Abstract

The global distribution and climatology of ice clouds are among the main uncertainties in climate modelling and prediction. In order to retrieve ice cloud properties from remote sensing measurements, the scattering properties of all cloud ice particle types must be known. The Discrete Dipole Approximation (DDA) simulates scattering of radiation by arbitrarily shaped particles and is thus suitable for cloud ice crystals. The DDA models the particle as a collection of equal dipoles on a lattice, and is computationally much more expensive than approximations restricted to more regularly shaped particles. On a single computer the calculation for an ice particle of a specific size, for a given scattering plane at one specific wavelength can take several days. We have ported the core routines of the scattering suite “Amsterdam DDA” (ADDA) to the Open Computing Language (OpenCL), a framework for programming parallel devices like PC graphics cards (Graphics Processing Units, GPUs) or multicore CPUs. In a typical case we can achieve a speed-up on a GPU as compared to a CPU by a factor of 5 in double precision and a factor of 15 in single precision. Spreading the work load over multiple GPUs will allow calculating the scattering properties even of large cloud ice particles.

Keywords: Discrete Dipole Approximation, OpenCL, GPU

1. Introduction

1.1. Single Scattering at Cloud Ice

The global distribution and climatology of ice clouds represent one of the main uncertainties in climate analysis and prediction [1]. For their daily global observation, spaceborne microwave millimeter and submillimeter radiometers have been suggested [2] because of their ability to sense the interior of cloud. Many algorithms to retrieve geophysical properties of ice clouds assume a spherical shape of the single crystals when calculating their scattering properties. While this simplifying assumption is appropriate in the Rayleigh regime, i.e., for particles small compared to the wavelength (> 3 mm or $\nu < 100$ GHz), for higher frequencies the aspherical shape of the ice crystals more and more determines their scattering properties which in turn are required for the forward radiative transfer modeling of ice clouds. The main properties of the scattered radiation depend on the single scattering properties of every involved particle. There are no sufficient possibilities to measure directly the distributions of size, shape and orientation of the ice crystals in ice clouds and convective systems. However, a possible approach is to build a database of the scattering properties of different shapes and sizes of ice crystals based on forward scattering calculations [3]. There are several different methods to calculate the single scattering properties of ice crystals.

The T-Matrix method [3] is accurate for calculating scattering properties of spheroidal particles. The finite difference time domain (FDTD) method [5] is computationally expensive on particle sizes which exceed twenty times the wavelength. However, the most flexible method is the discrete dipole approximation (DDA) method [6] suited for arbitrary shapes. It is an accurate method to solve the Maxwell equations for single particle scattering. But DDA is computationally expensive and has a high memory consumption especially for larger particles. The field of application of FDTD and DDA is similar but DDA is better suited for small refractive indices [7]. In this study we use the Amsterdam DDA (ADDA) program package [8] because it is writ-
ten in the C programming language where an interface to OpenCL is provided.

1.2. Graphic Processing Units (GPUs) for Numerical Applications

In view of the growing number of multicore central processing units (CPUs), the growing numbers of cores per CPU and the reduction of clock rate it seems to be the right time to rethink the algorithmic approaches which have been developed for serial computers. A well suited application for parallel computers are for example linear filters applied to large datasets like images. That kind of task is nowadays often done by graphic processing units (GPUs) with up to 1600 processing cores, in contrast to CPUs with up to 6 cores. On the other hand the instruction set of a CPU is a lot bigger than that of a GPU, i.e., a GPU can perform more complex mathematical operations in fewer cycles than a CPU. As a consequence, GPUs appear attractive to apply mathematically simple function on large datasets, as it is the case for DDAA where a system of linear equations is solved.

A unified standard framework for programming multicore CPUs, GPUs and also other parallel computing devices is the Open Computing Language (OpenCL) by the Khronos group in December 2008 [3]. The GPU vendor Nvidia released an example with their Software Development Kit (SDK) for OpenCL which does an DDAA calculation [10]. In this report, the DDAA method, as realized in the DDAA software package, is analysed for the possibility to implement it in OpenCL, and the possible speedup over the CPU implementation will be estimated.

2. Theory of Discrete Dipole Approximation (DDA)

2.1. Overview

The DDAA approach for calculating single scattering properties of particles approximates the shape of an ice crystal by a number of small dipoles on a cubic lattice. An incident wave is scattered by all dipoles. Every dipole will scatter the incident wave into all directions, among them the directions of all other dipoles so that not only the incident wave arrives at one specific dipole but also the scattered waves from all other dipoles. Therefore the self consistent solution for the dipole moments must satisfy the \( 3N \) linear equations, with \( N \) as the number of dipoles,

\[
\sum_{j=0}^{N} A_{ij} P_j = E_{inc,i}, \quad i = 1, \ldots, N. \tag{1}
\]

\( A \) is the interaction matrix of the dipoles and \( E_{inc} \) is the incident wave. \( P \) is the solution vector to find. For an anisotropic medium the matrix elements \( A_{ij} \) are in turn \( 3 \times 3 \) matrices. The diagonal elements are \( A_{ii} = \alpha_{ii}^{-1} \) where \( \alpha_i \) is the complex polarizability tensor of the \( i \)th dipole. For isotropic media this tensor can be treated as scalar. The off-diagonal elements depend only on the wave number \( k = 2\pi v/c \), and on \( r_{ij} = x_i - x_j \) the vector displacement, where \( \omega \) is the angular frequency and \( c \) is the speed of light. \( x_i \) and \( x_j \) are the 3-D vectors describing the position of the \( i \)th and the \( j \)th dipole, respectively. If a lattice spacing \( d \) is chosen, there still remains the choice of the polarizabilities of the dipoles expressed by the polarizability tensor \( \alpha \). For \( \lim d/\lambda \rightarrow 0 \) the Clausius-Mossotti relation can be applied. It describes the relationship between macroscopic permittivity and microscopic polarizability. For our case, a finite \( d/\lambda \), this approximation needs some corrections. The goal is to represent a continuous material of a permittivity \( \varepsilon \). To achieve this goal one can choose the dipole polarizability in such a way that an infinite lattice of those polarizabilities will have the same dispersion relation \( k(\omega) \) as the continuum material. This approach will include \( O(kd^2) \) and \( O(kd^4) \) corrections to the Clausius-Mossotti relation [11]. Due to its lattice character it is called lattice dispersion relation (LDR).

