BULETINUL INSTITUTULUI POLITEHNIC DIN IAŞI
(BULLETIN OF THE POLYTECHNIC INSTITUTE OF IAŞI)
Published by "Gheorghe Asachi" Technical University of Iaşi
Section Mathematics. Theoretical Mechanics. Physics

Volume 70(74), No. 1-2, 2024
DOI: 10.2478/bipmf-2024-0001

GPGPU PROGRAMMING FOR DIPOLAR FIELD CALCULATION

SEBASTIAN PRICOP[1] and RĂZVAN VASILE ABABEI[1,2,*]

[1] "Alexandru Ioan Cuza" University, Faculty of Physics, Blvd. Carol I, no 11, 700753, Iași, Romania
[2] "Alexandru Ioan Cuza" University, Research Center with Integrated Techniques for Atmospheric Aerosol Investigation in Romania, RECENT Air, Laboratory of astronomy and astrophysics, Astronomy Observatory, 5-7 Mihail Sadoveanu, 700490 Iași, Romania
**Corresponding author e-mail: razvan.ababei@uaic.ro*

Citation:

PRICOP SEBASTIAN and ABABEI RĂZVAN VASILE

GPGPU PROGRAMMING FOR DIPOLAR FIELD CALCULATION

BULLETIN OF THE POLYTECHNIC INSTITUTE OF IAȘI, 2024, 70(74), 1-2, 7-18;

DOI: 10.2478/bipmf-2024-0001.

Received: November 14, 2023

Accepted: January 8, 2024

Publication: January 22, 2024

© 2024 *Pricop S. and Ababei R.V.*

This is an open access article licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND 4.0)

Abstract. Accelerating computational processes is paramount in numerical infrastructure development, particularly in applications such as the finite element method (FEM) and extensive calculations for simulating 3D processes in materials. In this work, we introduce a novel technique for computing the magnetostatic field of an ellipsoid particle, leveraging CUDA on a graphical card for parallel processing. The implementation on a GPU resulted in a remarkable 20-fold improvement in calculation speed. This achievement not only expedites research tasks, but also enables the exploration of larger and more intricate simulations, facilitating quicker model refinements and deeper insights into material behaviours under various conditions. The utilization of GPU computing aligns with the broader trend in scientific research and engineering, offering a versatile solution for diverse computational challenges beyond this specific task of magnetism. Overall, our work contributes to the ongoing effort to harness high-performance computing (HPC) technologies for accelerated and more efficient simulations in materials science and related fields.

Keywords: CUDA, GPGPU programming, Magnetostatic Field.

1. Introduction

In recent decades, High-Performance Computing (HPC) has made significant strides, primarily attributed to the extensive parallelization and optimization of algorithms. Presently, envisioning scientific computation or numerical modeling without the incorporation of parallel algorithms and substantial computational resources (both hardware and software) is inconceivable. This necessity is generated from the capability to execute numerous operations concurrently, highlighting the indispensable role of parallelization in contemporary computational approaches.

The magnetostatic interaction is a particular case of dipolar field interaction which arises from two magnetic particles (or two magnetic dipoles) separated by a certain distance **r**. Here, is the expression for magnetostatic energy [1] is given by:

$$E = \frac{\mu_0}{4\,\pi r^3}\left[\boldsymbol{\mu_1}\cdot\boldsymbol{\mu_2} - \frac{3}{r^2}(\boldsymbol{\mu_1}\cdot\boldsymbol{r})(\boldsymbol{\mu_2}\cdot\boldsymbol{r})\right], \tag{1}$$

where μ_1, μ_2 denotes the magnetic moment of those particles. As depicted in the formula above, the effect is greater with an increased radius, making it much more effective at longer distances. This interaction is also known as a long-range interaction. This energy significantly contributes to any magnetic Hamiltonian system and holds importance in the calculation across various micro-magnetic simulations, including recording media, hyperthermia, magnetic sensors, nanoparticles, etc.

