Speedup of Micromagnetic Simulations with C++ AMP

On Graphics Processing Units

Ru Zhu

Graceland University, Lamoni, Iowa 50140 USA

Abstract

A finite-difference Micromagnetic solver, Grace, is presented utilizing the C++ Accelerated Massive Parallelism (C++ AMP). The high speed performance of a single Graphics Processing Unit (GPU) is demonstrated compared to a typical CPU-based solver. The speed-up of GPU to CPU is shown to be two orders of magnitude for problems with larger sizes. This solver is based on C++ AMP and can run on GPUs from various hardware vendors, such as NVIDIA, AMD and Intel, regardless of whether it is dedicated or integrated graphics processor. A preview version of the solver is available online.

Keywords: Micromagnetics; GPU; C++ AMP

1. Introduction

Micromagnetic simulations are important tools to study magnetic dynamics and design new magnetic devices. Micromagnetic solvers running on Central Processing Unit (CPU) such as OOMMF [1] and magpar [2] have been widely used in magnetism research. Micromagnetic simulations of complex magnetic structures require fine geometrical discretization, and are time consuming.

There has been growing research work on applying general purpose Graphics Processing Units (GPU) in the fields of Micromagnetics, such as MuMax, FastMag, GPMagnet and MicroMagnum [3] – [8]. Due to the high computing power of GPU units, these works have achieved considerable speed-ups as compared to previous CPU based implementations. On the other hand, general purpose GPU units are cheap, most costing less than $1000. Therefore complex micromagnetic simulations can be done at much lower cost.

However, these implementations are exclusively based on NVIDIA’s parallel computing platform Compute Unified Device Architecture (CUDA) and their applications are limited to NVIDIA GPUs. In 2012, Microsoft released its own parallel programming library named C++ AMP which is an open specification and is cross-hardware platform on Windows [9]. The most recent version of C++ AMP can also run on Linux [10]. Software based on C++ AMP can run on virtually all latest GPUs, including those from NVIDIA, AMD and Intel. The purpose of this work then is to implement a cross-hardware platform micromagnetic solver for solving the Landau-Lifshitz-Gilbert (LLG) equation.

1 Email address: zhu@graceland.edu.
Section 2 discusses the formulation of the magnetic dynamics and LLG equation, especially the formulas used to calculate the demagnetization field. Section 3 describes the software implementation of the micromagnetic solver. Section 4 presents the performance of this micromagnetic solver at various problem sizes, and compares it with a popular CPU-based micromagnetic solver. In this section the solver is also validated with \( \mu_{\text{mag}} \) standard problem 3 and 4.

### 2. Principle

Consider a magnetization vector \( \vec{M} = (M_x, M_y, M_z) \) in a computational cell that belongs to the computing region. Its saturation magnetization \( M_s = \sqrt{M_x^2 + M_y^2 + M_z^2} \). The magnetic energy density related to this vector can be written down as

\[
\varepsilon = A[(\nabla \frac{M_x}{M_s})^2 + (\nabla \frac{M_y}{M_s})^2 + (\nabla \frac{M_z}{M_s})^2] + K_u \frac{(M_y^2 + M_z^2)}{M_s^2}
- \frac{1}{2} (\mu_0 \vec{H}_{\text{demag}} \cdot \vec{M}) - (\mu_0 \vec{H}_{\text{extern}} \cdot \vec{M})
\]

(1)

The right hand side of (1) consists of the exchange, anisotropy, demagnetization and Zeeman energy densities, where \( A \) is the material exchange constant, \( K_u \) is the uniaxial anisotropy constant, \( \mu_0 \) is the vacuum permeability, \( H_{\text{demag}} \) is the demagnetization field and \( H_{\text{extern}} \) is the external field. The anisotropy energy is assumed to be uniaxial with an easy axis on the \( x \) direction.

The change of magnetization vector is caused by the effective magnetic field \( H_{\text{eff}} \) derived from the magnetic energy density:

\[
\vec{H}_{\text{eff}} = -\frac{\delta \varepsilon}{\delta \vec{M}} = \vec{H}_{\text{exch}} + \vec{H}_{\text{anis}} + \vec{H}_{\text{demag}} + \vec{H}_{\text{extern}}
\]

(2)

where \( \frac{\delta \varepsilon}{\delta \vec{M}} \) gives the functional derivative of \( \varepsilon \) with respect to \( \vec{M} \). In (2) the exchange field and anisotropy field are represented by \( \vec{H}_{\text{exch}} \) and \( \vec{H}_{\text{anis}} \) respectively.

