Technical Report TR-2012-03

Parallel Ray Tracing Simulations with MATLAB for Dynamic Lens Systems

Nicolai Wengert and Dan Negrut

August 10, 2012

Abstract

Ray tracing simulations are required for investigating the dynamical behavior of optical systems. By means of image simulations, an exposed image can be generated. However, this requires a high number of rays which have to be traced through an optical system. Since all rays are independent of each other, they can be traced individually using parallel computing.

In the first part, an overview to GPU computing using the Parallel Computing Toolbox in MATLAB will be given. The second part outlines the algorithm for the ray tracing simulation. This includes basic ray tracing formalisms as well as lens kinematics for rigid lenses. Finally, an example is used to demonstrate speedup results obtained using MATLAB parallel computing on the GPU.

Keywords: Parallel Computing, MATLAB, CUDA, Ray Tracing, Lens Kinematics

Contents

1 Introduction

Over the last five years it has become apparent that future increases in computational speed are not going to be fueled by advances in sequential computing technology. There are three main walls that the sequential computing model has hit. Firstly, there is the power dissipation wall caused by the amount of energy that is dissipated per unit area by ever smaller transistors. Since the amount of power dissipated scales with the square of the clock frequency, steady further clock frequency increases, which in the past were responsible for most of the processing speed gains, are unlikely.

The second wall, that is, the memory wall, arose in sequential computing as a manifestation of the gap between processing power and memory access speed, a gap that grew wider over the last decade. A single powerful processor will likely become data starved, idling while information is moved back and forth between the chip and main memory over a bus typically clocked at 10 to 30 GB/s. Ever larger caches alleviate the problem yet technological and cost constraints associated with large caches can't reverse this trend.

Thirdly, investments in Instruction Level Parallelism (ILP), which mainly draws on instruction pipelining, speculative execution, and branch prediction to speed up execution represent an avenue that has already exhausted its potential. Both processor and compiler designers capitalized on opportunities for improved performance in sequential computing, which came at no cost to the software developer. Augmenting branch prediction and speculative execution beyond current instruction horizons comes at a power price that in many cases increases exponentially with horizon depth.

Nowadays, performance improvements are realized by making many processors working simultaneously. Since GPU (graphics processing unit) cards provide good architectures for massively parallel tasks, they are of growing interest beyond graphics computations. Especially in science, General Purpose Computing on GPUs (GPGPU) is used for more and more applications. Herein, it will be used for ray tracing simulations in optical systems with moving lenses. This topic is of interest in photolithography, where vibrations of lenses in the lithography objective cause problems. Ray tracing in combination with a Monte Carlo algorithm can be used to simulate the imaging behavior for moving lenses. This helps design engineers to understand the problem of vibrations in lens systems.

Image simulations have to be done at many time instants, so-called snapshots. The exposed image can be obtained by integrating these snapshots over time. However, those exposure simulations only represent a small area of the image plane. For estimating the exposure quality for the full image plane, many exposure simulations are necessary.

In the simplest optical case, for a medium quality, a single snapshot requires at least one million rays. Characterizing optical phenomena like diffraction requires even more rays. Diffraction plays an important role in photolithography, so this must not be ignored. Therefore, exposure simulations might require more than one billion rays. Since all rays are independent of each other, the image simulation is a typical SIMD (single instruction, multiple data) problem. GPU computing provides a good way to solve this SIMD problem efficiently.

2 GPU computing in MATLAB using the Parallel Computing Toolbox

With the Parallel Computing Toolbox, MATLAB offers two possibilities for running simulations on the GPU: the GPUArray avenue and the use of CUDA kernels. Currently, there are some general restrictions in the Parallel Computing Toolbox:

- it is solely based on CUDA, i.e. only CUDA-capable GPUs are supported
- for calculations in double precision, a GPU card with computing capability of at least 1.3 is required
- multiple GPU cards are not supported

Assuming a CUDA-capable GPU card is available and that the latest CUDA drivers and toolkit have been installed [5, 6], the command `gpuDevice` can be used to check if the card is installed correctly. Upon successful return, this function call will provide a report about the parameters of the GPU card available on the system.

In the following, the GPUArray environment and the execution of CUDA kernels will be described briefly. Further information and examples can be found in the documentation of the Parallel Computing Toolbox [4].