However, in some cases, including its contribution becomes prohibitively expensive due to the long computational time required. For instance, Evans et al. [2] demonstrated that for a relatively small particle, the runtime for a single update on an individual core can be roughly 5 seconds. Now, let's consider a process that is supposed to perform millions of updates of total energy, including the dipolar energy. In such a scenario, the calculation of magnetostatic energy might take millions of seconds (over one month), making it excessively costly for computation.

Therefore, in this work, we will present a couple of methods to optimize and parallelize the dipolar field calculation, such as the brute force method implemented both in serial and CUDA, highlighting the main advantages and disadvantages of each.

2. Methods

So far, the easiest way to implement the dipole field calculation is the most trivial one, also, known as the brute method which is supposed to calculate the interactions between each particle with all neighbouring particles. It is a **N**x**N** floating point operations where **N** is the total

number of particles. Another elegant way of doing this is to split the system in macro-cells, each macro-cell containing a certain number of particles with following coordinates determined from the magnetic" centre of mass" given by next equation [2]:

$$p_{mc} = \sum_i^N \frac{\mu_i p_i^\alpha}{\mu_i}. \quad (2)$$

The magnetic moment for a single macro-cell will be given by the sum of all magnetic moments incorporated on that cell. It is quite simple but the efficiency increases as long the volume of the macro-cell increases as well. However, the accuracy may suffer with an increased macro-cell sizes, a good balance between efficiency and accuracy is needed. Also, this method can be combined with dipole approximation method where the radii of the particles can be pre-calculated. This can be a bit more difficult since we have to define a nine component tensor for the interaction of particle *i* with particle *j* but it requires only one loop over all sites which is a great advantage. These 9-components have been calculated by Newell *et al*. [3] generalizing the demagnetization tensor for an internal or external field due to the pattern of magnetization. Therefore, the relation between H and magnetization is looks like: H(α) = (3 × 3) × M where each component is independent on the other particle. One of the most difficult strategy to implement is Fourier transformations which is supposed to respect several conditions. First of all the arrangement of the particles must be periodical and the sizes should be finite. If our system fulfils all these requirements, then FFT method can be applied successfully by implementing three steps. The first step means to transform the fields, magnetization, and radius components into Fourier space using direct Fourier transformations followed by the calculation of magnetic fields. The last step consists into transformation of the Fourier fields and magnetization components to real space using revers Fourier transformations. This procedure is a bit more complicated due to the three steps and special mathematics required. It has been successfully implemented in many software packages dedicated to simulate micro-magnetic processes such as MuMax [4], OOMMF [5], VAMPIRE [2] etc. providing good scalability and high accuracy of the results.

Our C++ code follows a sequence of four steps as depicted in Figure 1, emphasizing the necessity to maintain their order. The computational workload is efficiently handled by the GPU's threads, where each thread is responsible for computing the magnetostatic field of individual atoms in the system. This procedure necessitates specific CUDA statements, presented in Appendix A, including operations like transferring arrays to GPU memory, invoking the routine by specifying the

required number of threads and devices, and modifying the recursive loops for CUDA integration. Lastly, it is important to copy back the results into CPU memory leaving the GPU memory empty.

