```
/****
 *
 * GPU accelerated Monte Carlo simulation of the 2D Ising model
 *
 * Copyright (C) 2008 Tobias Preis (http://www.tobiaspreis.de)
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 3 of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public
 * License along with this program; if not, see
 * http://www.gnu.org/licenses/.
 *
 * Related publication:
 *
 * T. Preis, P. Virnau, W. Paul, and J. J. Schneider,
 * Journal of Computational Physics 228, 4468-4477 (2009)
 * doi:10.1016/j.jcp.2009.03.018
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <cutil.h>

#define FLAG_PRINT_SPINS 0
#define FLAG_ENERGY 0
#define T_START 3.00
#define T_FACTOR 0.9
#define T_END 2.00
#define GLOBAL_ITERATIONS 100
#define RANDOM_A 1664525
#define RANDOM_B 1013904223

#define BLOCK_SIZE 256

const unsigned int N=4*BLOCK_SIZE*BLOCK_SIZE;
const unsigned int n=2*BLOCK_SIZE;

/****
 *
 *  Function declaration
 *
 */
void calc(int argc,char** argv);
void cpu_function(double*,int*);
__global__ void device_function_main(int*,int*,int*,float,bool);

/****
 *
 *  Main function
 *
 */
int main(int argc,char** argv) {
```

```
  calc(argc,argv);
}

/****
 *
 *  Calc
 *
 */
void calc(int argc,char** argv) {

  printf("
-------------------------------------------------------------------------- \n");
  printf(" *\n");
  printf(" *  GPU accelerated Monte Carlo simulation of the 2D Ising model\n");
  printf(" *\n");
  printf(" *  Copyright (C) 2008 Tobias Preis (http://www.tobiaspreis.de)\n");
  printf(" *\n");
  printf(" *  This program is free software; you can redistribute it and/or\n");
  printf(" *  modify it under the terms of the GNU General Public License\n");
  printf(" *  as published by the Free Software Foundation; either version\n");
  printf(" *  3 of the License, or (at your option) any later version.\n");
  printf(" *\n");
  printf(" *  This program is distributed in the hope that it will be use-
ful,\n");
  printf(" *  but WITHOUT ANY WARRANTY; without even the implied warranty
of\n");
  printf(" *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the\n");
  printf(" *  GNU General Public License for more details.\n");
  printf(" *\n");
  printf(" *  You should have received a copy of the GNU General Public\n");
  printf(" *  License along with this program; if not, see\n");
  printf(" *  http://www.gnu.org/licenses/\n");
  printf(" *\n");
  printf(" *  Related publication:\n");
  printf(" *\n");
  printf(" *  T. Preis, P. Virnau, W. Paul, and J. J. Schneider,\n");
  printf(" *  Journal of Computational Physics 228, 4468-4477 (2009)\n");
  printf(" *  doi:10.1016/j.jcp.2009.03.018\n");
  printf(" *\n");

  printf(" ----------------------------- Ising model
----------------------------- \n");
  printf(" Number of Spins: %d \n",N);
  printf(" Start Temperature: %f \n",T_START);
  printf(" Decreasing Factor: %f \n",T_FACTOR);
  printf(" Final Temperature: %f \n",T_END);
  printf(" Global Iterations: %d \n",GLOBAL_ITERATIONS);

  //Init
  CUT_DEVICE_INIT(argc,argv);
  srand48(23);

  //Allocate and init host memory for output arrays
  int num_entries=0;
  for(double t=T_START; t>=T_END; t=t*T_FACTOR) num_entries++;
  unsigned int mem_out_size=sizeof(float)*num_entries;
  float* h_T=(float*) malloc(mem_out_size);
  float* h_E=(float*) malloc(mem_out_size);
  unsigned int mem_ref_out_size=sizeof(double)*num_entries;
  double* h_ref_E=(double*) malloc(mem_ref_out_size);
  num_entries=0;
  for(double t=T_START; t>=T_END; t=t*T_FACTOR) {
    h_T[num_entries]=t;
    num_entries++;
```

```
  }