According to (1) and (2), the \( x \) component of exchange field is

\[
H_{\text{exch},x} = \frac{2A}{\mu_0 M_s^2} \nabla^2 M_x.
\]

(3)

To derive the exchange field \( H_{\text{exch}} \) we need to discretize the computing region properly and consider the magnetizations of neighboring computational cells. The entire computing region is divided into \( n_x \times n_y \times n_z \) cells, each cell with an equal volume of \( \delta x \delta y \delta z \). The cells are labeled with indices.
$0 \leq i \leq n_x - 1,$
$0 \leq j \leq n_y - 1,$
$0 \leq k \leq n_z - 1.$

Note that $i, j$ and $k$ are zero-indexed to follow the convention of C++ programming language.

According to (3), the Cartesian components of the effective field can be expressed as [12]

$$
H_{\text{exchange},x} = \frac{-2A}{\mu_0 M_s^2} \left\{ M_z(i+1,j,k) - 2M_z(i,j,k) + M_z(i-1,j,k) \right\}
+ \frac{M_z(i,j+1,k) - 2M_z(i,j,k) + M_z(i,j-1,k)}{\delta_y^2}
+ \frac{M_z(i,j,k+1) - 2M_z(i,j,k) + M_z(i,j,k-1)}{\delta_z^2}.
$$

Other components of $\vec{H}_{\text{exchange}}$ can be obtained by replacing $x$ with $y$ or $z$ in (4).

According to (1) and (2),

$$
H_{\text{anis},x} = \frac{2K_u}{\mu_0 M_s^2} M_z.
$$

The LLG equation in the low damping limit is [11]

$$
\frac{d\vec{M}}{dt} = -\gamma \left( \vec{M} \times \mu_0 H_{\text{eff}} \right) - \frac{\alpha \gamma}{(1 + \alpha^2) M_s} \left[ \vec{M} \times (\vec{M} \times \mu_0 \vec{H}_{\text{eff}}) \right]
$$

where $\alpha$ is the damping constant, and $\gamma$ is the gyromagnetic ratio.

To speed up the micromagnetic simulation, it is necessary to decrease per-step simulation time, most of which is consumed by the calculation of the demagnetization field. The brute force calculation of demagnetization field is known to be proportional to the square of the number $N$ of the computational cells [13]. However, this calculation can be accelerated by taking advantage of the discrete convolution theorem and the fast Fourier transform (FFT) [14]. With the zero-padding method introduced later, non-periodic boundary conditions can be used.

Consider a rectangular magnetic sample can be divided into $n_x \times n_y \times n_z$ cells in three-dimensional space. After zero padding the input data size increases to $2n_x \times 2n_y \times 2n_z$, as demonstrated by Fig. 1. Since demagnetization field data after FFT contains complex numbers, there is an additional factor of two in the output size of FFT.
Fig. 1 A cross-sectional view of a rectangular magnetic sample after zero-padding.

In the case of a finite three-dimensional computing problem, the demagnetization field can be calculated as

\[ H_{\text{demag}}(i, j, k) = \sum_{l=0}^{n-1} \sum_{m=0}^{n-1} \sum_{n=0}^{n-1} M(l, m, n) K(l - i, m - j, n - k) \]  

(7)

or

\[ H_{\text{demag},x}(i, j, k) = \sum_{l=0}^{n-1} \sum_{m=0}^{n-1} \sum_{n=0}^{n-1} \left( M_x(l, m, n) K_{xx}(l - i, m - j, n - k) \right) \]

\[ \quad + M_y(l, m, n) K_{xy}(l - i, m - j, n - k) \]

\[ \quad + M_z(l, m, n) K_{xz}(l - i, m - j, n - k) \]  

(8)

where \( K \) is the demagnetization kernel matrix, of which the formulation can be found in [13]. Other components of \( H_{\text{demag}} \) can be obtained by permutation of indices in (8).