2.1 The GPUArray environment

The MATLAB's GPUArray environment is a simple way of using parallel computing on GPUs. It can be used for MATLAB built-in functions as well as for user-written MATLAB code. In both cases, three steps are necessary:

- copy data sets from MATLAB workspace to the GPU with the `gpuArray` command
- execute built-in and/or user-written functions
- copy results back to the MATLAB workspace with the `gather` command

A list of supported built-in functions can be found in the MATLAB Help. The advantages and disadvantages are as follows:

⊕ it is easy to use

⊕ no additional settings required regarding thread execution

⊖ unlikely to take advantage of the full potential of the graphics card, as this requires adjusting settings/parameters that are not user accessible in this approach

User-written functions are executed by the `arrayfun` command. Unfortunately, there are some restrictions which make this method not suitable for extensive calculations:

⊖ calling subfunctions is not allowed

⊖ indexing is not allowed, i.e. matrices or vectors cannot be used and constant data has to be passed as a set of scalars

Examples for both using built-in functions and user-written functions can be found in the Appendix.

2.2 Running CUDA kernels

In conventional CUDA programming, a CUDA kernel is embedded in C or Fortran code. A CUDA kernel contains the function which is to be run in parallel as well as the information, how the "work" is to be distributed on the GPU. The Parallel Computing Toolbox supports a similar approach from within. In fact, the syntax for CUDA kernel call embedding turns out to be simpler. MATLAB allocates memory on the GPU automatically. Furthermore, additional data created during the kernel function call is cleared automatically. This saves a lot of commands regarding memory management. Result data on the GPU is treated by MATLAB the same way as in the GPUArray environment, i.e. it is defined as a GPUArray object. The memory on the GPU is deallocated by clearing those GPUArray objects.

Assuming a CUDA file including a kernel function is available, the first step is to compile this file and to create a PTX file (parallel thread execution) by means of the nvcc compiler available on the host machine.

```
nvcc -ptx -arch=sm_13 myCuFile.cu
```

PTX is a pseudo-assembly language which is translated into binary code by the graphics driver [1]. A compiled PTX file with a higher computing capability than 1.3 is currently not supported by the Parallel Computing Toolbox. The PTX file is required to set up a CUDA kernel object in MATLAB, e.g. by

```
kern = parallel.gpu.CUDAKernel('myCuFile.ptx');
```

This CUDA kernel object `kern` can be regarded as a MATLAB function. It stores kernel execution configurations, namely block size and grid size, which are set as follows:

```
kern.ThreadBlockSize = [64 64 1];
kern.GridSize = [64 1];
```

The CUDA kernel is executed by

```
[out1, out2, ...] = feval(kern, in1, in2, ...);
```

where the output arguments are GPUArray objects which can be copied to the MATLAB workspace by means of the `gather` command. The input arguments can be predefined as GPUArray objects, if wanted, but it is not required. An example is given in the appendix. Note that

- the nvcc compiler searches for the `__global__` function to define an entry point

- the kernel function in the CUDA file must not have a return value. In C, it should return void.
- output arguments in the `__global__` function call are defined by using pointers
- the output arguments `out1,...,outn` refer to the first n arguments of the function call, e.g. for
  `out1 = feval(kern, in1, ...);`
  the kernel function call has to be
  `__global__ void kernelFun(double *out1, ...)`
  The input argument `in1` is only used for preallocating `out1`. See Appendix A.1.3 for a full example.

The advantages and disadvantages of this method are as follows:

⊕ existing CUDA code can be easily invoked from within MATLAB

⊕ the execution efficiency can be tuned by adjusting grid size and block size

⊕ it is possible to execute several CUDA kernels in sequence without copying data from or to the GPU

⊖ for complex code, a good knowledge about CUDA is required

3 Ray tracing with moving lenses

This section gives a short introduction to investigating the imaging behavior of lens systems with moving lenses. First, a ray tracing formalism is introduced. This formalism allows for computing the path of a single ray through a lens system. This is expanded to moving lenses next. The section closes with a discussion of accurate image generation based on a large number of rays.

3.1 Basic ray tracing formulas

Ray tracing is part of the geometrical optics which neglects wave-optical effects. It is based on Fermat's principle which in turn follows from Huygen's principle. Mathematically, a light ray is a straight line described by a position vector $\boldsymbol{r}$ and a direction vector $\boldsymbol{d}$. In optical systems, the ray changes directions at surfaces. The quantities of interest are the intersection points of a ray and the surfaces, and the ray directions at these intersection points:

$$ {}^{s}\boldsymbol{r}_i = \begin{bmatrix} {}^{s}r_{i,x} \\ {}^{s}r_{i,y} \\ {}^{s}r_{i,z} \end{bmatrix} \qquad \text{and} \qquad {}^{s}\boldsymbol{d}_i = \begin{bmatrix} {}^{s}d_{i,x} \\ {}^{s}d_{i,y} \\ {}^{s}d_{i,z} \end{bmatrix} . \tag{1} $$

The index s determines the number of a lens system S_s in which the vector is defined, and i is the number of the surface hit by the ray.