  //Allocate and init host memory for simulation arrays
  unsigned int mem_size=sizeof(int)*N;
  unsigned int mem_size_random=sizeof(int)*BLOCK_SIZE*BLOCK_SIZE;
  int* h_random_data=(int*) malloc(mem_size_random);
  int* h_S=(int*) malloc(mem_size);
  unsigned int mem_size_out=sizeof(int)*BLOCK_SIZE;
  int* h_out=(int*) malloc(mem_size_out);
  h_random_data[0]=1;
  for(int i=1;i<BLOCK_SIZE*BLOCK_SIZE;i++) {
    h_random_data[i]=16807*h_random_data[i-1];
  }
  for(int i=0;i<N;i++) {
    if(drand48()>0.5) h_S[i]=-1;
    else h_S[i]=1;
  }

  //Create and start timer
  float gpu_sum=0;
  unsigned int timer=0;
  CUDA_SAFE_CALL(cudaThreadSynchronize());
  CUT_SAFE_CALL(cutCreateTimer(&timer));
  CUT_SAFE_CALL(cutStartTimer(timer));

  //Allocate device memory for arrays
  int* d_random_data;
  int* d_S;
  int* d_out;
  CUDA_SAFE_CALL(cudaMalloc((void**) &d_random_data,mem_size_random));
  CUDA_SAFE_CALL(cudaMalloc((void**) &d_S,mem_size));
  CUDA_SAFE_CALL(cudaMalloc((void**) &d_out,mem_size_out));

  //Stop and destroy timer
  CUDA_SAFE_CALL(cudaThreadSynchronize());
  CUT_SAFE_CALL(cutStopTimer(timer));
  float gpu_dt_malloc=cutGetTimerValue(timer);
  gpu_sum+=gpu_dt_malloc;
  printf("\n ---------------------------------- GPU
---------------------------------- \n");
  printf(" Processing time on GPU for allocating: %f (ms) \n",gpu_dt_malloc);
  CUT_SAFE_CALL(cutDeleteTimer(timer));

  //Create and start timer
  timer=0;
  CUDA_SAFE_CALL(cudaThreadSynchronize());
  CUT_SAFE_CALL(cutCreateTimer(&timer));
  CUT_SAFE_CALL(cutStartTimer(timer));

  //Copy host memory to device and create mirror of d_S

CUDA_SAFE_CALL(cudaMemcpy(d_random_data,h_random_data,mem_size_random,cudaMemcpy
HostToDevice));
  CUDA_SAFE_CALL(cudaMemcpy(d_S,h_S,mem_size,cudaMemcpyHostToDevice));

  //Stop and destroy timer
  CUDA_SAFE_CALL(cudaThreadSynchronize());
  CUT_SAFE_CALL(cutStopTimer(timer));
  float gpu_dt_mem=cutGetTimerValue(timer);
  gpu_sum+=gpu_dt_mem;
  printf(" Processing time on GPU for memory transfer: %f (ms) \n",gpu_dt_mem);
  CUT_SAFE_CALL(cutDeleteTimer(timer));

  //Print spins
```

```
  if(FLAG_PRINT_SPINS) {
    CUDA_SAFE_CALL(cudaMemcpy(h_S,d_S,mem_size,cudaMemcpyDeviceToHost));
    for(int i=0;i<BLOCK_SIZE*2;i++) {
      for(int j=0;j<BLOCK_SIZE*2;j++) {
     if(h_S[i*BLOCK_SIZE*2+j]>0) printf("+ ");
     else printf("- ");
      }
      printf("\n");
    }
    printf("\n");
  }

  //Create and start timer
  timer=0;
  CUDA_SAFE_CALL(cudaThreadSynchronize());
  CUT_SAFE_CALL(cutCreateTimer(&timer));
  CUT_SAFE_CALL(cutStartTimer(timer));

  //Calc energy
  num_entries=0;
  dim3 threads(BLOCK_SIZE);
  dim3 grid(BLOCK_SIZE);
  for(float t=T_START;t>=T_END;t=t*T_FACTOR) {
    double avg_H=0;
    for(int
global_iteration=0;global_iteration<GLOBAL_ITERATIONS;global_iteration++) {
      device_function_main<<<grid,threads>>>(d_S,d_out,d_random_data,t,true);
      device_function_main<<<grid,threads>>>(d_S,d_out,d_random_data,t,false);


CUDA_SAFE_CALL(cudaMemcpy(h_out,d_out,mem_size_out,cudaMemcpyDeviceToHost));
      int energy_sum=0;
      for(int i=0;i<BLOCK_SIZE;i++) energy_sum+=h_out[i];
      avg_H+=(float)energy_sum/N;
    }
    h_E[num_entries]=avg_H/GLOBAL_ITERATIONS;
    num_entries++;
  }

  //Stop and destroy timer
  CUDA_SAFE_CALL(cudaThreadSynchronize());
  CUT_SAFE_CALL(cutStopTimer(timer));
  float gpu_dt_main=cutGetTimerValue(timer);
  gpu_sum+=gpu_dt_main;
  printf(" Processing time on GPU for main function: %f (ms) \n",gpu_dt_main);
  printf(" Total processing time on GPU: %f (ms) \n",gpu_sum);