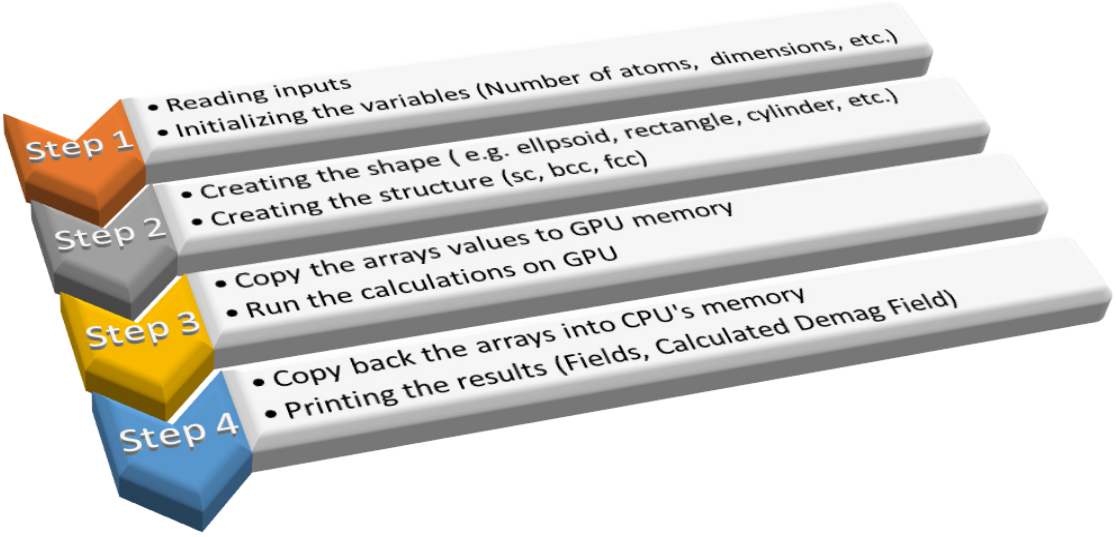


Fig. 1. Flowchart illustrating the main steps of the program.

3. Results

For the parallel version of the code, we opted for the CUDA approach due to its straightforward scalability and ease of implementation. As mentioned earlier, CUDA offers distinct advantages over other methods, being more flexible and highly efficient, resulting in significant speed improvements in calculations. I emphasize that the future of High-Performance Computing will likely hinge on GPGPU programming. This trend is evidenced by numerous trademarked systems that involve extensive computation in statistical processes and security algorithms, having shifted towards OpenCL and CUDA platforms to drastically reduce energy and resource consumption.

Returning to the calculation of the dipole field, we compared two approaches using a simple algorithm known as the brute force algorithm. Implementing this algorithm on CUDA resulted in a speed-up of over 20 times. However, we are confident that an optimized algorithm such as the Fast Fourier Transform (FFT) method on CUDA could achieve exceptional performance and scalability. Hence, we are currently in the process of developing a cuFFT method. Utilizing FFT methods in a CUDA version necessitates specific libraries like cuFFT. These libraries enable the transformation of an N^2-dimensional problem into an N*log(N) dimension by leveraging the periodicity of the crystal structure.

This will reduce the computational time as the CPU must perform less calculation with a factor of *f*:

$$f=N/\log(N)$$

It's important to note that this approach is applicable solely to crystal lattices devoid of defects or impurities in order to respect the periodicity of the magnetostatic field.

At present, we offer a comparison between the brute force method implemented on CPU and GPU, illustrating how scalability varies with the number of threads. For the CPU tests, we utilized level 3 optimization by adding the flag -O3 during code compilation. Our GPU units, installed on the Anlaf Cluster, comprise older graphics cards such as the GT 218 with Tesla architecture [6], supporting the initial version of CUDA. To execute the CUDA version, please ensure your device supports at least CUDA 1.1, and if you are running it on a cluster, a job script submission may be necessary. I have included the required script for this purpose.

Regarding the CPU tests, I employed level 3 of optimization (O3) on an Intel Xeon processor clocked at 2.00 GHz, using the GCC compiler version 4.4.6.

To showcase the scalability of this method, we conducted tests on three systems of different sizes: a small system containing 63,000 particles (atomic sites), a medium system with 71,674 atomic sites, and a larger system containing 125,000 atomic sites. Figure 2 illustrates how the runtime varies with the number of threads used in the aforementioned cases.