In sequential ray tracing used herein, the order of the surfaces a ray has to pass is predetermined. It is also assumed that there are only spherical surfaces. For the algorithm, it is important to consider the sign convention regarding the surface radii, see Fig. 1.

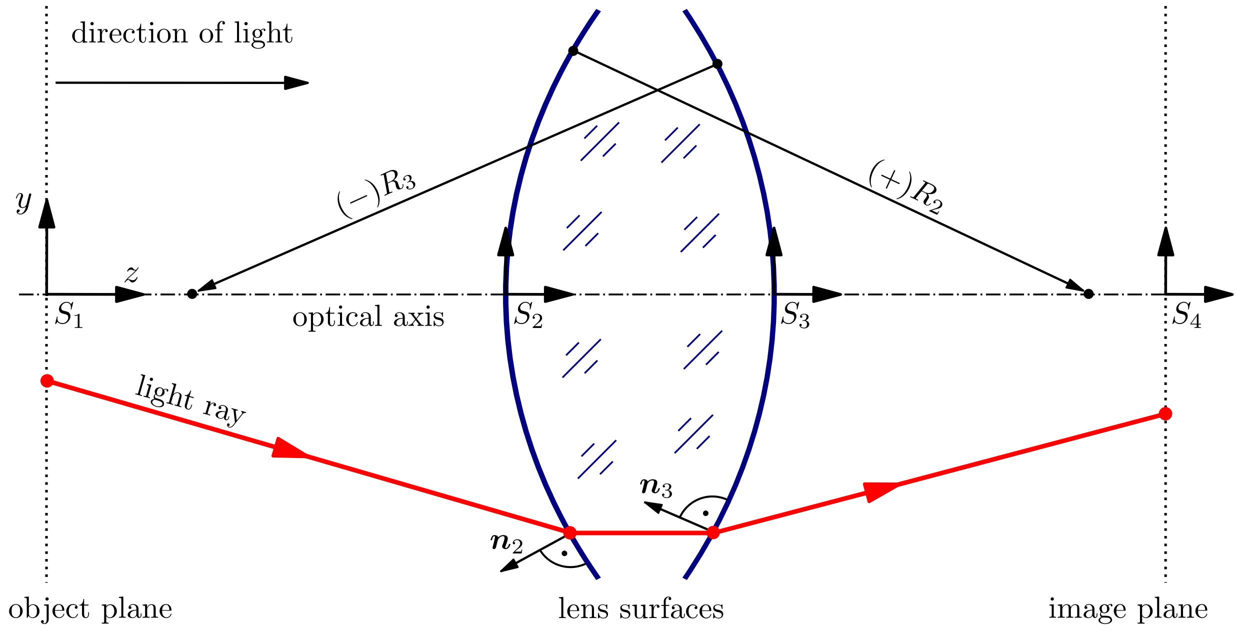


Figure 1: Sign convention for ray tracing with lenses

An optical system has n surfaces, including the object plane and the image plane. Every surface i has a surface system S_i. The ray is specified at the object plane by ${}^1\boldsymbol{r}_1$ and ${}^1\boldsymbol{d}_1$.

Before computing the intersection point of a ray and a surface i, the ray must always be transformed into the surface system S_i. This transformation will be described in Sec. 3.2. The new ray position can be calculated according to [3],

$$ {}^i\boldsymbol{r}_i = {}^i\boldsymbol{r}_{i-1} + H\,{}^i\boldsymbol{d}_{i-1}, \tag{2}$$

with the intermediate parameters

$$ F = {}^i d_{i-1,z} - \frac{{}^i\boldsymbol{r}_{i-1} \cdot {}^i\boldsymbol{d}_{i-1}}{R_i} \tag{3}$$

$$ G = \frac{|{}^i\boldsymbol{r}_{i-1}|}{R_i - 2\,{}^i r_{i-1,z}} \tag{4}$$

$$ H = \frac{G}{H + \sqrt{H^2 - G/R_i}}, \tag{5}$$

and the radius of the new surface R_i.

