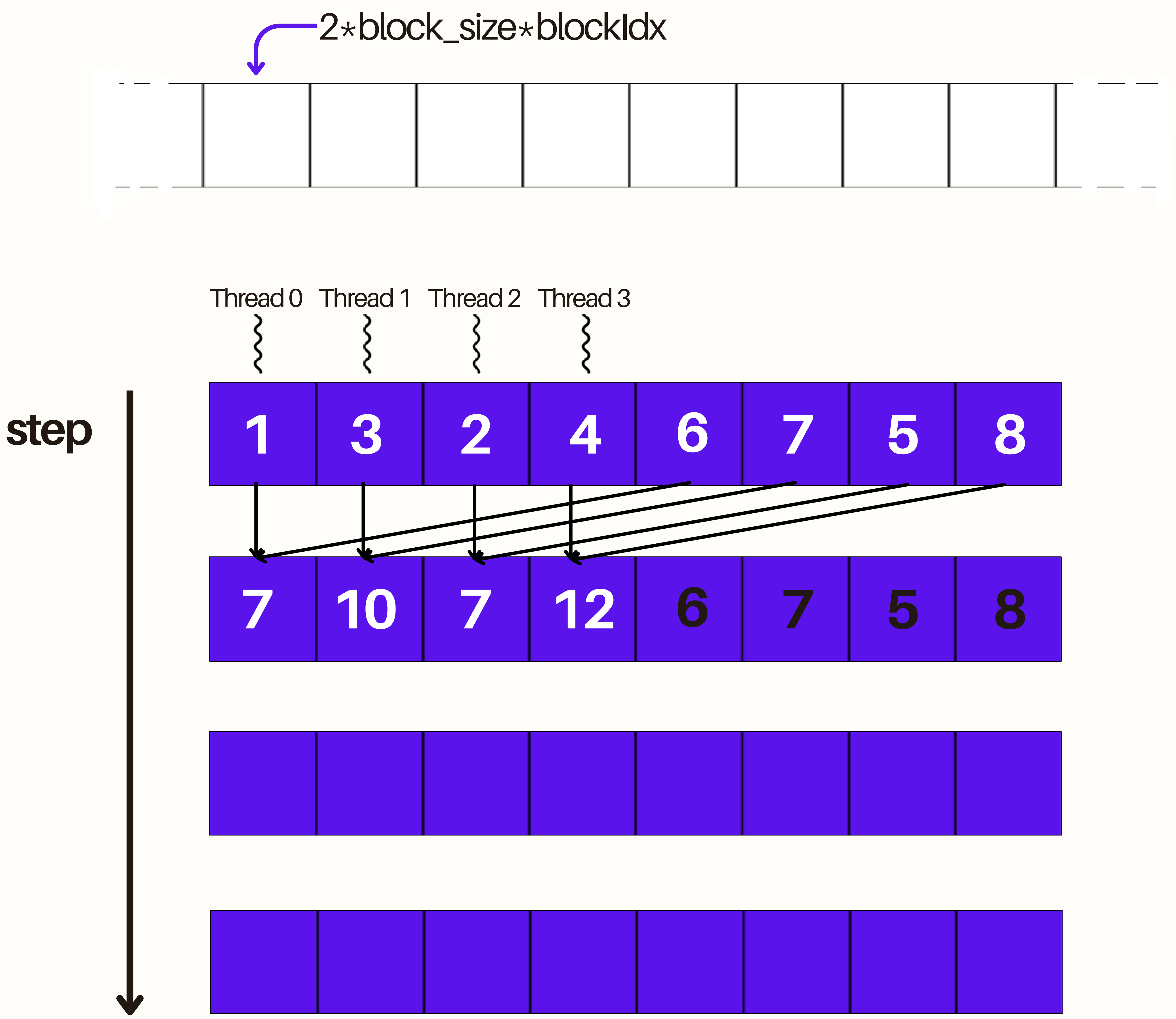
$$\sigma_{idx}^{(color)}$$

energy

$$\sigma_{idx}^{(color)} \cdot S_{nn,idx}^{(op.color)}$$

__shared__ float* **shared_F**

Parallel Reduction

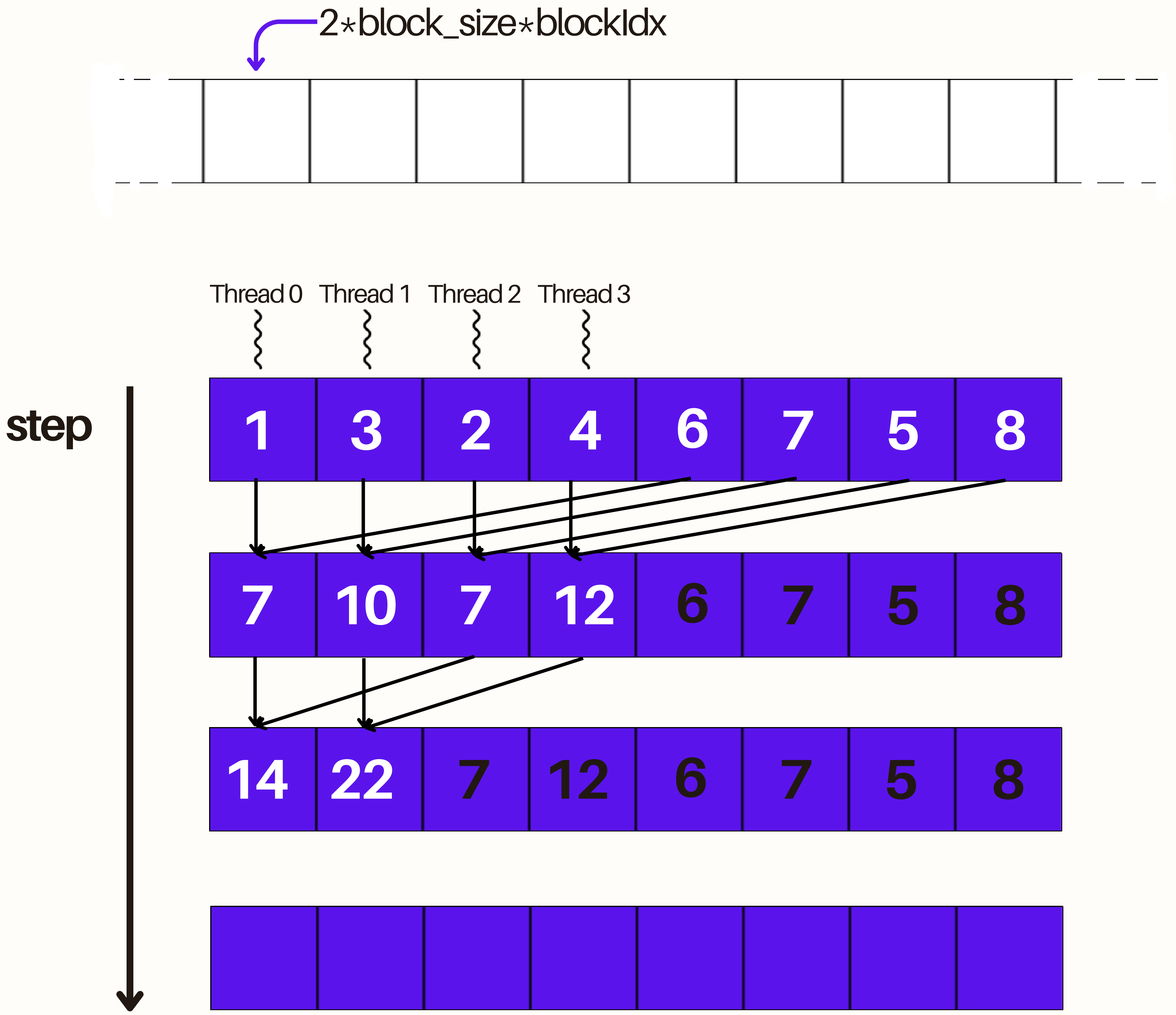



```
__device__ signed char* d_white
```

$$F(\sigma_{idx}^{(color)}) = \begin{cases} \sigma_{idx}^{(color)} & \text{magnetization} \\ \sigma_{idx}^{(color)} \cdot S_{nn,idx}^{(op.color)} & \text{energy} \end{cases}$$

```
__shared__ float* shared_F
```