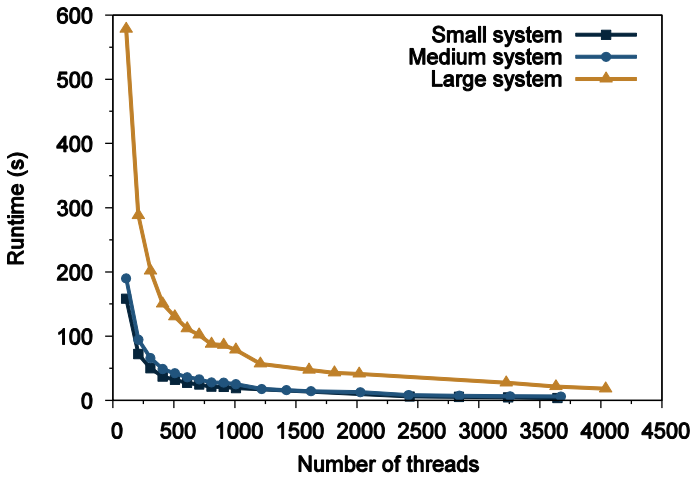


Fig. 2. Runtime scaling of Demag code for three different cases: small, medium, and large system.

The figure above demonstrates an extraordinary exponential decrease in runtime facilitated by CUDA with an increased number of threads. It's noteworthy that while more threads could be utilized, their efficiency is most pronounced when scaling up the system sizes. For smaller systems, there exists a "saturation" point due to synchronization, where the number of threads covers all sites.

Figure 3 is presented in a log-log plot alongside the ideal scaling to aid in comprehending the relationship. To assess the efficiency of this scalability, the following graphs display the speed-up achieved by CUDA for the aforementioned three different cases. To calculate the speed-up factor, we divided the CPU runtime by the time required for different numbers of threads. Consequently, we obtained a runtime of 97 seconds for the small system, 128 seconds for the medium-sized system, and 387 seconds for the large system in serial code.

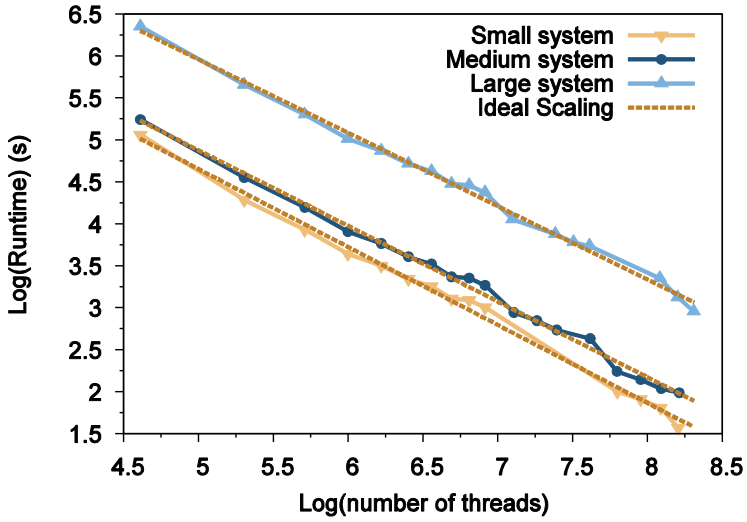


Fig. 3. In this graph are plotted the log-log scale of the Figure 2 including ideal scaling.

The graph below shown in Figure 4 showcases an intriguing trend where, for several threads numbering under 100, the speed-up factor remains less than 1. This observation implies that individual CUDA cores exhibit lesser computational power compared to the CPU. However, it's crucial to consider that with modern graphic cards, the capability to utilize more than millions of threads becomes feasible, resulting in an exceptional performance boost.

This shift in perspective highlights the potential of harnessing the power of newer graphic cards, where the sheer volume of available threads contributes to an outstanding performance. In the context of a system characterized by dimensions such as rx = ry = 10nm and rz = 20nm, comprising a substantial 579,349 atoms, we obtained noteworthy results for the magnetostatic field, as illustrated in Figure 5. These findings shed light on the intricate behavior of the magnetostatic field within this particular system, offering valuable insights into its characteristics and interactions.