  CUT_SAFE_CALL(cutDeleteTimer(timer));

  //Check kernel execution
  CUT_CHECK_ERROR("Kernel execution failed");

  //Print spins
  if(FLAG_PRINT_SPINS) {
    CUDA_SAFE_CALL(cudaMemcpy(h_S,d_S,mem_size,cudaMemcpyDeviceToHost));
    for(int i=0;i<BLOCK_SIZE*2;i++) {
      for(int j=0;j<BLOCK_SIZE*2;j++) {
     if(h_S[i*BLOCK_SIZE*2+j]>0) printf("+ ");
     else printf("- ");
      }
      printf("\n");
    }
  }

  //Create and start timer
```

```
  timer=0;
  CUDA_SAFE_CALL(cudaThreadSynchronize());
  CUT_SAFE_CALL(cutCreateTimer(&timer));
  CUT_SAFE_CALL(cutStartTimer(timer));

  //Reference solution
  cpu_function(h_ref_E,h_S);

  //Print spins
  if(FLAG_PRINT_SPINS) {
    printf("\n");
    for(int i=0;i<BLOCK_SIZE*2;i++) {
      for(int j=0;j<BLOCK_SIZE*2;j++) {
     if(h_S[i*BLOCK_SIZE*2+j]>0) printf("+ ");
     else printf("- ");
      }
      printf("\n");
    }
  }

  //Stop and destroy timer
  CUDA_SAFE_CALL(cudaThreadSynchronize());
  CUT_SAFE_CALL(cutStopTimer(timer));
  float cpu_sum=cutGetTimerValue(timer);
  printf("\n ---------------------------------- CPU
---------------------------------- \n");
  printf(" Total processing time on CPU: %f (ms) \n",cpu_sum);
  CUT_SAFE_CALL(cutDeleteTimer(timer));
  printf("\n Speedup: %fX \n\n",(cpu_sum/gpu_sum));

  //Cleaning memory
  free(h_T);
  free(h_E);
  free(h_ref_E);
  free(h_random_data);
  free(h_S);
  free(h_out);
  CUDA_SAFE_CALL(cudaFree(d_random_data));
  CUDA_SAFE_CALL(cudaFree(d_S));
  CUDA_SAFE_CALL(cudaFree(d_out));
}

/****
 *
 *  Device function main
 *
 */
__global__ void device_function_main(int* S,int* out,int* R,float t,bool flag) {

  //Energy variable
  int dH=0;
  float exp_dH_4=exp(-(4.0)/t);
  float exp_dH_8=exp(-(8.0)/t);

  //Allocate shared memory
  __shared__ int r[BLOCK_SIZE];

  //Load random data
  r[threadIdx.x]=R[threadIdx.x+BLOCK_SIZE*blockIdx.x];
  __syncthreads();

  if(flag) {

    //Create new random numbers
```

```
    r[threadIdx.x]=RANDOM_A*r[threadIdx.x]+RANDOM_B;

    //Spin update top left
    if(blockIdx.x==0) { //Top
      if(threadIdx.x==0) { //Left
        dH=2*S[2*threadIdx.x]*(
                                S[2*threadIdx.x+1]+
                                S[2*threadIdx.x-1+2*BLOCK_SIZE]+
                                S[2*threadIdx.x+2*BLOCK_SIZE]+
                                S[2*threadIdx.x+N-2*BLOCK_SIZE]);
      }
      else {
        dH=2*S[2*threadIdx.x]*(
                                S[2*threadIdx.x+1]+
                                S[2*threadIdx.x-1]+
                                S[2*threadIdx.x+2*BLOCK_SIZE]+
                                S[2*threadIdx.x+N-2*BLOCK_SIZE]);
      }
    }
    else {
      if(threadIdx.x==0) { //Left
        dH=2*S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x]*(

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+1]+

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x-1+2*BLOCK_SIZE]+

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]+

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x-2*BLOCK_SIZE]);
      }
      else {
        dH=2*S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x]*(

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+1]+

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x-1]+

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]+

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x-2*BLOCK_SIZE]);
      }
    }

    if(dH==4) {
      if(fabs(r[threadIdx.x]*4.656612e-10)<exp_dH_4) {

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x]=-S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.