Once Eq. (1) is solved, one can calculate the extinction and absorption cross sections [12] \( C_{ext} \) and \( C_{abs} \) as

\[
C_{ext} = \frac{4\pi k}{|E_0|^2} \sum_{j=0}^{N} \Im \left( E_{inc,j}^* \cdot |P_j| \right) \tag{2}
\]

\[
C_{abs} = \frac{4\pi k}{|E_0|^2} \sum_{j=0}^{N} \left| \Im \left( P_j \cdot \left( \alpha_j^{-1} \right) |P_j^*| - \frac{2}{3} k^3 |P_j|^3 \right) \right| \tag{3}
\]

with \( \Im \) imaginary part and \( E_0 \) the amplitude of the incident wave. Crystals of complex shape of high aspect ratios require a lot of dipoles for their accurate representation which will lead to huge matrices \( A \). But even for simple crystals the system is too huge to be stored in memory of a personal computer, e.g., for a \( 100 \times 100 \times 100 \) dipole representation of a crystal, \( A \) is a \( 10^6 \times 10^6 \) matrix.

2.2. System of Linear Equations in DDA

The most common direct method to solve a system of linear equations, the LU-decomposition, has \( O(n^3) \) computation time and \( O(n^2) \) storage requirements. It expresses the matrix as a product of one upper triangle matrix and one lower triangle matrix. The system of linear equations can then be solved directly through forward and backward substitution without using the Gaussian elimination method. The LU-decomposition has
the advantage that the resulting matrices can be reused if other similar systems have to be solved. Otherwise it is computationally about the same as the Gaussian elimination method. The complete matrix holds \( 9N^2 \) complex numbers, since every \( A_i \) is a \( 3 \times 3 \) matrix. This means in double precision \( \mathbb{D} \), this matrix needs \( 9N^2 \cdot 16 \) bytes in memory (half in single precision \( \mathbb{S} \)), so for a system of 5000 unknowns this is about 3.6 Gigabyte and a system with 5 million unknowns would take 3.6 Petabyte. Therefore it is necessary to exploit symmetries and use iterative methods avoiding to store the whole matrix in memory. Krylov-Space methods, such as Conjugate Gradient \( \mathbb{CG} \)-variants and quasi-minimal residual \( \mathbb{QMR} \) are suited to solve the system of linear equations of the \( \mathbb{DDA} \). All Krylov-space methods just need to access the matrix by matrix vector multiplication which is a \( O(N^2) \) operation.

2.3. Toeplitz and Circulant Matrices

If the dipoles are located on a periodic lattice, the matrix structure is suitable to do matrix vector multiplications as convolutions since the matrix has Block-Toeplitz-symmetry \( \mathbb{B} \). With \( t = 2n - 1 \) as the number of different values in the \( n \times n \) Toeplitz matrix, it looks like

\[
\begin{pmatrix}
a_n & a_{n+1} & \ldots & a_{t-1} & a_t \\
a_{n-1} & a_n & \ldots & a_{t-2} & a_{t-1} \\
\vdots & a_{n-1} & a_n & \ldots & a_{t-3} \\
a_2 & \ldots & \ldots & a_{n-1} & a_n \\
a_1 & a_2 & \ldots & a_{n-2} & a_{n-1} \\
\end{pmatrix}
\]

The result is the circulant matrix

\[
C = \begin{pmatrix}
0 & a_1 & \ldots & a_{n-2} & a_{n-1} \\
a_t & 0 & a_1 & \ldots & a_{n-2} \\
\vdots & a_t & 0 & \ldots & \vdots \\
a_{n+2} & \ldots & \ldots & a_1 \\
a_{n+1} & a_{n+2} & \ldots & a_t & 0
\end{pmatrix}
\]

Figure 1: Number of operations for multiplication of a circulant matrix with a vector

so that not all points of the lattice are occupied by a dipole. That fact ruins just the diagonal of the Toeplitz matrix, since the non diagonal elements does not depend on dipole polarizabilities \( \mathbb{M} \) and the polarizability for a lattice point used by a dipole is obviously different from an empty lattice point. To be able to still use the \( \mathbb{DDA} \)-based matrix vector multiplication, one can set the diagonal elements equal to zero and do the \( \mathbb{FFT} \) convolution while the diagonal term can be calculated independently and added later \( \mathbb{DDA} \). This principle is used by all current \( \mathbb{DDA} \)-scattering programs.

2.4. Green’s Function

In \( \mathbb{DDA} \) the dipoles of the scatterers are localized on a discrete lattice on a constant background for which \( \mathbb{ADDA} \) uses Green’s functions \( \mathbb{G} \). This allows to just
calculate the non-void dipoles while it cares about the background. The total electric field \( E \) at position \( r \) is given by

\[
E(r) = E^0(r) + \int_V dV' G(r, r') \cdot k_0^2 \Delta \varepsilon(r) E(r'),
\]

where \( E \) is the electric field of the incident wave, \( V \) is the volume of the grid of the scatterer, \( G \) the dyadic Green’s function, \( k_0 \) the vacuum wave number, \( \varepsilon \) the dielectric tensor. \( \Delta \varepsilon(r) = \varepsilon(r) - \varepsilon_{0} = 0 \) for empty grid points. The resulting field is completely determined by the field inside the scatterer. This also applies to any point of the background. The definition of the dyadic Green’s function is in our case

\[
\nabla \times \nabla \times G(r, r') - k_0^2 \varepsilon(r) G(r, r') = \delta(r - r'),
\]

where \( \mathbf{l} \) is the unit dyad.