For calculating the new ray direction ${}^{i}\boldsymbol{d}_i$ at the intersection point ${}^{i}\boldsymbol{r}_i$, the surface normal ${}^{i}\boldsymbol{n}_i$ is required,

$$
{}^{i}\boldsymbol{n}_i = -\frac{1}{R_i}\,{}^{i}\boldsymbol{r}_i + \begin{bmatrix}0\\0\\1\end{bmatrix}. \tag{6}
$$

The formula for the new ray direction is derived from Snell's law [3],

$$
{}^{i}\boldsymbol{d}_i = \frac{1}{n_1}\left(n_0\,{}^{i}\boldsymbol{d}_{i-1} + J\,{}^{i}\boldsymbol{n}_i\right), \tag{7}
$$

with the refraction index n_0 of the medium before the surface and n_1 after the surface, and the intermediate parameters

$$
I = n_0\,({}^{i}\boldsymbol{d}_{i-1} \cdot {}^{i}\boldsymbol{n}_i) \tag{8}
$$

$$
J = \sqrt{n_1^2 - n_0^2 + I^2} - I. \tag{9}
$$

3.2 Lens kinematics

If an optical system is subjected to dynamical excitation, the lenses will move and tilt. The motion can be simulated by means of multibody system dynamics. In a postprocessing step, it can be extracted and subsequently used with optical systems. For a lens j, the motion is described by the displacement $\boldsymbol{\rho}_j$ and the orientation $\boldsymbol{A}_j$, see Fig. 2. The initial position $\boldsymbol{p}_j$, displacement $\boldsymbol{\rho}_j$, and orientation matrix $\boldsymbol{A}_j$ are defined in the inertial system S_1.

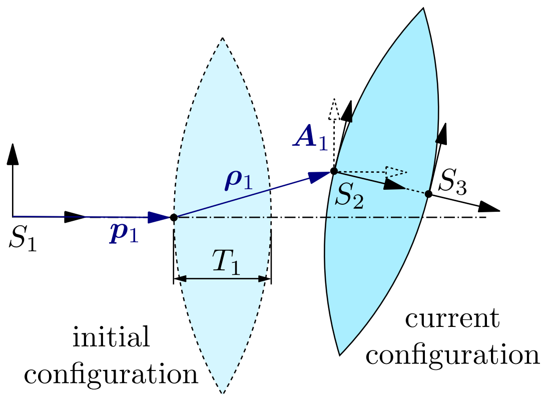


Figure 2: Rigid motion of the first lens of an optical system

Before computing the intersection point ${}^{i}\boldsymbol{r}_i$ of a ray and a surface i, the previous intersection point ${}^{i-1}\boldsymbol{r}_{i-1}$ has to be transferred to the coordinate system S_i. The formula for the transformation depends on whether the ray will hit another surface of the current lens or a new lens. In the first case, the transformation reads

$$ {}^{i}\boldsymbol{r}_{i-1} = {}^{i-1}\boldsymbol{r}_{i-1} - \begin{bmatrix} 0 \\ 0 \\ T_j \end{bmatrix}, \tag{10} $$

with T_j the thickness of the current lens j. If the ray will hit a new lens, the index j is the index of the new lens and the transformation becomes

$$ {}^{i}\boldsymbol{r}_{i-1} = \boldsymbol{A}_j^T \cdot \boldsymbol{A}_{j-1} \cdot \left({}^{i-1}\boldsymbol{r}_{i-1} + \begin{bmatrix} 0 \\ 0 \\ T_{j-1} \end{bmatrix} \right) + \boldsymbol{A}_j^T \cdot \left(\boldsymbol{p}_{j-1} + \boldsymbol{\rho}_{j-1} - \boldsymbol{p}_j - \boldsymbol{\rho}_j \right). \tag{11} $$

There are two special cases. If the old surface is the object plane, $\boldsymbol{A}_{j-1} = \boldsymbol{I}$ and $\boldsymbol{p}_{j-1} = \boldsymbol{\rho}_{j-1} = \boldsymbol{0}$. If the new surface is the image plane, $\boldsymbol{A}_j = \boldsymbol{I}$ and $\boldsymbol{\rho}_j = \boldsymbol{0}$.