By applying DFT theorem to both sides of the equation, we can get

\[ \tilde{H}_{\text{demag},x}(i, j, k) = \tilde{M}_x(i, j, k) \cdot \tilde{K}_{xx}(i, j, k) \]

\[ \quad + \tilde{M}_y(i, j, k) \cdot \tilde{K}_{xy}(i, j, k) \]

\[ \quad + \tilde{M}_z(i, j, k) \cdot \tilde{K}_{xz}(i, j, k) \]  

(9)

Finally, the demagnetization field \( H_{\text{demag}} \) can be obtained by taking the inverse FFT of \( \tilde{H}_{\text{demag}} \),

\[ H_{\text{demag},x} = \text{FFT}^{-1}(\tilde{H}_{\text{demag},x}). \]  

(10)

Note that \( H_{\text{demag}} \) contains only real numbers because \( \tilde{H}_{\text{demag},x} \) is the dot product of FFTs of two real arrays.

The C++ AMP code of this key process is shown in Append. B.
3. Implementation

GPUs have intrinsically different hardware architecture from CPUs, notably for its large number of Arithmetic Logic Units (ALU) that was initially designed for graphics rendering but now also used for general purpose computing. Since GPU is specialized for computing-intensive, highly parallel computation, it is ideally suitable for micromagnetic simulations in which large number of computational cells can be processed in parallel. This idea is schematically illustrated by Fig. 2.

![Diagram of CPU and GPU hardware architecture](image)

The GPU has more ALUs dedicated to data processing. However, the amounts of cache and RAM are limited on GPU.

C++ AMP was implemented with High Level Shading Language (HLSL), which was initially designed for Microsoft’s graphics library DirectX [9]. Compared with popular GPU programming languages such as CUDA, it is fully cross-hardware platform, which means the programs written in C++ AMP can be migrated to another hardware vendor without any modification. Compared with other cross-hardware platform GPU programming language such as Open Computing Language (OpenCL), it has much simplified Application Programming Interface (API), thus reducing the programming effort of programmers. Readers can refer to Append. A for a comparison between APIs of OpenCL and C++ AMP.

GPUs usually have their own memory, also known as graphic memory. The data I/O is very fast between its ALUs and its own memory (> 100 GB/s), compared to the I/O between GPU and CPU (about 10 GB/s). Therefore the bottleneck to boost GPU computing performance is the data transfer from CPU to GPU or vice versa. In the micromagnetic solver presented, the only data transfer between CPU and GPU takes place when the initial conditions of the computing region are specified and when the final data is calculated by GPU and transferred back to CPU for display. In this way the simulation speed can be maximized.

As mentioned before, the most time-consuming part of micromagnetic simulation is the calculation of demagnetization field. In each time step, the calculation requires three different phases: (i) Perform FFTs to magnetization components; (ii) Do member-wise product of result of previous step and FFT of demagnetization tensors; (iii) Carry out inverse FFT of result of (ii).
The details are shown in Append. B. In three-dimensional space, there are six FFTs to perform for each time step. The FFTs of demagnetization tensor $K$ have been carried out at the beginning of simulation and will not be taken later, since $K$ is constant. The formulation of calculating $K$ was adopted from [13] which showed good accuracy in validation discussed later.

A FFT library based on C++ AMP has been implemented before [15]. It is adapted to the calculation of demagnetization field in the micromagnetic solver. At the point of publication the FFT library can only handle single-precision floats so this solver is currently limited to single-precision computing. The author expects the software to gain double precision calculation capability with the update of C++ AMP FFT library.

4. Results

To evaluate the speedup of simulation gained by GPU, The micromagnetic standard problem 3 [16] was used to test the performance of this solver. A cubic magnetic particle with exchange constant $A = 1 \times 10^{-11} J/m$, $M_s = 1000 kA/m$ and $H_{anis} = 100 kA/m$ was divided in to grids of $N \times N \times N$ and the minimum energy state is reached by applying the LLG equation to each computational cell. As shown in (5) the anisotropy field is maximized when magnetization is parallel or anti-parallel with the easy axis. The relaxation process involves the magnetization dynamics under the influence of demagnetization field, exchange field and uniaxial anisotropy field.