x];
      }
    }
    else if(dH==8) {
      if(fabs(r[threadIdx.x]*4.656612e-10)<exp_dH_8) {

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x]=-S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.
x];
      }
    }
    else {

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x]=-S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.
x];
    }
```

```
    //Create new random numbers
    r[threadIdx.x]=RANDOM_A*r[threadIdx.x]+RANDOM_B;

    //Spin update bottom right
    if(blockIdx.x==BLOCK_SIZE-1) { //Bottom
      if(threadIdx.x==BLOCK_SIZE-1) { //Right
        dH=2*S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]*(

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2]+

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]+

S[2*threadIdx.x+1]+

S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x]);
      }
      else {
        dH=2*S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]*(

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE+2]+

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]+

S[2*threadIdx.x+1]+

S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x]);
      }
    }
    else {
      if(threadIdx.x==BLOCK_SIZE-1) { //Right
        dH=2*S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]*(

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2]+

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]+

S[2*threadIdx.x+1+4*BLOCK_SIZE*(blockIdx.x+1)]+

S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x]);
      }
      else {
        dH=2*S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]*(

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE+2]+

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]+

S[2*threadIdx.x+1+4*BLOCK_SIZE*(blockIdx.x+1)]+

S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x]);
      }
    }

    if(dH==4) {
      if(fabs(r[threadIdx.x]*4.656612e-10)<exp_dH_4) {

S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]=-S[2*threadIdx.x+1+4*BLO
CK_SIZE*blockIdx.x+2*BLOCK_SIZE];
      }
    }
    else if(dH==8) {
      if(fabs(r[threadIdx.x]*4.656612e-10)<exp_dH_8) {
```

```
S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]=-S[2*threadIdx.x+1+4*BLO
CK_SIZE*blockIdx.x+2*BLOCK_SIZE];
      }
    }
    else {

S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]=-S[2*threadIdx.x+1+4*BLO
CK_SIZE*blockIdx.x+2*BLOCK_SIZE];
    }

    __syncthreads();

  }
  else {

    //Create new random numbers
    r[threadIdx.x]=RANDOM_A*r[threadIdx.x]+RANDOM_B;

    //Spin update top right
    if(blockIdx.x==0) { //Top
      if(threadIdx.x==BLOCK_SIZE-1) { //Right
        dH=2*S[2*threadIdx.x+1]*(
                                S[2*threadIdx.x+2-2*BLOCK_SIZE]+
                                S[2*threadIdx.x]+
                                S[2*threadIdx.x+1+2*BLOCK_SIZE]+
                                S[2*threadIdx.x+1+N-2*BLOCK_SIZE]);
      }
      else {
        dH=2*S[2*threadIdx.x+1]*(
                                S[2*threadIdx.x+2]+
                                S[2*threadIdx.x]+
                                S[2*threadIdx.x+1+2*BLOCK_SIZE]+
                                S[2*threadIdx.x+1+N-2*BLOCK_SIZE]);
      }
    }
    else {
      if(threadIdx.x==BLOCK_SIZE-1) { //Right
        dH=2*S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x]*(

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2-2*BLOCK_SIZE]+

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x]+

S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]+

S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x-2*BLOCK_SIZE]);
      }
      else {
        dH=2*S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x]*(

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2]+

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x]+

S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]+

S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x-2*BLOCK_SIZE]);
      }
    }

    if(dH==4) {
      if(fabs(r[threadIdx.x]*4.656612e-10)<exp_dH_4) {
```

```
S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x]=-S[2*threadIdx.x+1+4*BLOCK_SIZE*block
Idx.x];
      }
    }
    else if(dH==8) {
      if(fabs(r[threadIdx.x]*4.656612e-10)<exp_dH_8) {