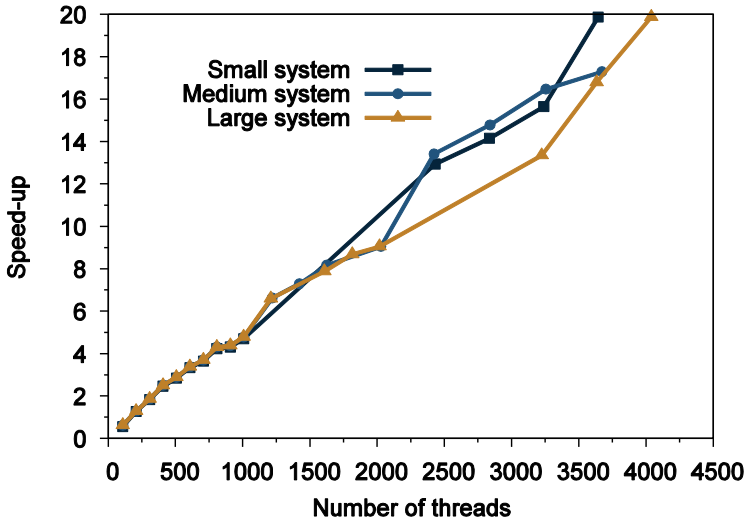


Fig. 4. In this graph it is illustrated the speed-up factor as a function of number of threads. The speed-up factor is calculated with this formula: Su =Tthreads/Tcpu.

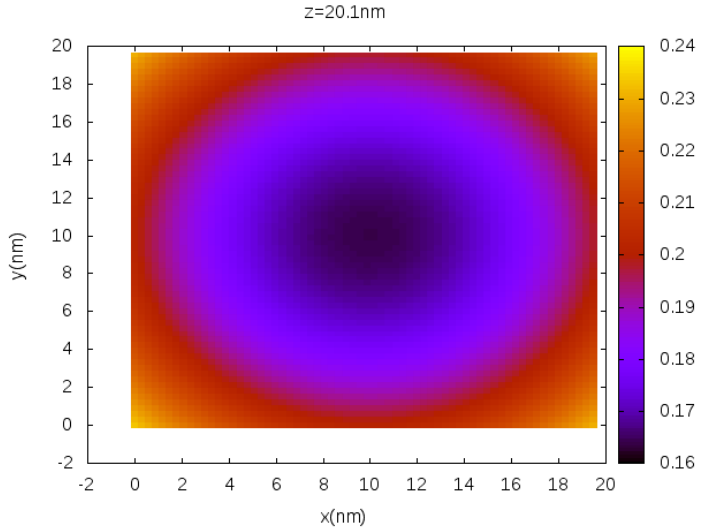


Fig. 5. In this image is represented a transversal plane of the ellipsoid at r = 0.1nm corresponding to a z coordinate of 20.1nm simulated on CUDA version. The color represents the strength of the magnetic field in Tesla. The runtime taken for this simulation is 268.059 seconds with a bandwidth of 4.02 GB/s running on approximatively 1 million threads.

4. Conclusions and further work

CUDA platform stands out as an immensely potent tool in scientific computation, particularly when faced with extensive calculations. In the context of demagnetization field calculations, the scalability achieved aligns remarkably with the ideal scenario, resulting in a significant speed-up factor of 20 times faster compared to the serial code implementation. This substantial enhancement in computational speed not only reduces the time required for calculations

but also empowers the simulation of larger structures with smaller mesh cells, providing a more detailed view of complex systems in materials science.

Building upon this success, our subsequent objective revolves around leveraging the technique of multi-GPUs programming. This strategic move is geared toward enabling simulations that transcend the confines of micromagnetic scales. By harnessing the power of multiple GPUs working in tandem, we aim to extend our simulations to encompass structures that surpass the conventional limitations, thereby advancing our understanding of materials and their behaviors at an even broader scale.