To benchmark the solver, two different GPUs were tested: an NVidia GTX 650 Ti and an AMD Radeon HD 7970 GHz Edition GPU. Both GPUs were both running on Intel Xeon E5410 CPU with 4GB of RAM. The AMD Radeon 7970 GPU is among the fastest on the consumer market but still cost less than $500. The NVidia GTX 650 Ti GPU is a middle-end product which cost less than $100. For comparison, the benchmark of MuMax [4] on GTX 650 Ti and CPU micromagnetic solver OOMMF is also presented, with data from the same hardware system which contains a quad-core Intel Xeon E5410 CPU. Dimensions with factors of two, five and ten were benchmarked to demonstrate the performance of solvers varying with problem size, as shown in table 1. However this magnetic solver can solve problems of any size limited by the graphic memory allocable by the GPU. It must be pointed out though, OOMMF simulation was done in double precision, but benchmark on GPUs was in single precision only. Furthermore, OOMMF has advanced ODE solvers than the Euler method benchmarked and also a conjugate gradient solver, which further reduces its simulation time.

It is noticeable that at smaller problem sizes (number of cells < 10,000) GPU solver is not significantly faster or even slower than CPU solver. This is caused by two factors. The first factor is that the data I/O overhead. The data transfer between GPU and CPU’s main memory takes time. For a smaller problem size the calculation on GPU can be completed very soon, so in this case the computing power of GPU will not be fully utilized. For larger problems the data I/O time can be negligible when compared to the computing time. The second factor is the kernel launching overhead of GPU. This overhead is a constant regardless of the problem size, thus it is significant when the problem size is small.
It can also be noticed in Fig. 3 and 4 that the per-step simulation times of all simulators show a stair-case like behavior with respect to input size. This is because the dominant part of micromagnetic simulation is the FFT calculation done by FFT libraries in which the calculation performance is optimized for power-of-two input sizes. For other input factors, the per-step time is significantly larger. In Fig. 4, it can be observed that the greatest speedup factors were achieved at non power-of-two numbers, in which case the FFT library on GPUs outperforms that of OOMMF.

For extremely large simulation sizes where number of cells is in the magnitude of millions, budget GPUs are limited by graphics memory and cannot support running of the program. This is reflected in the last two rows of Table 1.

Table 1. Per-step simulation time needed by CPU and GPU solvers for different 3D problem sizes with the Euler algorithm. Numbers are in milliseconds. Mumax version 3.6.2 Windows 64-bit was used in this benchmark. It can be observed that Grace outperforms Mumax for smaller input problem sizes but about 2x slower for large inputs.

<table>
<thead>
<tr>
<th>Size</th>
<th>CPU Xeon E5410</th>
<th>NVidia GTX 650 Ti (Mumax3)</th>
<th>NVidia GTX 650 Ti (Grace)</th>
<th>AMD Radeon 7970 (Grace)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 (8³)</td>
<td>3.42</td>
<td>1.09</td>
<td>0.733</td>
<td>0.677</td>
</tr>
<tr>
<td>4,096 (16³)</td>
<td>4.63</td>
<td>1.20</td>
<td>0.764</td>
<td>0.683</td>
</tr>
<tr>
<td>8,000 (20³)</td>
<td>9.53</td>
<td>1.87</td>
<td>1.94</td>
<td>1.02</td>
</tr>
<tr>
<td>15,625 (25³)</td>
<td>11.4</td>
<td>2.05</td>
<td>3.21</td>
<td>1.19</td>
</tr>
<tr>
<td>32,768 (32³)</td>
<td>14.6</td>
<td>2.46</td>
<td>3.14</td>
<td>1.24</td>
</tr>
<tr>
<td>64,000 (40³)</td>
<td>62.3</td>
<td>6.57</td>
<td>9.54</td>
<td>2.87</td>
</tr>
<tr>
<td>125,000 (50³)</td>
<td>79.1</td>
<td>12.7</td>
<td>22.3</td>
<td>4.94</td>
</tr>
<tr>
<td>262,144 (64³)</td>
<td>117</td>
<td>14.0</td>
<td>31.2</td>
<td>8.62</td>
</tr>
<tr>
<td>512,000 (80³)</td>
<td>778</td>
<td>40.5</td>
<td>71.1</td>
<td>18.1</td>
</tr>
<tr>
<td>100,000 (100³)</td>
<td>933</td>
<td>79.2</td>
<td>N/A</td>
<td>31.4</td>
</tr>
<tr>
<td>2,097,152 (128³)</td>
<td>1382</td>
<td>103</td>
<td>N/A</td>
<td>56.9</td>
</tr>
</tbody>
</table>
Fig. 3. Time need to carry out one time step at different 3D problem sizes. Note the stair-case shape behavior due to slow FFT calculations for non-power-of-two input sizes.