2.5. Polarized Light, Stokes Vector and Mueller Matrix

Polarized and partially coherent light is commonly described by the four-element Stokes vector. The Mueller matrix \( M \) transforms the incident Stokes vector into the outgoing Stokes vector \([13]\). ADDA calculates the Mueller matrices for the whole \((y, z)\) scattering plane. The scattering angle in this plane is \( \theta \). The element \( M_{11} \) describes the ratio of incident and outgoing photons. The degree of polarization of the outgoing light is given by

\[
P = \frac{\sqrt{|M_{31}|^2 + |M_{32}|^2 + |M_{41}|^2}}{M_{11}}.
\]

2.6. Preconditioning and Iterative Solvers

An important point in the DDA method is the right preconditioning, i.e., transforming the initial matrix in order to make it more sparse, resulting in faster matrix inversion due to reduced number of iterations. Preconditioning can be written as

\[
Q_1 A Q_2^{-1} (Q_2 x) = Q_1 y.
\]

\( Q_1 \) and \( Q_2 \), are the left and right preconditioners, respectively. The choice of an appropriate preconditioner depends on the method used as well as on the kind of system to solve. The simplest preconditioner is the Jacobi preconditioner which is just a diagonal matrix. Not only an appropriate preconditioner but also the iterative method is important to obtain reasonable calculation times. Flatau \([14]\) reviewed the different methods and recommended the Biconjugate Gradient Stabilized (BCGSTAB) method with left Jacobi preconditioner, but he did not give the number of dipoles of the scatterer on which the iterative methods were tested. Yurkin and Hoekstra, the authors of ADDA, pointed out that quasi-minimal residual (QMR), Biconjugate Gradient (BCG) and Biconjugate Gradient Stabilized (BCGSTAB) are efficient methods for solving the DDA task. For convergence reasons for some crystal shapes ADDA uses by default the QMR method but also offers methods like BCGSTAB \([12]\). The convergence depends also on the precision used for storing the matrix elements \([8]\). Currently SP and SI calculations take about the same time on consumer \([11]\). The first GPU, which can do DP computations, but only at one twelfth speed, were introduced in mid 2008. Since on modern GPUs, the computation time for DP calculation become less different from SI calculations, we focus on DP calculations in this study. We also discuss a SI implementation and compare the performance and limitations of both implementations.

3. Overview over Open Computing Language (OpenCL)

OpenCL is the name of a parallelization software framework. It is an Application Programming Interface (API) and a programming language at the same time. It is used to distribute computational work load over several platforms and devices. The best value of computational power per money and computational power per power consumption available nowadays are high end gaming GPUs which are basically highly parallel computation units. OpenCL offers a way to program such a GPU. The OpenCL language is similar to the programming language C and provides interfaces to C and C++. There are also some simplified interfaces to high level languages like Python.

3.1. Comparison of Parallelization Frameworks

There are several concepts to use parallel processing power. Message Passing Interface (MPI) \([17]\) focuses on network distributed computing on multiple computers, Open Multiple-Processing (OpenMP) \([16]\) focuses on multiple CPU cores, Compute Unified Device Architecture (CUDA) \([15]\) is a proprietary implementation of the vendor Nvidia for programming their GPUs, and OpenCL \([12]\) similar to CUDA but capable to use all computing devices with several cores in one computer like multicore CPUs, GPUs and physics processing units (PPUs). We decided to use OpenCL since it is not restricted to a specific architecture.
3.2. Graphics Processing Units (GPUs) and Central Processing Units (CPUs) Architecture for Data Processing

The CPU reads data from the host memory into its cache, does processing and writes the result back to host memory. A GPU works similarly. It transfers some data from host memory into the device memory, from device memory into registers, then processes the data, writes it back to the device memory and then back to host memory. The main difference between GPUs and CPUs is that the GPUs are designed for data parallelism which is also called single instruction multiple data (SIMD). On CPUs, this is achieved by many cores working on the data in parallel. OpenCL allows the programmer to manage all different memory types like global device memory, local work group memory, and registers of the GPU. This is needed to get the best performance out of a GPU. But one can also achieve a speedup by just converting the SIMD task to OpenCL and have them calculated on a GPU. Even simple operations like memory copy or zeroing array elements can thus be accelerated because of the high speed of the GPU memory.

Some properties for computations on recent CPUs and GPUs relevant here are shown in Table 1. The highest values of the ranges given in the GPU column are not realized in one device, e.g., there is recently no GPU which achieves 2700 million floating point operations per second (FLOPS) in SP calculations and has 6 GB of Memory. The maximum memory on a GPU is usually a bit lower for consumer GPUs. The upper values of peak performances exclude double GPU on one board solutions since they are handled as two different devices if the device drivers support to use both GPUs.

Table 1: Typical properties of CPUs and GPUs which are relevant for computations

<table>
<thead>
<tr>
<th>Property</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of cores</td>
<td>up to 6</td>
<td>≤ 1600</td>
</tr>
<tr>
<td>memory speed [GB/s]</td>
<td>5-10</td>
<td>100-180</td>
</tr>
<tr>
<td>mem to host [GB/s]</td>
<td>5-10</td>
<td>3-5</td>
</tr>
<tr>
<td>peak DP [GFLOPS]</td>
<td>20 - 100</td>
<td>40-650</td>
</tr>
<tr>
<td>peak SP [GFLOPS]</td>
<td>20 - 100</td>
<td>800-2700</td>
</tr>
<tr>
<td>max memory [GB]</td>
<td>4-48</td>
<td>1-6</td>
</tr>
</tbody>
</table>

3.3. OpenCL Kernels

A kernel in OpenCL is a function which has to be executed many times with slightly different input like different indices. In principle a kernel can be seen as a loop (even with multiple loop variables) over a C function and can be used as this. Inside an OpenCL kernel there is no access to the host memory, so all data needed must be copied to the device memory first. The memory transfer from host to device and vice versa is slow and should be avoided whenever possible. This is the main bottleneck in GPU computations nowadays.