3.3 Monte-Carlo-based image simulation

Image simulations enable the evalutation of the imaging behavior of an optical system. By using ray tracing, the image simulation approximates the intensity distribution at a screen, see Fig. 3. In general, a large number of rays is required, and the higher this number, the closer the approximation gets to an exact solution. The number of reqired rays for a good approximation depends on the resolution of the screen. Since usually a randomized algorithm is used where more rays yield a more accurate result, this image simulation is a part of Monte-Carlo methods.

For the ray tracer, a bundle of rays has to be created. The positions at the image plane and the directions are defined randomly within lower and upper limits. These limits depend on the image which is to be projected, and on optical issues like the entrance pupil, see e.g. [2] for more information.

After tracing each individual ray, the intersection points at the image plane are obtained in form of a spot diagram. A region at the image plane has to be extracted and subdivided into pixels. The number of rays hitting any of the pixels at that screen is counted. Dividing this number by the overall number of rays yields the normalized intensity at the respective pixel,

$$ \text{normalized intensity of a pixel} = \frac{\text{number of rays hitting the pixel}}{\text{overall number of rays}}. \tag{12} $$

This requires all rays of the bundle to have the same intensity. Equal intensities are allowed due to the randomly defined rays. Furthermore, the distribution of the rays within the limits must be uniform.

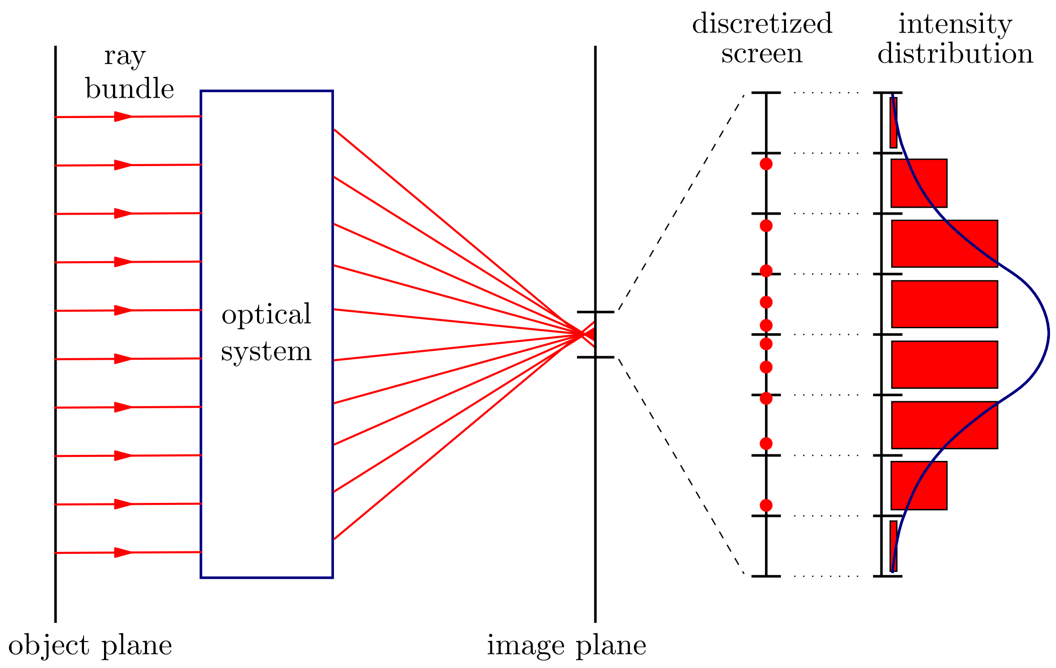


Figure 3: Basic principle of image simulations

4 Numerical Experiments

In this section, an example will be shown where the image simulation based on ray tracing is applied. First, a MATLAB implementation based on the formalism in Sec. 3 is compared to ZEMAX, a commercial optics software. After this, the GPU computing approaches introduced in Sec. 2 are compared to the sequential Matlab code. All simulations use double precision. For comparing the code implementations, the following hardware has been used:

- CPU_1: Intel Core i5-2450M
- CPU_2: Intel Xeon CPU E5520
- GPU_1: NVIDIA GeForce GT 540M
- GPU_2: NVIDIA GeForce GTX 480
- GPU_3: NVIDIA Tesla C2050 (specifically designed for GPGPU)

4.1 Image simulations with a lens triplet

In Fig. 4, a lens system consisting of three lenses can be seen. It projects the letter ”F” on a screen which is shown in the image simulation.