Fig. 4. Speed-up of GPU solver on an AMD Radeon HD 7970 GHz Edition compared to CPU solver OOMMF. The speed-up increases with problem size.
To validate the simulation result, the cubic particle was discretized to $10 \times 10 \times 10$ cells, and the transition point from flower state to vortex state was found to be near $l = 8.47l_{ex}$, where $l$ is the edge length of the cube and $l_{ex} = \sqrt{\frac{2A}{\mu_0M_s^2}}$ is the intrinsic length scale. The magnetizations of the cubic particle before and after the transition were shown in Fig. 5 and 6.

![Diagram](image)

**Fig. 5** In μmag standard problem 3, the magnetization in cubic particle in flower state at $l = 8.47l_{ex}$. 


Fig. 6 In $\mu$mag standard problem 3, the magnetization in cubic particle in vortex state at $l = 8.6l_{\text{ex}}$.

The $\mu$mag standard problem 4 [17] was also used to validate the result of calculation. In this problem a rectangular film sample was divided in to 500×125×3 cells, with a mesh size of 1nm × 1nm × 1nm. The exchange constant of the sample was set as $1.3\times10^{-11} J/m$, and the saturation magnetization was $8.00\times10^5 A/m$. There was no anisotropy present. The system was relaxed to S-state by setting a large damping constant before a switching field 1 of (-24.6 mT, 4.3 mT, 0 mT) was applied. During the switching the damping constant $\alpha$ is set to 0.02. The input file is as follows. Note that the magnetization and switching field unit has been converted to $kA/m$.

```
-simulation 10000 1000000 5e-7
# writeInterval timesteps dt
-rectang 1000 250 6
# nx ny nz
-material 0.02 1.3e-11 800 0 0 0 0 0 0
# alpha A (J/m) M_init.x M_init.y M_init.z Hkx Hky Hkz
-readInitState
-externfield -19.576 3.422 0. 0 3000000 3000000
# x y z startTime decayTime stopTime
```
Fig. 7. Average magnetization versus time during the reversal in μmag standard problem 4, field 1. OOMMF simulation results are also presented for comparison.

Fig. 8. Magnetization distribution when $M_x$ first crosses zero in μmag standard problem 4, field 1. The domain wall can be clearly seen at left 1/3 and 2/3 of the sample.

According to Fig. 7 and 8 the average magnetization results and the magnetization distribution from Grace is in good agreement with that from OOMMF [17]. The result is thus reliable.
5. Summary

A GPU-based micromagnetic solver is presented to address the slow speed problem of large simulation problems. The speed boost relative to CPU simulations is significant at problem with large input sizes. This solver can not only run on expensive professional workstations but also economy personal laptops and both achieve considerable speed-ups.

In the future, the solver is expected to fully explore the potential of GPU computing by utilizing the shared memory and reduce the kernel launching overhead by reducing the number of kernel function calls. This solver can be downloaded from https://github.com/cygnusc/grace.

Acknowledgements

This work is supported by Graceland University professional development program. The author wishes to acknowledge Dr. Pieter B. Visscher of the University of Alabama, for helpful discussions of demagnetization field calculation.

References


Append. A. Comparison between APIs of OpenCL and C++ AMP.

The two code snippets shown calculate the matrix product of global_A and global_B. It can be seen that C++ AMP has a simplified API as compared to OpenCL which benefits the development process.