3.4. OpenCL Memory Model

Understanding the memory architecture of OpenCL is important in order to obtain reasonable performance. Figure 2 is a rough sketch of the memory model in OpenCL where the dots denote that there may more units of this type. A workitem is executed by one core. Workitems can be clustered into workgroups. The Workitems in one workgroup can share data through the local memory. Splitting a problem into workgroups can be useful, e.g., if the data points cannot be calculated independently or to use faster memory. One obvious rule for good performance is to use the fastest memory for the variables which have to be accessed most frequently. The memory latency relation of a GPU in OpenCL can be expressed as

\[
global\ memory \gg local\ memory \gg registers. \quad (11)
\]

So it is useful to copy data on which many operations have to be performed from global to private memory. The best way to copy data is by coalesced access. That means to let the cores read or write the global memory aligned, because this can be done simultaneously. Otherwise the cores have to wait to get access to the memory. Sometimes this is called “memory scattering”.

The local Memory is organized in banks. Banks can be accessed simultaneously as long as only one core has
to access the same bank. If more cores have to access the same bank there will be a bank conflict and the access would be serialized. Bank conflicts and unaligned access to global memory are the main reason for not getting the peak performance out of the device in practical applications.

3.5. Latency Hiding and Occupancy

A general technique to bypass the waiting time of the cores for memory access is called latency hiding. To hide latency one needs to keep cores busy by arithmetic operations without the need to access the corresponding memory. Of course this is not possible when we use OpenCL for simple operations, e.g., setting all array elements to zero. Due to the limited size of the local memory and the register memory, the GPU trades memory for occupancy. That means, that every core and work group which is busy by a kernel uses exactly the same amount of memory. If more memory is needed than is available the GPU will send cores and workgroups to idle to use their memory for the other ones. This will lower the occupancy of the GPU. This simplified explanation of the OpenCL architecture on GPUs will help to understand the details of our implementation of the matrix vector multiplication of ADDA. A more detailed overview of memory scattering, latency hiding and occupancy on Nvidia GPUs is given in the Nvidia “OpenCL Optimizations” presentation [18].

3.6. Native Functions and Double Precision Calculations

GPUs are designed to handle single precision floats, since no higher precision is needed for their original purpose. Simple mathematical functions, e.g., sin and exp, can be executed natively by hardware if called with a native in front. This allows the GPU to perform such functions about 15 times faster. In OpenCL, DP is not supported by default but by an extension, since most current GPUs support simple DP calculations in about one third to one twelfth of the speed of SX calculations. Native functions like the ones for SX floats are still not available. ADDA is completely written in DX, therefore performance will suffer for every used non-native function on DX floating point variables. This disadvantage is negligible if there are not too many arithmetic operations in a kernel so that latency on memory access is also a limiting factor (Section 3.3). There is also the possibility to use some operations optimized for speed with a little cost in accuracy. As an example we discuss the mad-function which calculates the product of two variables and adds a third one to the result. The result is not as accurate as a*b+c but it is done in fewer cycles, depending on the GPU. This function can be used wherever speed is more important than precision. Some variants of mad-functions are used inside the Apple FFT library discussed in Section 3.2. On the other hand there is the fused multiplication addition (FMA) function which calculates a*b+c too, but does not round after the multiplication but after the addition. Therefore the result can be more accurate than a*b+c.

3.7. Optimization Utilities

To obtain information about how many cycles the GPU needs for a specific operation, one can use the ATI Stream SDK and the Stream Kernel Analyzer [19], which converts the kernel into assembly code of AMD/ATI GPUs. The fewer lines this code has, the fewer cycles are needed for the specific operation. To look into the assembler code is more important than is done in this study) has native support for DP floating point variables. There is a trade-off between the hardware support of different GPU vendors, especially for different GPU architectures of Nvidia called “Fermi” (not used in this study) has native support for DP floating point variables. Another restriction is that no variable sized arrays are supported in OpenCL, so there are no malloc and free commands (like in C) in OpenCL kernel files, so the GPUs have no memory heap. Pointers to functions and recursions are also not allowed. Due to different architectures of GPUs, especially for GPUs from different vendors, there cannot be a fastest version of one kernel for all GPUs. But there is an elegant workaround to the variable sized arrays and the architecture issue. The OpenCL kernel files are usually compiled at runtime of the program, so that they do not have to be

\footnote{The effect and also the definition of a bank conflict and of unaligned memory access varies a lot with the GPU vendor and even model revision due to different bank sizes and allowed word length.}

\footnote{The difference between the native and emulated functions depend on hardware, e.g., a factor of 11 for the sin-function on a AMD Radeon 5870}

\footnote{The documentation on this is very sparse, so we could not figure out if DX native functions are a current limitation of OpenCL so that DX native functions would also not be available on the latest GPUs of Nvidia which have full DX support.}
constant at compile time. That allows to write a program which optimizes the kernel file during runtime depending on the GPU architecture or the different tasks. The collection of kernels does not even have to be in a file, a concatenation of strings is also possible to compile as kernel. This programming paradigm is known as meta-programming. It is not limited to the usage with OpenCL. There are also some linear algebra libraries using meta-programming like Automatically Tuned Linear Algebra Software (ATLAS) [21] or GPU ATLAS (GATLAS) [22]. These automatically tuned approaches are usually faster than static implementations. Here we use a modified OpenCL FFT example from Apple which employs meta-programming. It automatically tunes OpenCL kernels to generate a plan for every used FFT size before executing it.

4. Analysis and Timing of Amsterdam DDA

4.1. Used Hardware

All timing results were obtained on ordinary consumer computer hardware. The relevant details are listed in Table 2. The CPU is a dual core CPU, but for the timing of ADA just one core is used since we compiled ADA to sequential mode.

Table 2: Used Hardware

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>AMD Athlon II x2 240e</td>
<td>clockspeed</td>
<td>2.800 Mhz</td>
</tr>
<tr>
<td>RAM</td>
<td>4096 MB</td>
<td>GPU</td>
<td>Nvidia GeForce GTX 260 (216 Cores)</td>
</tr>
<tr>
<td>GPU memory</td>
<td>896 MB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Nvidia GeForce GTX 260 used in this study is a consumer gaming graphics card. Its fast [A] capabilities (about 800 Giga [FLOPS]) makes it well suited for scientific calculations. However, the [DP] performance is just about 65 Giga [FLOPS]. It was one of the first devices that can handle [DP] calculations. It was about 150 Euro in late 2009 and is not available anymore. Its successor model is the Nvidia GeForce GTX 460 [GPU].