Acknowledgments: the author is grateful to Doctor Richard Evans and Doctor Matthew Ellis. All the calculations have been performed by using the HPC systems from University of York.

REFERENCES

[1] Stephen Blundell, *Magnetism in Condensed Matter*, Oxford University Press.

[*2*] Evans Richard F.L. et al., *Atomistic spin model simulations of magnetic nanomaterials*, Journal of Physics: Condensed Matter, 26.10 (2014): 103202.

[3] Newell Andrew J., Wyn Williams and David J. Dunlop, *A generalization of the demagnetizing tensor for nonuniform* magnetization, Journal of Geophysical Research: Solid Earth 98.B6 (1993): 9551-9555.

[4] Vansteenkiste Arne et al., *The design and verification of* MuMax3, AIP advances 4.10 (2014).

[5] Mahalingam S. Sundara, Manikandan B.V. and Arockiaraj S., *Review–micromagnetic simulation using OOMMF and experimental investigations on nano composite magnets*, Journal of Physics: Conference Series. Vol. 1172, No. 1, IOP Publishing, 2019.

[6] https://en.wikipedia.org/wiki/CUDA

APPENDIX A:

```
// This code is dedicated to calculate the magnetostatic field and the demag field
// This code is coming with GNU licence and no warranty  is provided
// This code is accelerated using Cuda platform provided by nVidia
// This code is free to use and it can be downloaded from following link: https://github.com/maxxwave/dipole
// ------------------------------------------------------------------------------------------------------------------------
#include <iostream>
#include <cuda.h>
#include <cmath>
// this library is needed for our further fft method which is under progress
#include <cufft.h>
```

```
#include <stdio.h>
#include "../hdr/storage.h"
#include "../hdr/create.h"
#include "../hdr/cufields.h"
namespace cuda{
  // defining the pre-factor constant as a register for speed up the access
  //check_cuda_errors (__FILE__, __LINE__);
  // we declare this constant as a register to be very fast each time when it is called
  __device__ __constant__ double cupf =  0.00009274;;          // this is the pre-factor that incorporates the magnetic
        moment
  // and other. See field.cpp line 51
  __device__ __constant__ double cupf_2 = 8.0*3.14*0.3333333333333;
  __global__ void demag_field(const unsigned long int  index,
                              double * sx_d,
                              double * sy_d,
                              double * sz_d,
                              double * Hx_d,
                              double * Hy_d,
                              double * Hz_d,
                              double * x_d,
                              double * y_d,
                              double * z_d,
                              double * H_tot_d ){
     // declaring the radii components as a register in order to speed up the calculations
     register double dx=0.0;
     register double dy=0.0;
     register double dz=0.0;
     register double r=0.0;
     register double r_cube=0.0;

    // a synchronization point is needed to ensure all threads will work simultaneously
    __syncthreads();

    // loop over all threads and assign a thread per atom
```

```
// this loop is dedicated to cover all atoms(sites) if the number of atoms is larger than the maximum number of
      threads allowed
for ( register int tdx = blockIdx.x * blockDim.x + threadIdx.x;          //here we define the index
  tdx < index;
  tdx += blockDim.x * gridDim.x // we load a maximum threads allowed
  ){