S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x]=-S[2*threadIdx.x+1+4*BLOCK_SIZE*block
Idx.x];
      }
    }
    else {

S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x]=-S[2*threadIdx.x+1+4*BLOCK_SIZE*block
Idx.x];
    }

    //Create new random numbers
    r[threadIdx.x]=RANDOM_A*r[threadIdx.x]+RANDOM_B;

    //Spin update bottom left
    if(blockIdx.x==BLOCK_SIZE-1) { //Bottom
      if(threadIdx.x==0) { //Left
        dH=2*S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]*(

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE+1]+

S[2*threadIdx.x+4*BLOCK_SIZE*(blockIdx.x+1)-1]+

S[2*threadIdx.x]+

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x]);
      }
      else {
        dH=2*S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]*(

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE+1]+

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE-1]+

S[2*threadIdx.x]+

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x]);
      }
    }
    else {
      if(threadIdx.x==0) { //Left
        dH=2*S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]*(

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE+1]+

S[2*threadIdx.x+4*BLOCK_SIZE*(blockIdx.x+1)-1]+

S[2*threadIdx.x+4*BLOCK_SIZE*(blockIdx.x+1)]+

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x]);
      }
      else {
        dH=2*S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]*(

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE+1]+

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE-1]+
```

```
S[2*threadIdx.x+4*BLOCK_SIZE*(blockIdx.x+1)]+

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x]);
      }
    }

    if(dH==4) {
      if(fabs(r[threadIdx.x]*4.656612e-10)<exp_dH_4) {

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]=-S[2*threadIdx.x+4*BLOCK_S
IZE*blockIdx.x+2*BLOCK_SIZE];
      }
    }
    else if(dH==8) {
      if(fabs(r[threadIdx.x]*4.656612e-10)<exp_dH_8) {

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]=-S[2*threadIdx.x+4*BLOCK_S
IZE*blockIdx.x+2*BLOCK_SIZE];
      }
    }
    else {

S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]=-S[2*threadIdx.x+4*BLOCK_S
IZE*blockIdx.x+2*BLOCK_SIZE];
    }

  }

  //Transfer random data back to global memory
  R[threadIdx.x+BLOCK_SIZE*blockIdx.x]=r[threadIdx.x];

  if(!flag) {

    //For reduction shared memory array r is used
    if(FLAG_ENERGY) {

      //Calc energy
      if(blockIdx.x==BLOCK_SIZE-1) { //Bottom
     if(threadIdx.x==BLOCK_SIZE-1) { //Right

dH=-S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x]*(S[2*threadIdx.x+4*BLOCK_SIZE*block
Idx.x+1]+S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE])

-S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x]*(S[2*threadIdx.x+1+4*BLOCK_SIZE*bloc
kIdx.x+1-2*BLOCK_SIZE]+S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE])

-S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]*(S[2*threadIdx.x+4*BLOCK_
SIZE*blockIdx.x+1+2*BLOCK_SIZE]+S[2*threadIdx.x])

-S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]*(S[2*threadIdx.x+4*BLOC
K_SIZE*blockIdx.x+2]+S[2*threadIdx.x+1]);
     }
     else {

dH=-S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x]*(S[2*threadIdx.x+4*BLOCK_SIZE*block
Idx.x+1]+S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE])

-S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x]*(S[2*threadIdx.x+1+4*BLOCK_SIZE*bloc
kIdx.x+1]+S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE])

-S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]*(S[2*threadIdx.x+4*BLOCK_
SIZE*blockIdx.x+1+2*BLOCK_SIZE]+S[2*threadIdx.x])
```

```
-S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]*(S[2*threadIdx.x+4*BLOC
K_SIZE*blockIdx.x+2+2*BLOCK_SIZE]+S[2*threadIdx.x+1]);
     }
      }
      else {
     if(threadIdx.x==BLOCK_SIZE-1) { //Right

dH=-S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x]*(S[2*threadIdx.x+4*BLOCK_SIZE*block
Idx.x+1]+S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE])

-S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x]*(S[2*threadIdx.x+1+4*BLOCK_SIZE*bloc
kIdx.x+1-2*BLOCK_SIZE]+S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE])

-S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]*(S[2*threadIdx.x+4*BLOCK_