4.2. Detection of Time Consuming Operations

In order to establish representative performance characteristics, a number of different shapes were used to figure out the typical time consumption of the different subroutines of ADA. The chosen scatterer properties and ADA calls are shown in Table 3.

Table 3: Timing of shapes

<table>
<thead>
<tr>
<th>shape</th>
<th>(n_D)</th>
<th>(N[k])</th>
<th>(I)</th>
<th>t[s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>cylinder</td>
<td>10</td>
<td>1.78</td>
<td>107.4</td>
<td>1892</td>
</tr>
<tr>
<td>coated1</td>
<td>20</td>
<td>1.31, 1.78</td>
<td>102.2</td>
<td>129</td>
</tr>
<tr>
<td>coated2</td>
<td>40</td>
<td>1.2, 1.05</td>
<td>248.9</td>
<td>83</td>
</tr>
<tr>
<td>sphere1</td>
<td>60</td>
<td>1.05</td>
<td>556.0</td>
<td>15</td>
</tr>
<tr>
<td>sphere2</td>
<td>100</td>
<td>1.05</td>
<td>2482.9</td>
<td>24</td>
</tr>
</tbody>
</table>

with \(n_D\) as number of dipoles per wavelength along \(x\) axis and \(n_x\) as the total number of dipoles along the \(x\) axis. As ADA is written in C and the code is open source, the first step is to compile ADA for profiling with the -pg compiler option of GNU Compiler Collection (GCC) which provides timing information and the number of function calls. The most time-consuming functions and the number of calls are shown in Table 4. The MatVec function consumes a lot of runtime. The calls of the Fourier transform do not appear separately since ADA uses an external library called Fastest Fourier Transform in the West (FFTW) [23] for the Fourier transformation which was not compiled for profiling. The times for the FFT are included in the times of MatVec. The routine MatVec corresponds to the matrix-vector multiplication, explained in Section 2.2 and is the most time consuming part of the algorithm. It takes about 70-90% of the calculation time.

Table 4: Time consumption and number of calls of ADA routines

<table>
<thead>
<tr>
<th>shape</th>
<th>% MatVec</th>
<th>MatVec time</th>
<th>MatVec calls</th>
<th>other routines</th>
<th>other routines time</th>
</tr>
</thead>
<tbody>
<tr>
<td>cylinder</td>
<td>90.91</td>
<td>313.83</td>
<td>1893</td>
<td>9.09</td>
<td>31.38</td>
</tr>
<tr>
<td>coated1</td>
<td>88.77</td>
<td>33.20</td>
<td>130</td>
<td>11.23</td>
<td>4.20</td>
</tr>
<tr>
<td>coated2</td>
<td>85.75</td>
<td>52.27</td>
<td>84</td>
<td>14.25</td>
<td>8.69</td>
</tr>
<tr>
<td>sphere1</td>
<td>64.45</td>
<td>25.15</td>
<td>16</td>
<td>35.55</td>
<td>13.87</td>
</tr>
<tr>
<td>sphere2</td>
<td>73.01</td>
<td>178.30</td>
<td>25</td>
<td>26.99</td>
<td>65.91</td>
</tr>
</tbody>
</table>

4.3. Implementation of the Matrix-Vector Multiplication

The dipole matrix is Toeplitz for every dimension (Section 2.3). ADA uses this feature to keep the
matrix with the dipole information in this natural 3-dimensional structure as a rank 3 tensor. The mathematical idea in ADDA on the matrix vector multiplication is to do a 3D-FFT of the tensor, then multiply the result with another tensor element by element and transform it back via the inverse 3D-FFT to carry out the complete matrix vector multiplication as explained Section 5.1. ADDA uses the convention that the longest dimension of the scatterer is aligned to the $x$-direction. After the $\text{FFT}$ in $x$ direction, the further calculation is split into independent slices because a multidimensional $\text{FFT}$ consists of a series of independent 1D-$\text{FFT}$ for each dimension. That allows to spread the calculation over several cores or even several computers using $\text{MPI}$. At first glance it might seem to be a good choice to also split the computational task at this point and then let these slices be calculated in parallel on the cores of the $\text{GPU}$. However, Section 5 will show that this is not possible. In Figure 5(b) we used the built-in precise timing compiler option of ADDA to determine the timing of the subparts of the MatVec routine. The run-times for the parts which are executed for every slice are added up over all slices, so that the percentage values in Figure 5(b) are percentages of one complete matrix vector multiplication.

As scatterer we chose a sphere with about 2.6 million dipoles, where one matrix vector multiplication took about 15 seconds. The distribution of the time consumption of the subparts does not vary much with the size of the grid.

Figure 5(b) shows a symmetric order. The subparts $\text{arith1}$ to $\text{ fftY}$ and $\text{ ifftY}$ to $\text{arith5}$ are one 3D-FFT and one inverse 3D-FFT respectively. So $\text{arith1}$ does the inverse of $\text{arith5}$ and $\text{arith2}$ does the inverse of $\text{arith4}$. The difference in the execution time occur because during the 3D-FFT some arrays are set to zero before the arithmetic subparts are executed. Those times are included, so that one matrix vector multiplication takes exactly the sum of the time of the subparts to run. The $\text{arith3}$ subpart is the only subpart which uses the largest matrix needed, the $D\text{matrix}$ denoted as $D$. This matrix is basically

$$D = -\text{FFT}(G)/N_{\text{grid}},$$

where $N_{\text{grid}}$ denotes total number of dipoles and empty grid points in the doubled grid (Section 4.3) and $G$ the Green’s tensor (Section 4.2) so that for a vector we obtain essentially

$$-G \cdot x = i\text{FFT}(D\ast\text{FFT}(x)), \quad i\text{FFT}(\text{FFT}(x)) = N_{\text{grid}} \cdot x,$$

where the $*$ denotes convolution. This is the exact operation done in MatVec where the convolution itself takes place in $\text{arith3}$. There are some other approaches for the matrix vector multiplication. Barrows et al. developed an $O(N \log N)$ algorithm with minimal memory requirements of $O(2^M N)$, where $M$ denotes the number of Blocks in the Block-Toeplitz matrix, which also might be suitable for parallel implementing in $\text{OpenCL}$.