Parallel Reduction



__device__ signed char* **d_white**

$$F(\sigma_{idx}^{(color)}) =$$

magnetization

$$\sigma_{idx}^{(color)}$$

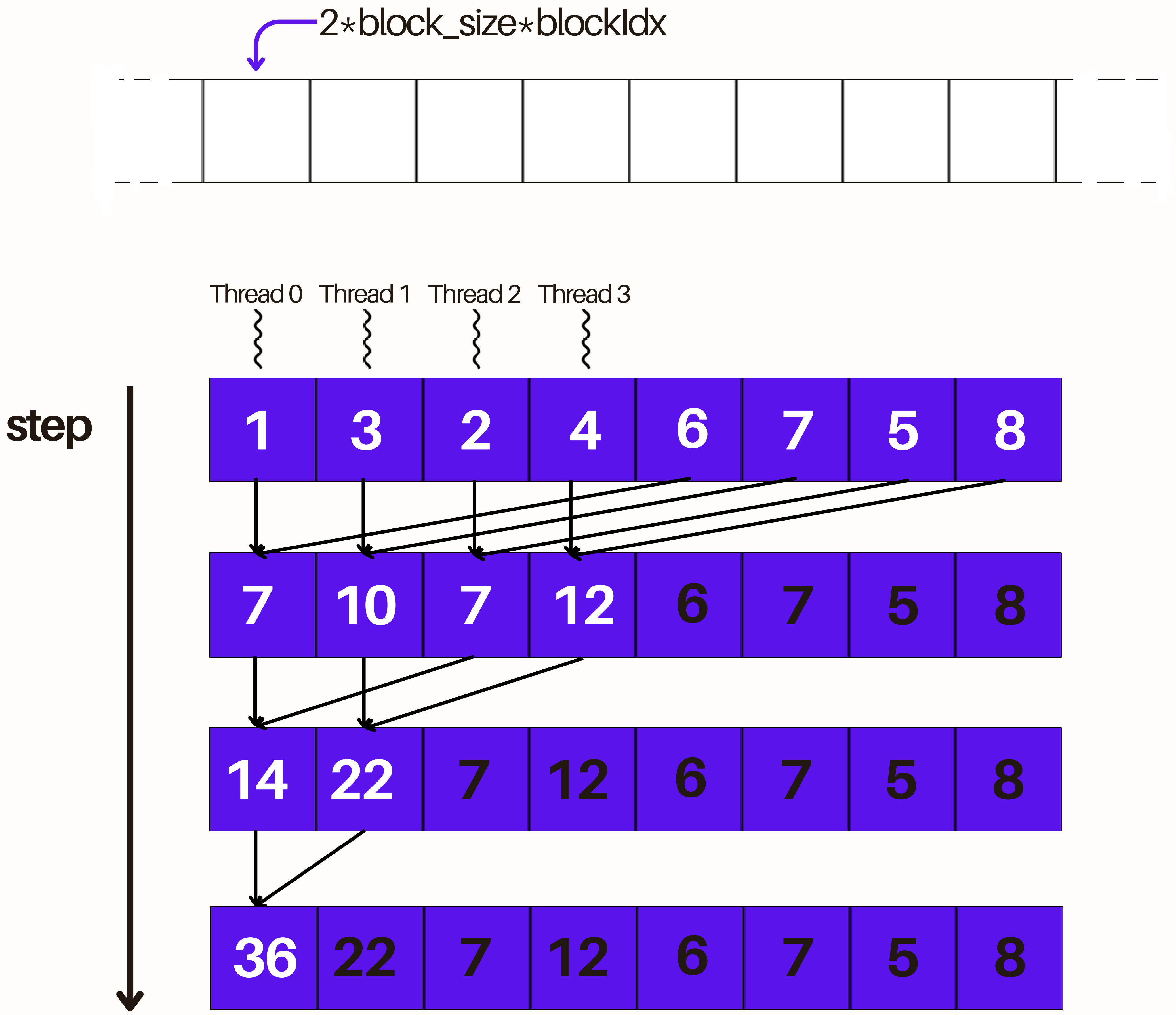
energy

$$\sigma_{idx}^{(color)} \cdot S_{nn,idx}^{(op.color)}$$

__shared__ float*

shared_F

Parallel Reduction



__device__ signed char* **d_white**

$$F(\sigma_{idx}^{(color)}) =$$

magnetization $\sigma_{idx}^{(color)}$

energy $\sigma_{idx}^{(color)} \cdot S_{nn,idx}^{(op.color)}$

__shared__ float* **shared_F**

Parallel Reduction

```
__global__ void block_sum(signed char *color, signed char *op_color,
                          float *blockF, int nrow, int ncol, bool black_color, bool energy_func) {
    int bdim = blockDim.x;
    int bx = blockIdx.x;
    int tx = threadIdx.x;
    int start_idx = 2 * bdim * bx;
    __shared__ float sharedF[2 * BLOCK_SIZE]; //We process 2*threads_per_block indices of the array
    // We have to initialize to zero so that excess addresses don't influence the reduction
    sharedF[tx] = 0.0f;
    sharedF[tx + bdim] = 0.0f;
    // [...] Filling sharedF with initial data
    __syncthreads(); // Sync the accesses
    // Loop summation with halving strides
    for (int stride = bdim; stride > 0; stride >>= 1) {
        if (tx < stride) {
            sharedF[tx] += sharedF[tx + stride];
        }
        __syncthreads(); //Wait for reduction step to end before accessing other addresses
    }
    // Write block result to global memory
    if (tx == 0) {
        blockF[blockIdx.x] = sharedF[0];
    }
}
```

Parallel Reduction

- In order to finalize observable calculation a **reduction of block results** is performed on the host

```
float complete_reduction(float *d_blockResults, int num_blocks) {
    //Move to host
    float *h_blockResults = (float*)malloc(num_blocks * sizeof(float));
    CUDA_CHECK(cudaMemcpy(h_blockResults, d_blockResults,
                          num_blocks * sizeof(float), cudaMemcpyDeviceToHost));

    //Final reduction
    float total = 0.0f;
    for (int i = 0; i < num_blocks; i++) {
        total += h_blockResults[i];
    }

    free(h_blockResults);
    return total;
}
```