SIZE*blockIdx.x+1+2*BLOCK_SIZE]+S[2*threadIdx.x+4*BLOCK_SIZE*(blockIdx.x+1)])

-S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]*(S[2*threadIdx.x+4*BLOC
K_SIZE*blockIdx.x+2]+S[2*threadIdx.x+1+4*BLOCK_SIZE*(blockIdx.x+1)]);
     }
     else {

dH=-S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x]*(S[2*threadIdx.x+4*BLOCK_SIZE*block
Idx.x+1]+S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE])

-S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x]*(S[2*threadIdx.x+1+4*BLOCK_SIZE*bloc
kIdx.x+1]+S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE])

-S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]*(S[2*threadIdx.x+4*BLOCK_
SIZE*blockIdx.x+1+2*BLOCK_SIZE]+S[2*threadIdx.x+4*BLOCK_SIZE*(blockIdx.x+1)])

-S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]*(S[2*threadIdx.x+4*BLOC
K_SIZE*blockIdx.x+2+2*BLOCK_SIZE]+S[2*threadIdx.x+1+4*BLOCK_SIZE*(blockIdx.x+1)]
);
     }
      }
      __syncthreads();

    }
    else {

      //Calc magnetisation
      dH=S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x]
     +S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x]
     +S[2*threadIdx.x+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE]
     +S[2*threadIdx.x+1+4*BLOCK_SIZE*blockIdx.x+2*BLOCK_SIZE];
      __syncthreads();

    }

    //Save partial results back to shared memory in new structure
    r[threadIdx.x]=dH;

    //Reduction on GPU
    for(unsigned int dx=1;dx<BLOCK_SIZE;dx*=2) {
      if(threadIdx.x%(2*dx)==0) {
     r[threadIdx.x]+=r[threadIdx.x+dx];
      }
      __syncthreads();
    }

    //Save in out
    if(threadIdx.x==0) out[blockIdx.x]=r[0];
  }
```

```
}

/****
 *
 *  CPU function
 *
 */
void cpu_function(double* E, int* S) {

  int random=23;
  int num_entries=0;

  for(double t=T_START;t>=T_END;t=t*T_FACTOR) {
    double avg_H=0;
    double exp_dH_4=exp(-(4.0)/t);
    double exp_dH_8=exp(-(8.0)/t);

    for(int
global_iteration=0;global_iteration<GLOBAL_ITERATIONS;++global_iteration) {
     if(FLAG_ENERGY) {
     //Energy
     double H=0;
     for(int x=0;x<n;++x) {
       for(int y=0;y<n;++y) {
         int xr=x+1,yd=y+1;
         if(xr==n) xr=0;
         if(yd==n) yd=0;
         H+=-S[y*n+x]*(S[y*n+xr]+S[yd*n+x]);
       }
     }
     avg_H+=H/N;
      }
      else {
     //Magnetisation
     double H=0;
     for(int x=0;x<N;++x) {
       H+=S[x];
     }
     avg_H+=H/N;
      }

      for(int x=0;x<n;++x) {
     for(int y=0;y<n;++y) {
       if((y*(n+1)+x)%2==0) {
         int xl=x-1,yl=y,xu=x,yu=y-1,xr=x+1,yr=y,xd=x,yd=y+1;
         if(x==0) {
           xl=n-1;
         }
         else if(x==n-1) {
           xr=0;
         }
         if(y==0) {
           yu=n-1;
         }
         else if(y==n-1) {
           yd=0;
         }

         //Initial local energy
         int dH=2*S[y*n+x]*(
                     S[yl*n+xl]+
                     S[yr*n+xr]+
                     S[yu*n+xu]+
                     S[yd*n+xd]
```

```
                );

    if(dH==4) {
      random=RANDOM_A*random+RANDOM_B;
      if(fabs(random*4.656612e-10)<exp_dH_4) {
     S[y*n+x]=-S[y*n+x];
      }
    }
    else if(dH==8) {
      random=RANDOM_A*random+RANDOM_B;
      if(fabs(random*4.656612e-10)<exp_dH_8) {
     S[y*n+x]=-S[y*n+x];
      }
    }
    else {
      S[y*n+x]=-S[y*n+x];
    }
  }
}
 }

 for(int x=0;x<n;++x) {
for(int y=0;y<n;++y) {
  if((y*(n+1)+x)%2==1) {
    int xl=x-1,yl=y,xu=x,yu=y-1,xr=x+1,yr=y,xd=x,yd=y+1;
    if(x==0) {
      xl=n-1;
    }
    else if(x==n-1) {
      xr=0;
    }
    if(y==0) {
      yu=n-1;
    }
    else if(y==n-1) {
      yd=0;
    }

    //Initial local energy
    int dH=2*S[y*n+x]*(
                S[yl*n+xl]+
                S[yr*n+xr]+
                S[yu*n+xu]+
                S[yd*n+xd]
                );

    if(dH==4) {
      random=RANDOM_A*random+RANDOM_B;
      if(fabs(random*4.656612e-10)<exp_dH_4) {
     S[y*n+x]=-S[y*n+x];
      }
    }
    else if(dH==8) {
      random=RANDOM_A*random+RANDOM_B;
      if(fabs(random*4.656612e-10)<exp_dH_8) {
     S[y*n+x]=-S[y*n+x];
      }
    }
    else {
      S[y*n+x]=-S[y*n+x];
    }
  }
}
 }
```

```
    }
    E[num_entries]=avg_H/GLOBAL_ITERATIONS;
    num_entries++;
  }
}
```