5. Parallel Implementation of the Matrix Vector Multiplication in OpenCL

5.1. Splitting into Slices
The algorithm of the matrix vector multiplication in ADDA is designed to be executed in parallel by multiple $\text{CPUs}$ by splitting the matrix into independent slices.
We take a closer look at the possibility to use this approach for distributing the work onto the GPU cores. In this approach, there will be as many slices as the x-dimension of the crystal-grid has elements. For some GPU architectures this will result in a low occupancy of the cores when there are more cores than slices. But even with an equal number of cores and slices, the performance will not be that good because it needs at least a few ten thousand executions of a kernel (parallel tasks) to be efficient. There is one more reason why the splitting into slices is inefficient on GPU. One core on the GPU has limited registers which are used for private memory storage. By splitting into slices, one core has to evaluate 4 FFTs, 3 arithmetic routines and 2 transposes. The FFTs and the routine arith3 use a lot of temporary variables stored in registers. If all register memory is used up, OpenCL lowers the occupancy to have enough registers (Section 5.2). This would result in very low occupancy and therefore low performance. For these reasons, we took another approach to achieve reasonable performance. We utilize one kernel for every subpart of the MatVec routine (every box in Figure 5(a)). So the kernels are executed more often and mainly processing one data point at one GPU core. The required time to start a kernel is quite low, but still noticeable. So usually it is not efficient to loop over kernel calls. A Faster solution is to add a new index for a dimension to the kernel to do the same thing. We kept the splitting into slices instead of performing a 3D-FFT because it can still be useful with the use of multiple GPUs. With little programming effort, the slices can be spread to several GPUs and then be calculated independently in parallel. Theoretically there is no limit to the number of devices calculating in parallel in OpenCL but the goal might be four since actual consumer motherboards have up to four expansion slots usable for graphic cards.

5.2. Reducing Memory Transfer

For a grid of $128 \times 128 \times 128$ all the matrices needed by the MatVec routine occupy 588MB of memory if using CPU. In Section 4 we mentioned that the bottleneck of GPU calculations is the memory transfer, so our goal is to reduce it to a minimum. This can be achieved by uploading the data needed by the MatVec routine first, and keep them in the GPU memory during the whole matrix vector multiplication or even better during the whole ADDA runtime. While the data is in the GPU memory, the CPU cannot modify the data or even access it, so we have to do all calculations on the GPU in order to keep memory transfer low. Fortunately only a small amount of data (the result vector and the argument vector) have to be transferred to/from host to exchange them with the iterative solver. The Bmatrix, mentioned in 3.3, can be kept in device memory for the whole ADDA runtime.

5.3. Arithmetics and Transpositions

The arithmetic operations (arith1 to arith5) and transpositions (TYZ) are mainly memory copy operations, so the naming is a bit misleading. By just converting them, without much optimization to GPU architecture, we achieve a fair improvement in speed because of the higher memory speed of the GPUs.

5.4. Fast Fourier Transform (FFT) Details

The Fast Fourier transform parts are all batched 1D-FFTs, i.e., a sequence of small FFTs, of the same size. Unfortunately there is no FFT library like FFTW or CUDA FFT (CUFFT) available for OpenCL yet. Therefore we tried three different implementations. All of these are only capable of performing power-of-two-sized FFTs.

One FFT per GPU Core. As first implementation, a radix-2 decimation in time (DIT) algorithm is used [24]. This approach employs one core of the GPU for one FFT. The execution of the corresponding kernel produces the whole batched FFT. The performance of this approach was poor because (a) all of the cores have to access the global memory very often (high latency) and (b) the number of FFTs to execute are smaller than the number of cores inside the slice loop (low occupancy). This implementation needs about a factor 5 the time of the FFTW library on the CPU in DP.

One FFT per Workgroup, multiple GPU Cores for One FFT. As base for this approach, we modified the source code of the Cooley Tukey implementation of the clFFT library which is also a DIT radix-2 implementation. The clFFT library is a massive parallelization framework for FFTs, but is apparently not being developed any further and is not directly usable for our purpose. However after some effort this approach is faster than the last one because of more efficient usage of the GPU cores. One FFT is computed on many cores using the local memory to share data between different cores of one workgroup. This is possible because the size of one of the batched FFT fits into the 16 KB of local memory per work group which is available on our GPU. While this approach is a lot more optimized than the last one it still does not meet the performance of the FFTW library on the CPU and runs at about the half of the speed of the FFTW in DP.
Apples OpenCL FFT. Apple developed a FFT library \cite{26} meant as example for OpenCL programming. It uses meta programming (Section 4.6) and is based on the work of Volkov and Kazian \cite{27} and Naga et al. \cite{28}. It uses a mixed radix approach originally developed by Cooley and Tukey \cite{29} with power of 2 radixes up to 32 resulting in fewer operations for larger FFTs. It is written in C++, uses some Apple-specific libraries and was designed for SP so a few customizations had to be done, to run it in DP under the Linux operating system as part of a C program. Since there are actually no native functions available for DP, we expect this library to be a lot slower for DP than for SP for which it was designed. The Apple FFT library also use mad-functions (Section 4.6) which slightly lower the accuracy. Since this library does not return the exact FFT in DP this might have a negative influence on the convergence of the iterative solver as on the result. We compare the convergence and result of the original matrix vector multiplication with our implementation in SP. On our GPU the Apple FFT is a bit faster in DP than the FFTW library on our CPU. One interesting aspect is that the batch size is not involved in the FFT-planning phase of the Apple FFT implementation. This means that the kernel file which is generated during the planning phase does not know about the number of total executions of same-sized FFTs. This will not reduce the performance if the FFT size is large enough. In our case, however, the FFT size is quite small (maximal twice the longest grid dimension of the scatterer) so that there might be unused CPU cores for one FFT. If the batch size would influence the plan as well there might be a faster implementation than the actual one.