OpenCL

```c
cl_int err;
cl_context context = clCreateContextFromType(NULL, CL_DEVICE_TYPE_GPU,
NULL, NULL, &err);
size_t devicesSize;
err = clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &
devicesSize);
cl_device_id *devices = (cl_device_id*) malloc(devicesSize);
err = clGetContextInfo(context, CL_CONTEXT_DEVICES, devicesSize,
devices, NULL);
cl_command_queue commandQueue = clCreateCommandQueue(context,
devices[0], 0, &err);
cl_program program = clCreateProgramWithSource(context, 1, &source,
NULL, &err);
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "matrixmult", &err);

cl_mem global_A = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR,
size*sizeof(float), (void*) A, &err);
cl_mem global_B = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR,
size*sizeof(float), (void*) B, &err);
cl_mem global_C = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
size*sizeof(float), NULL, &err);

err = clSetKernelArg(kernel, 0, sizeof(int), (void*)&size);
err = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*)&global_A);
err = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void*)&global_B);
err = clSetKernelArg(kernel, 3, sizeof(cl_mem), (void*)&global_C);

size_t localWorkSize[] = {32, 32};
size_t globalWorkSize[] = {size, size};
err = clEnqueueNDRangeKernel(commandQueue, kernel, 2, NULL,
globalWorkSize, localWorkSize, 0, NULL, NULL);

__kernel __attribute__((reqd_work_group_dim(32, 32, 1)))
void matrix_mult(int dim, const __global float *global_A, const
__global float *global_B, __global float *global_C)
{
    int rowID = get_local_id(1);
    int colID = get_local_id(0);
    __local float A[32][32];
    __local float B[32][32];
    float sum = 0.0f;
    for (int i = 0; i < dim; i += 32)
    {
        ...
    }
```
{  
  A[rowID][colID] = global_A[get_global_id(1) * dim + i + colID];  
  B[rowID][colID] = global_B[(i + rowID) * dim + get_global_id(0)];  
  barrier(CLK_GLOBAL_MEM_FENCE);  
  for (int k = 0; k < 32; k++)  
  {  
    sum += A[rowID][k] * B[k][colID];  
  }  
  barrier(CLK_GLOBAL_MEM_FENCE);  
  global_C[get_global_id(1) * dim + get_global_id(0)] = sum;  
}

C++ AMP

array_view<const float, 2> global_A(size, size, A);  
array_view<const float, 2> global_B(size, size, B);  
array_view<float, 2> d_C(size, size, C);  
global_C.discard_data();

parallel_for_each(global_C.extent, [=] (tiled_index<32, 32> t_idx)  
restrict(amp)  
{
  int row = t_idx.local[0];  
  int col = t_idx.local[1];  
  tile_static float A[32][32];  
  tile_static float B[32][32];  
  float sum = 0.0f;  
  for (int i = 0; i < size; i += 32)  
  {  
    A[row][col] = global_A(t_idx.global[0], i + col);  
    B[row][col] = global_B(i + row, t_idx.global[1]);  
    t_idx.barrier. wait();  
    for (int k = 0; k < 32; k++)  
    {  
      sum += A[row][k] * B[k][col];  
    }  
    t_idx.barrier. wait();  
  }  
  global_C[t_idx.global] = sum;  
});
Append. B. C++ AMP code to evaluate the demagnetization field. The x-component of demagnetization field ($H_x$) calculation is shown. Other components can be calculated by shifting the index. Meanwhile, $[\text{idx}].x$ represents the real part of the FFT and $[\text{idx}].y$ represents the imaginary part.

```cpp
transform.forward_transform(Mx,
reinterpret_cast<array<std::complex<float>, dims>&> (Mx_fft));

parallel_for_each (Hx_fft.extent, [&] (index<3> idx) restrict(amp)
{ /* element wise product: $H_x = K_{xx}M_x + K_{xy}M_y + K_{xz}M_z$

$(a + bi) \cdot (c + di) = (a \cdot c - b \cdot d) + (a \cdot d + c \cdot b)i$ */

Hx_fft[idx].x = Mx_fft[idx].x * Kxx_fft[idx].x - Mx_fft[idx].y *
Kxx_fft[idx].y + My_fft[idx].x * Kxy_fft[idx].x - My_fft[idx].y *
Kxy_fft[idx].y + Mz_fft[idx].x * Kxz_fft[idx].x - Mz_fft[idx].y *
Kxz_fft[idx].y;

Hx_fft[idx].y = Mx_fft[idx].x * Kxx_fft[idx].y +
Mx_fft[idx].y * Kxx_fft[idx].x + My_fft[idx].x * Kxy_fft[idx].y +
My_fft[idx].y * Kxy_fft[idx].x + Mz_fft[idx].x * Kxz_fft[idx].y +
Mz_fft[idx].y * Kxz_fft[idx].x;
});

transform.inverse_transform(reinterpret_cast<array<std::complex<float>, dims>&> (Hx_fft), Hx);
```