  for(int j=0; j < index; j++){
            if(tdx!=j){
              dx = x_d[j] - x_d[tdx];
              dy = y_d[j] - y_d[tdx];
            dz = z_d[j] - z_d[tdx];
            r =  sqrt(dx*dx + dy*dy + dz*dz);
            r_cube = 1.0/r*r*r;
            Hx_d[tdx] += cupf * (3.0*dx*(dx*sx_d[j]) - sx_d[j])*r_cube;
            Hy_d[tdx] += cupf * (3.0*dy*(dy*sy_d[j]) - sy_d[j])*r_cube;
            Hz_d[tdx] += cupf * (3.0*dz*(dz*sz_d[j]) - sz_d[j])*r_cube;

      }// end of if
      else{
         Hx_d[tdx] += cupf*(cupf_2*sx_d[tdx]);
         Hy_d[tdx] += cupf*(cupf_2*sy_d[tdx]);
         Hz_d[tdx] += cupf*(cupf_2*sz_d[tdx]);
      }// end of else
       // synchronization point
       __syncthreads();
    } // end of j for
    H_tot_d[0] += Hx_d[tdx];
    H_tot_d[1] += Hy_d[tdx];
    H_tot_d[2] += Hz_d[tdx];
    //synchronization point
    __syncthreads();
  } //end of tdx for
 } // end of demag kernel
}//end of namepscare
```

```
cudaEvent_t start_3, stop_3;
// create events
cudaEventCreate(&start_3);
cudaEventCreate(&stop_3);
// start measuring
cudaEventRecord(start_3);

// copy data from device to host and de-allocate the memory into device
cudaMemcpy(&atom_h_x[0], d_x, sizeof(double)*create::index, cudaMemcpyDeviceToHost);
cudaMemcpy(&atom_h_y[0], d_y, sizeof(double)*create::index, cudaMemcpyDeviceToHost);
cudaMemcpy(&atom_h_z[0], d_z, sizeof(double)*create::index, cudaMemcpyDeviceToHost);
cudaMemcpy(&atom_h_sx[0], d_sx, sizeof(double)*create::index, cudaMemcpyDeviceToHost);
cudaMemcpy(&atom_h_sy[0], d_sy, sizeof(double)*create::index, cudaMemcpyDeviceToHost);
cudaMemcpy(&atom_h_sz[0], d_sz, sizeof(double)*create::index, cudaMemcpyDeviceToHost);
cudaMemcpy(&atom_h_Hx[0], d_Hx, sizeof(double)*create::index, cudaMemcpyDeviceToHost);
cudaMemcpy(&atom_h_Hy[0], d_Hy, sizeof(double)*create::index, cudaMemcpyDeviceToHost);
cudaMemcpy(&atom_h_Hz[0], d_Hz, sizeof(double)*create::index, cudaMemcpyDeviceToHost);
cudaMemcpy(&H_total[0], d_H_total, sizeof(double)*3, cudaMemcpyDeviceToHost);

// a synchronization point is needed
cudaThreadSynchronize();

// stop the clock
cudaEventRecord(stop_3);
cudaEventSynchronize(stop_3);

float elapsed_time2 =0;
//computing elapsed time
cudaEventElapsedTime(&elapsed_time2, start_3, stop_3);

// print the bandwidth of the device according to CUDA documentation
// more information can be found on following website: https://devblogs.nvidia.com/parallelforall/author/mharris/
```

```
    std::cout<<"The bandwidth is:"<<"\t"<<
    (sizeof(double)*(9*create::index+3))/(elapsed_time2*1e6)<<"\t"<<"GB/s"<<std::endl;

    // deallocate memory from device: coordinate & spins values & field values & total field
    cudaFree(d_x);
    cudaFree(d_y);
    cudaFree(d_z);
    cudaFree(d_sx);
    cudaFree(d_sy);
    cudaFree(d_sz);
    cudaFree(d_Hx);
    cudaFree(d_Hy);
    cudaFree(d_Hz);
    cudaFree(d_H_total);
    // print out results
    std::ofstream cuoutfile;
    // declaring a separate outputfile for cuda results
    cuoutfile.open("Curesults.data");
    // print the header of results file
    cuoutfile << "x"<<"\t"<<"y"<<"\t"<<"z"<<"\t"<<"Hz"<<std::endl;
    // loop over all sites and print the spin and coordinates values
    for (int i=0; i<create::index; i++){
        cuoutfile << atom_h_x[i]<<"\t"
                  << atom_h_y[i]<<"\t"
                  << atom_h_z[i]<<"\t"
                  << atom_h_Hz[i]<<std::endl;
    }// end of for loop
    // close the file
    cuoutfile.close();
    return 0;
}
```