The Apple FFT planning phase takes a lot of time. In ADDA the creation of the plan for the three different FFT sizes used in the matrix vector multiplication actually takes about 30 seconds for arbitrary grid sizes. However this time becomes less important with growing number of iterations and growing crystal sizes, so basically for large run times. The Apple FFT showed the best performance out of the three tried implementations of OpenCL FFTs. However, in SP the Apple FFT is about 10 times faster on our CPU than in DP (Section 5.6).

6. Performance Comparisons of GPU and CPU implementations

6.1. Runtime

We decided to use the FFT implementation of Apple (Section 6.1) because it showed the best performance. The arithmetic routines were converted to OpenCL without architecture dependent customizations by using the guideline of Section 3. Figure 4 compares the runtime of our CPU implementations of the matrix-vector multiplication of ADDA with the original GPU implementation on a sphere in a grid of 128×128×128. We use a power-of-two grid to have comparable results since non-power-of-two grids must be expanded to the next power-of-two when using the GPU (Section 5.6). The CPU implementation shows a different distribution of time consumption of the subparts compared to Figure (5(a)). This is because the FFTW library is still a bit faster in transforming power-of-two sizes than in transforming other sizes since it uses a split radix approach. It utilizes higher power of 2 radixes like 4, 8, 16, 32 to reduce the number of operations.

The arithmetic routines were converted to OpenCL without GPU architecture dependent customizations by using the guideline of Section 3. Figure 4 compares the runtime of our GPU implementations of the matrix-vector multiplication of ADDA with the original GPU implementation on a sphere in a grid of 128×128×128. We use a power-of-two grid to have comparable results since non-power-of-two grids must be expanded to the next power-of-two when using the GPU (Section 5.6). The CPU implementation shows a different distribution of time consumption of the subparts compared to Figure (5(a)). This is because the FFTW library is still a bit faster in transforming power-of-two sizes than in transforming other sizes since it uses a split radix approach. It utilizes higher power of 2 radixes like 4, 8, 16, 32 to reduce the number of operations.

The arithmetic routines were converted to OpenCL without GPU architecture dependent customizations by using the guideline of Section 3. Figure 4 compares the runtime of our GPU implementations of the matrix-vector multiplication of ADDA with the original GPU implementation on a sphere in a grid of 128×128×128. We use a power-of-two grid to have comparable results since non-power-of-two grids must be expanded to the next power-of-two when using the GPU (Section 5.6). The CPU implementation shows a different distribution of time consumption of the subparts compared to Figure (5(a)). This is because the FFTW library is still a bit faster in transforming power-of-two sizes than in transforming other sizes since it uses a split radix approach. It utilizes higher power of 2 radixes like 4, 8, 16, 32 to reduce the number of operations.
and `arith5` differ a bit because the times include the data preparation, up- and download. The same applies to `arith2` and `arith4`. The `arith2`, `arith4` and `TYZ` (transpose) routines show nearly the same speed-up in `DP` and `SP` since they are mainly memory copy functions which are quite fast because of the high memory bandwidth of the GPU memory but do not depend on the arithmetic calculation speed. In all FFT routines, the fraction of arithmetic operations is higher and therefore the speed-up is a lot smaller in `DP` than in `SP`. Figure 8 compares in a double logarithmic plot, the time consumption for a matrix-vector multiplication on `CPU` and `GPU` for different grid sizes. For very small grids the `CPU` implementation is a lot faster because of the transfer times from and to `CPU` memory. For grid sizes of at least $32 \times 32 \times 32$ the `GPU` is faster than the `CPU`. This size is exceeded in most ice crystal representations. On the upper x axis the memory requirements (in MB for `DP` calculations) of the `MatVec` matrices for the corresponding sizes on the lower axis are shown.

![Figure 5: Total runtime of one matrix-vector multiplications for a sphere in different cubic grid sizes on `CPU` and `GPU` in `DP` and `SP`. Upper axis: memory used by `MatVec` in `DP` for the corresponding grid size. Lower axis: grid size along one axis, i.e., to the power of three equals total grid points.](image)

### 6.2. Accuracy

The Apple FFT library uses `mad`-functions and therefore has lower accuracy than the `FFTW`. We compare the convergence for the `QMR` solver with our implementation (using the Apple FFT) with the original `ADDX` (using the `FFTW`). We calculated a sphere in a $128 \times 128 \times 128$ grid using 1.1 million dipoles. With our `DP` implementation of the matrix-vector multiplication `ADDX` needed 5246 iterations and taking about 145 minutes while with the the original `ADDX` needed 5303 iterations and taking about 715 minutes. Therefore it seems that the reduced accuracy due to the `mad`-functions influences the convergence only slightly.

For an accuracy study of many `ADDX` scattering programs including `ADDX` see Penttilä et al. [31].

### 6.3. Limitations of the OpenCL Implementation

The Apple FFT implementation takes about 30 seconds for the planing phase in one `ADDX` run. It is only capable of doing power-of-two sized FFTs while `FFTW` is able to compute FFT sizes wich are $2^a \cdot 3^b \cdot 5^c \cdot 7^d \cdot 11^e \cdot 13^f$ with $a, b, c, d, e, f \in \mathbb{N}$ and $u, v \in [0, 1]$ which makes it a lot more flexible. Another limitation is the required memory avaiable on the `GPU`. The used memory just depends on the size of the grid as

$$192 \cdot (2 + x/2) \cdot y \cdot z + x \cdot (y + 1) \cdot (z + 1)$$

in bytes in `DP` where $x, y$ and $z$ denotes the grid size in the corresponding directions. The maximum total grid size tested was $128 \times 128 \times 128$, but larger grid sizes in one dimension are also possible if the whole grid still fits into `GPU` memory. Another restriction is the size of the $D_{\text{matrix}}$ (eq. 13), since it must be smaller than the maximum memory allocation size for one memory object in `OpenCL`, which depends on the device and its driver.