Size scaling

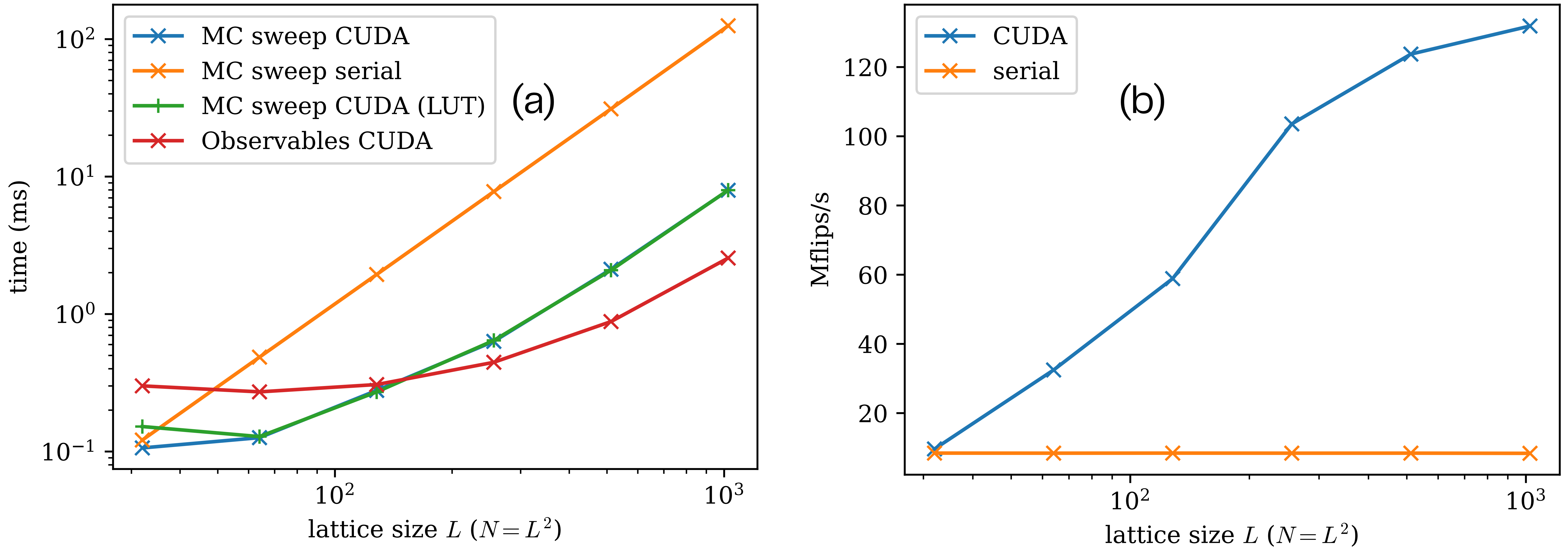


Figure: a) Compute time vs lattice size of various tasks. The $\mathcal{O}(N^2)$ nature of the MC step is evident in the serial code. For parallel reductions it should be $\mathcal{O}(\text{block\_size} \cdot \log N)$

b) Number of spin flips per second. Since this makes the number of operations independent of scale we see more clearly the effect of Warp scheduling **hiding the latency** of memory access, as parallel performance **improves with lattice size**