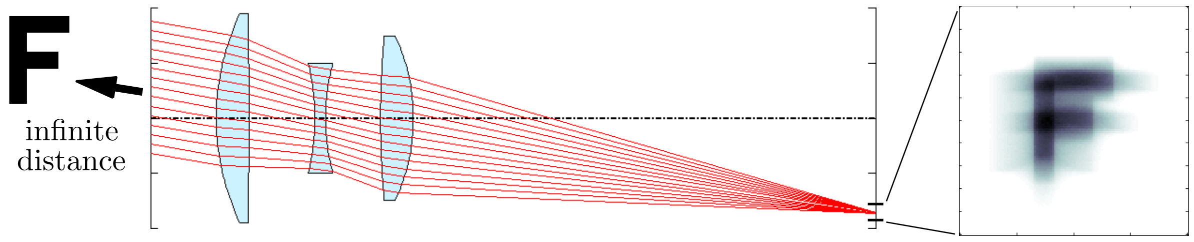


Figure 4: A lens triplet projecting the letter "F" on a screen

A randomly chosen set of lens displacements is used to compare different ray tracing codes. Four image simulations according to those displacements are illustrated in Fig. 5, sorted by the number of rays n_{rays} used for the simulation. The resolution of the screen is 128×128. Here, only sequential code was investigated using CPU$_1$. Below the image simulations are the computation times t_{MATLAB} for the MATLAB implementation and t_{ZEMAX} for the commericial optics software ZEMAX. The ray tracing kernel of ZEMAX is based on C and it is faster by a factor of 1.25. The computation times include the creation of the rays at the object plane, the ray tracing, and the computation of the image at the screen. For the MATLAB implementation, the ray tracing takes 72% of the overall time. This number depends on the lens system. In ZEMAX, the individual times cannot be measured separately.

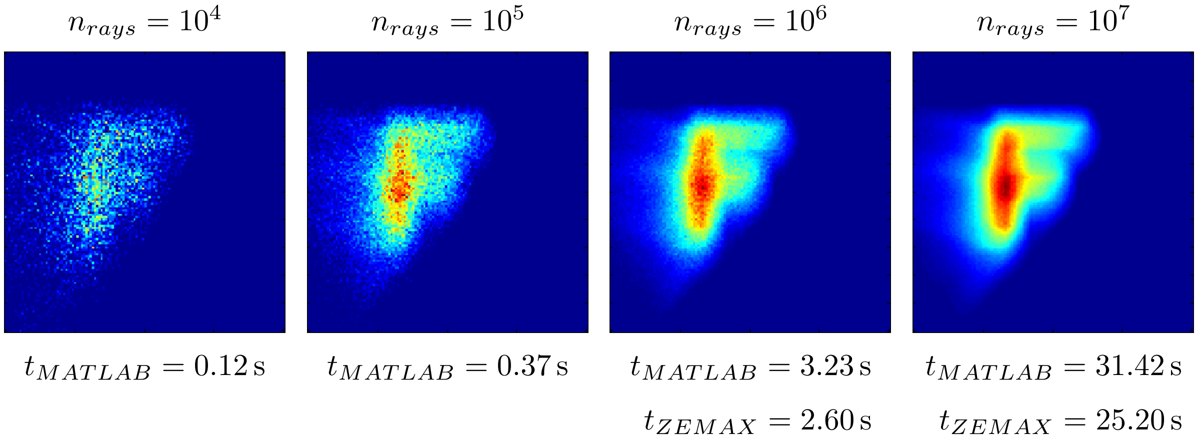


Figure 5: Image simulations with displaced lenses for different numbers of rays

4.2 Speedup results

The ray tracing part has been accelerated by means of GPU Computing. Figure 6 shows the computation time and speedups over the number of rays. The results were produced using CPU$_1$ for the sequential MATLAB Code and GPU$_1$ both for the GPUArray environment and for the CUDA kernel which invoked by MATLAB. The computation times for the GPU computations include the copying of the data to and from the GPU.