### 7. Application example

We demonstrate the functionality of the implementation by applying it to a 6-branch-bullet-rosette-shaped crystals in a grid of $76 \times 76 \times 76$ dipoles where the wavelength is one per crystal dimension. As a validation criterion Draine and Flatau pointed out that one needs at least $10|m|$ dipoles per wavelength to obtain an accurate result [31] with $m$ the complex refractive index. In our example we fulfill this condition with 76 dipoles per wavelength. The crystal consists of 4746 non-void dipoles which is about one per cent of the grid points. The rosette shape in the grid is shown in Figure 6. We compare the element $M_{ijkl}$ of the Mueller matrix with respect to the scattering angle $\theta$ around the $(y, z)$ plane for three different orientations of the scatterer defined by the three Euler angles $\alpha, \beta, \gamma$ [31]. The refractive index $m = 1.778 + 0.002i$ used here is a typical value.
of ice at $-20^\circ$C to $-60^\circ$C and 1 mm to 10 mm wavelength (30 GHz to 300 GHz). This range is at the same time also the maximum dimension of the corresponding crystal, which is typical for ice crystals in convective systems. The result is shown in Figure 6. For the Euler angles $\alpha = \beta = \gamma = 0^\circ$, scattering is symmetric around $\theta = 180^\circ$. At $\theta = 90^\circ$ and $\theta = 270^\circ$ the scattering has an absolute minimum, while at $\theta = 0^\circ$ and $\theta = 180^\circ$ there are strong peaks due to forward- and backscattering.

$\alpha = 0, \beta = 72, \gamma = 40$

$\alpha = 0, \beta = 72, \gamma = 60$

As second example, for a shorter wavelength, the polarization $P(\theta)$ is shown in Figure 8(a). For the Euler angles $\alpha = \beta = \gamma = 0^\circ$, the rosette is axis symmetric around all three axis and it has no preferred orientation in the atmosphere. Figure 8(b) shows the polarization of the ensemble of crystals with orientation homogeneously distributed over all possible directions. ADDA uses Rhomberg integration [8] for this purpose. Depending on the resolution, the range of orientation angles and on the complexity of the crystal this calculation can take several hours. ADDA can use the symmetries of the crystal to reduce the calculation time and calculates as many angles needed to reach a specific accuracy.

Figure 6: Rosette in a grid of 76x76x76 with a total of 4746 dipoles

Figure 7: Scattering intensity (Müller matrix element $M_{11}$) for a 6-branch-bullet-rosette with a size/wavelength ratio of 1

Figure 8: Degree of polarization of a rosette of the size of 76 dipoles of five times the wavelength, i.e., 15 dipoles per wavelength. (a) Euler angles $\alpha = 0^\circ, \beta = 0^\circ, \gamma = 0^\circ$, (b) averaged Orientation
This size of the grid is nearly worst case to our implementation since we have to expand the grid to $128 \times 128 \times 128$ to meet the power-of-two requirements of our Fourier transform. The original implementation with the FFTW expands the grid to $77 \times 77 \times 77$ since $77 = 7 \cdot 11$. Our actual implementation has to calculate a factor of $(128)^3/(77)^3 \approx 4.6$ more grid points than the original matrix vector multiplication of ADDA. The results of our OpenCL implementation of MatVec can have slightly different results due to the mad-functions in the Apple FFT as mentioned in Section 5 and 6. However, for this rather small rosette the results for GPU and CPU are identical within the accuracy ADDA provides the Mueller matrix data (11 significant digits) in DP.

8. Conclusions and Outlook

In this study, the potential to use GPUs for light scattering calculations with DDA has been investigated. The time consuming matrix vector multiplication of the DDA was reimplemented in OpenCL. The ADDA scattering program by Yurkin and Hoekstra was modified to use the OpenCL implementation of the matrix vector multiplication. The OpenCL programming example from Apple for FFT was modified to handle DP and used to calculate the FFTs needed for the matrix vector multiplication. The other parts were written to small OpenCL functions to perform one matrix vector multiplication with a minimum of memory transfer to and from the GPU. The modified ADDA achieved a speedup factor of around 5/15 at D15 for a $128 \times 128 \times 128$ grid on a Nvidia GTX 260 GPU compared to a 2.8 GHz AMD Athlon II X2 e240 CPU. On some subparts of the matrix vector multiplication the speedup is more than a factor of 20 (Figure 4).

We expect a total speedup up to a factor of 20/40 at DPs with a more recent GPU which has full native support of DP calculations like e.g., the Nvidia Tesla C2050/C2070. But even for the consumer GPUs like the Nvidia GTX 580 we expect a speedup of around 10/25 at DPs. The current limitation in grid size is $128 \times 128 \times 128$ dipoles, but one dimension can be up to 256, when the other dimensions are smaller. The limitation of the modified ADDA is the Fourier transformation which gets just a small speed-up at DP and is also limited to power-of-two sizes, so that in the worst case the shapes the GPU has to calculate nearly eight times more data than the CPU implementation. Since generally the interest in FFT and linear algebra functions is high, it is expected that suitable GPU libraries for OpenCL will become available in the near future. The Apple FFT example and the ViennaCL [13] library are a good start for computations in OpenCL. As an alternative to OpenCL CUDA must be mentioned because Nvidia already offers a linear algebra library called CUDA Basic Linear Algebra Subprograms (CUBLAS) and the FFT library CUFFT but both are limited to Nvidia GPUs.

As another approach to accelerate the smaller grid sizes one can think about doing a full 3D-FFT (also available in the Apple FFT example) since this would drastically reduce the number of kernel calls and thus improve the performance. However, starting from the current state it would be an easy task to extend our OpenCL implementation of the matrix vector multiplication to run on multiple GPUs. With little programming effort, a complete OpenCL FFT library which can handle non-power-of-two sizes, and a moderate sized GPU cluster of about four GPUs in one computer, we expect a speedup of factor of 50 compared to the original single CPU implementation. This is as fast as a cluster of 25 dual core CPU computers at about a tenth of power consumption and a fifth of acquisition cost. This is not as much of an improvement as expected. The main reason is the requirement to use DP calculations since recent GPUs are still two to eight times faster at SP calculations. Nevertheless we see a great potential in GPU computing with appropriate optimizations to perform computationally intensive tasks at much lower costs than with sequential computers.

Acknowledgements. Fruitful discussions with Thomas Wriedt on scattering and the suggestions of Maxim Yurkin on the possibility to contribute GPU calculations to the DDA are gratefully acknowledged.

References