Size scaling

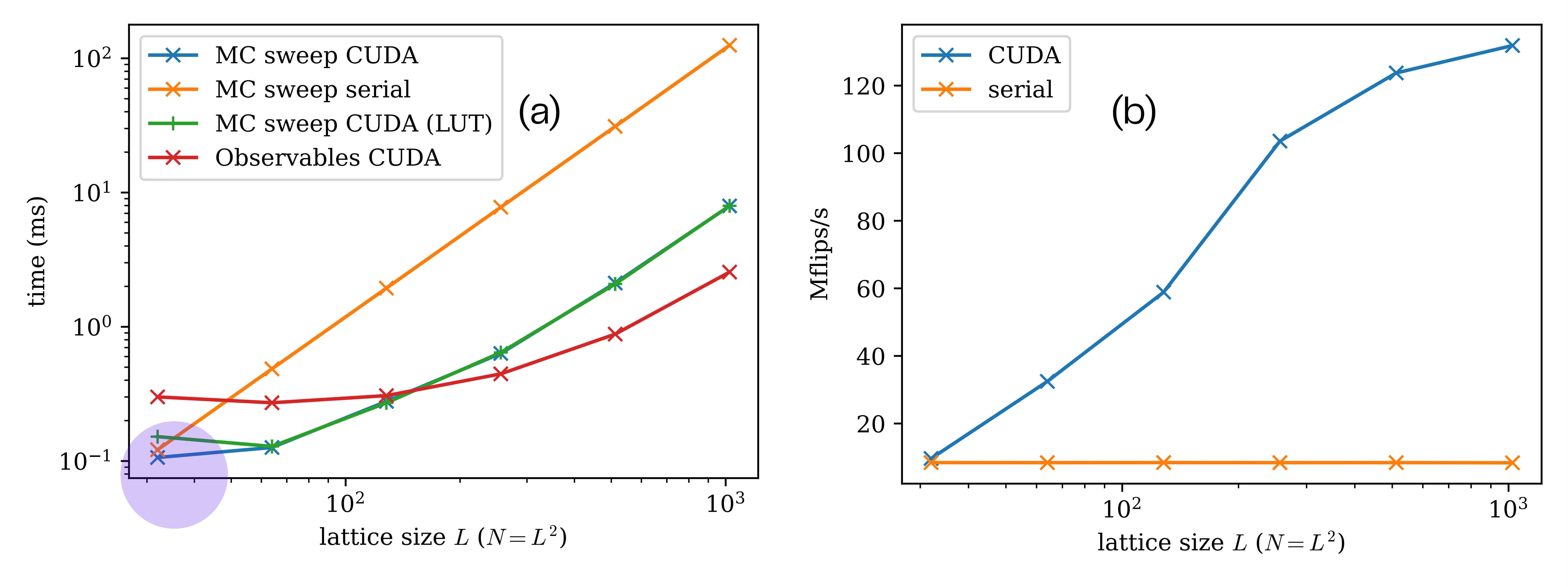
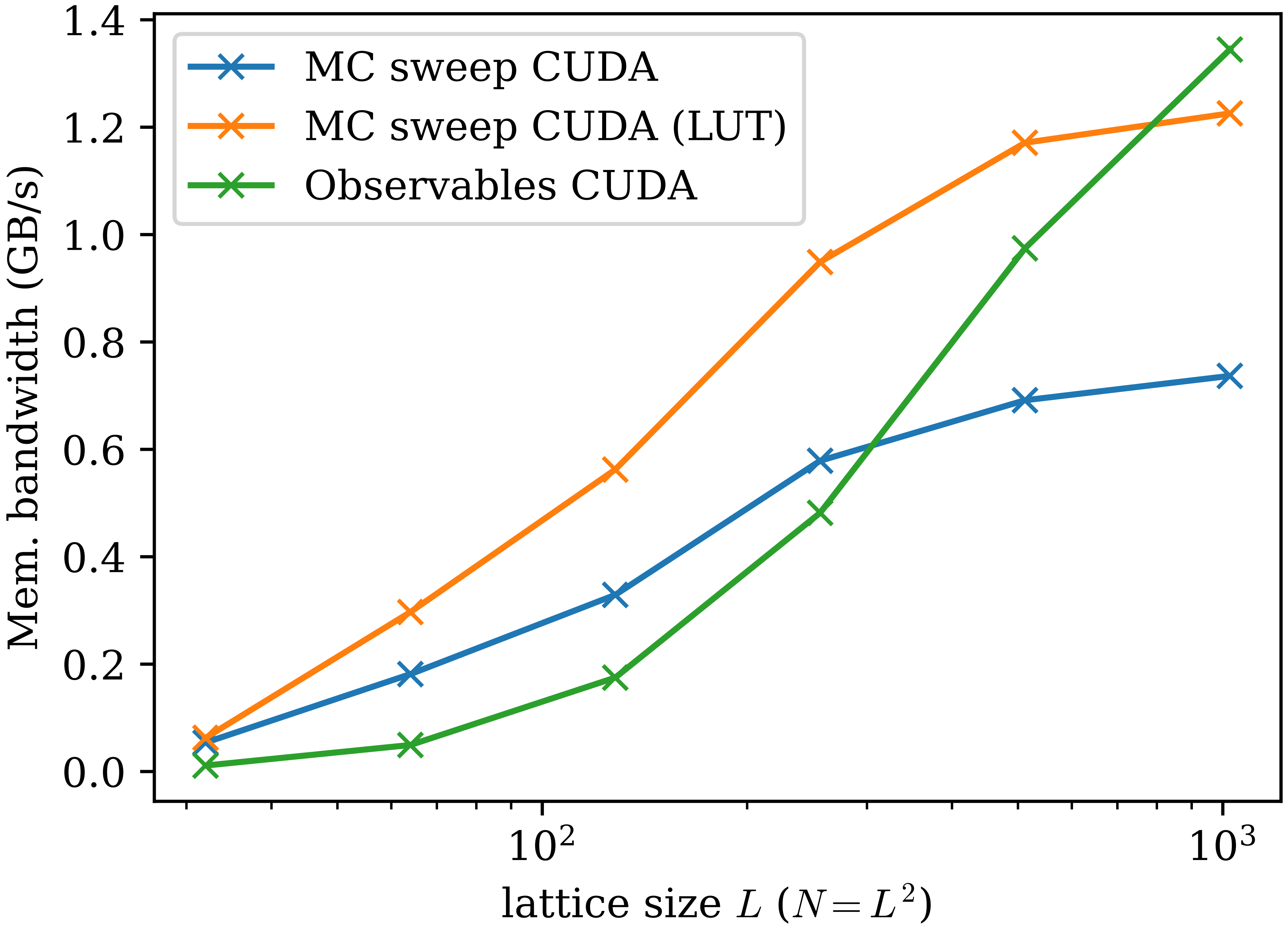


Figure: Highlighted is a region where **LUTs actually worsens performance**, while it becomes irrelevant for large enough lattices. This is because we're limited by memory accesses more than compute. This is expected and reiterates how optimizing serial code is fundamentally different from optimizing parallel code.

Memory bandwidth

Figure: Average memory bandwidth of various subtasks in parallel code. Again the higher memory pressure of LUTs is visible, with no performance improvement. Observable calculation requires 1.5 full lattice accesses from gloabl memory and the **transfer of block results** to host memory



Threads per block