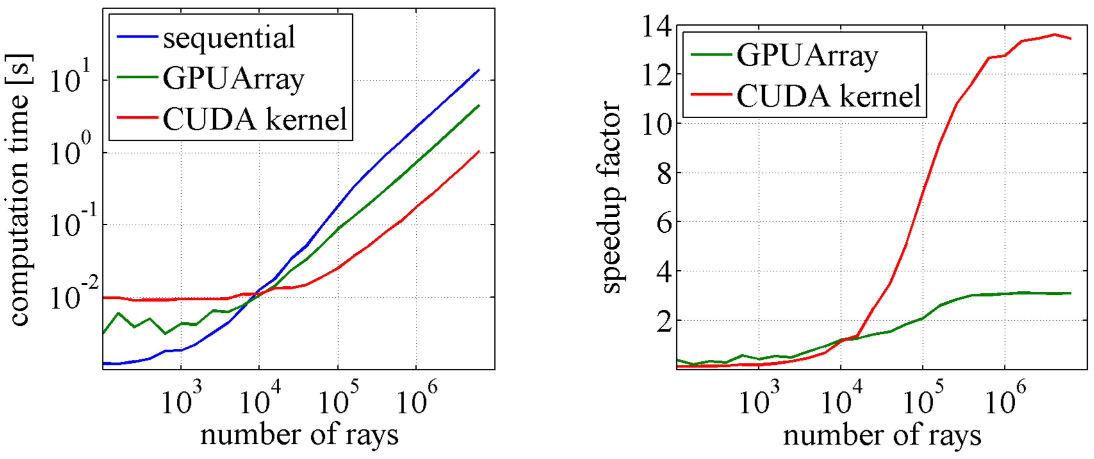


Figure 6: Computation time and speedup results of the ray tracing part

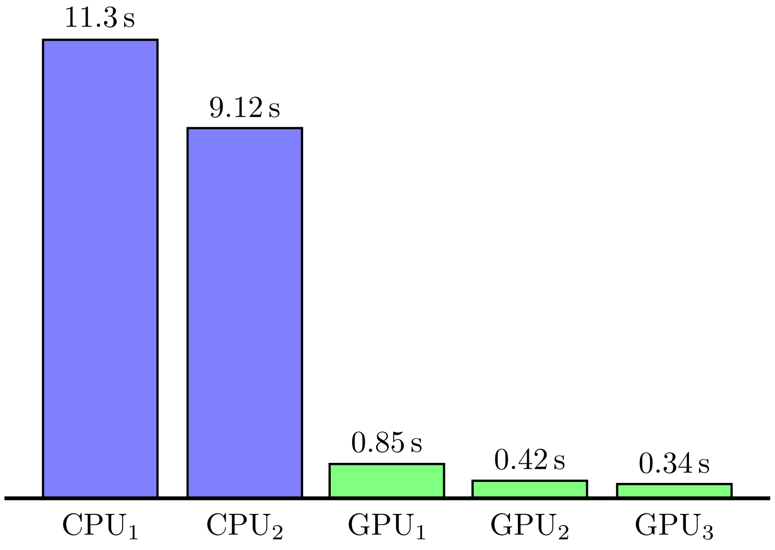
As expected, the sequential code is the slowest for high numbers of rays. It is the base for the speedup diagram. The GPUArray environment provides a small, but nevertheless, noticeable speedup factor of about 3 for more than 5 million rays. The best result is obtained by the CUDA kernel implementation for more than 1 million rays with a speedup factor of about 13. It can also be observed that the computation time increases linearly for high numbers of rays, whereas the speedup factors go to a limit.

Principally, the hardware has a huge influence on computation time. The same computations as in Fig. 6 were made using different hardware, see Fig. 7. The path of 5 million rays was calculated. Expectedly, the shortest computation time is achieved by the GPU card which is spefically designed for GPGPU (GPU_3).

5 Conclusions and outlook

By replacing the sequential ray tracing kernel in image simulations by a GPU-based kernel, a significant speedup can be attained. This advantage enables a lot of new possibilities. On the on hand, the fast computations allow for simulations with a high number of rays, i.e. beyond 100 million rays, which is required for diffraction simulations. On the other hand, image simulations for deformable lenses can be finished within a reasonable time. Those image simulations are slower by a factor of at least 10 compared to rigid lenses [7]. Moreover, MATLAB offers a good platform to embed the ray tracing kernel in a comprehensive simulation tool, e.g. for connecting the ray tracer to visualization tools.

Regarding the image simulation, there is still space for more GPU computing implementations. For example, the creation of rays at the object plane to form an object can be



speedup for GPU$_3$ with respect to

- CPU$_1$: 33.2
- CPU$_2$: 26.8
- GPU$_1$: 2.5
- GPU$_2$: 1.24

Figure 7: Computation time of the ray tracing kernels for 5 million rays using different hardware (the computation times of the GPUs include copying data to and from the GPU)

parallelized. This can be supported by creating random numbers on the GPU memory directly. Thus, copying data to the GPU can be avoided. Furthermore, the intersection points at the image plane can by prepared on the GPU before finishing the intensity distribution calculation on the GPU. Referring to the computation times in Sec. 4, it can be anticipated that the GPU-based image simulation is faster by one order of magnitude compared to the sequential computation in ZEMAX. All in all, GPU computing is a promising tool in the field of coupling dynamics and optics.

A Appendix

A.1 GPU computing in MATLAB – Examples

The computation of the following term,

$$\boldsymbol{d} = c_1\,\boldsymbol{a} + c_2\,\boldsymbol{b}, \tag{13}$$

where $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{d}$ are vectors of length n, will be demonstrated using three different approaches. The two "constants" c_1 and c_2 will be stored in

$$\boldsymbol{c} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}, \tag{14}$$

for demonstrating how to pass additional data. For reference, the straight MATLAB implementation looks like

```
nElements = 1000;
a = rand(nElements,1);
b = rand(nElements,1);
c = rand(2,1);
d = c(1)*a+c(2)*b;
```

A.1.1 Using GPUArray support with built-in functions

```
nElements = 1000;
a = rand(nElements,1);
b = rand(nElements,1);
c = rand(2,1);
a_gpu = gpuArray(a);  % copies array a onto the gpu
b_gpu = gpuArray(b);  %  - - - - - - b - - - - - -
d_gpu = c(1)*a_gpu+c(2)*b_gpu;  % performs operation on GPU
d = gather(d_gpu);  % brings back result from GPU
```

Note: In MATLAB R2012a and later, it is possible to use `parallel.gpu.GPUArray.rand` to create random numbers directly on the GPU. This is faster than copying data since the random numbers are generated directly in the GPU memory.

A.1.2 Using GPUArray support with user-written functions

```
nElements = 1000;
a = rand(nElements,1);
b = rand(nElements,1);
c = rand(2,1);
a_gpu = gpuArray(a);
b_gpu = gpuArray(b);
d_gpu = arrayfun(@calc_d,a_gpu,b_gpu,c(1),c(2));
d = gather(d_gpu);
```

The function `calc_d` is given by

```
function d = calc_d(a,b,c1,c2)
d = c1*a+c2*b;
```

A.1.3 Invoking a CUDA kernel from within MATLAB

These are the preparations for setting up the CUDA kernel object:

```
! nvcc -arch=sm_13 -ptx myCuFile.cu
kern = parallel.gpu.CUDAKernel('myCuFile.ptx');
kern.GridSize = 10;
kern.ThreadBlockSize = 100;
```

The choice of the grid size and block size is just for demonstration purposes. The next lines show the MATLAB code that embeds the CUDA kernel call:

```
nElements = 1000;
a = rand(nElements,1);
b = rand(nElements,1);
c = rand(2,1);
d = zeros(nElements,1);
d_gpu = feval(kern,d,a,b,c,nElements);
d = gather(d_gpu);
```

And, finally, the CUDA kernel itself in the file "myCuFile.cu":

```
// Subfunction for calculating each individual element in d
// ------------------------------------------------------------
__host__ __device__ void calc_d(double *a, double *b,
                                double *c, double *d)
{
    *d = c[0]*(*a)+c[1]*(*b);
}

// The kernel function (this is the entry point)
// ------------------------------------------------
__global__ void kernelFun(double *d, double *a,
                          double *b, double *c,
                          double nElements)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    calc_d(&a[i], &b[i], c, &d[i]);
}
```

References

[1] *Wikipedia, the Free Encyclopedia.* 2012, August 8. http://www.wikipedia.org.

[2] M. Born and E. Wolf. *Principles of Optics.* Cambridge University Press, Cambridge, 1999.

[3] H. Gross, editor. *Handbook of Optical Systems - Fundamentals of Technical Optics.* Wiley, Weinheim, 2005.

[4] MathWorks. *Parallel Computing Toolbox – Documentation.* 2012. http://www.mathworks.com/help/toolbox/distcomp.

[5] NVIDIA. *NVIDIA CUDA C Getting Started Guide for Linux.* 2011. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Getting_Started_Linux.pdf.

[6] NVIDIA. *NVIDIA CUDA C Getting Started Guide for Microsoft Windows*. 2011. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Getting_Started_Windows.pdf.

[7] N. Wengert and P. Eberhard. Using Dynamic Stress Recovery to Investigate Stress Effects in the Dynamics of Optical Lenses. In *Proceedings of the 7th ICCSM*, Zadar, Croatia, 2012.