Soft matter: Density functionals and simulation of simple molecules on graphics cards

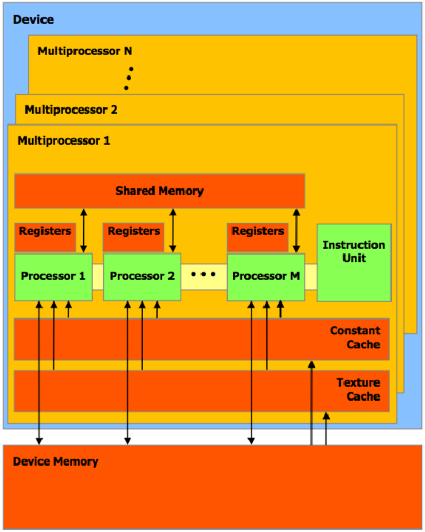
Marlon Ebert

Institut für Physik, Johannes Gutenberg-Universität Mainz

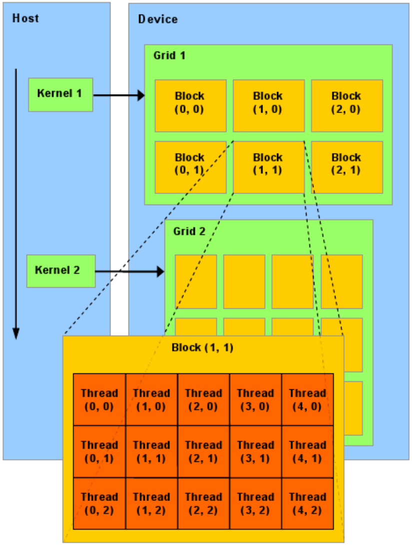
November 25, 2009

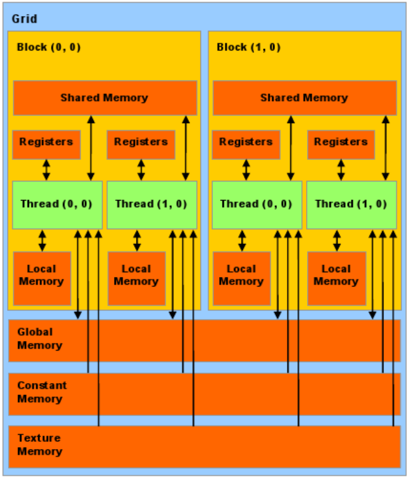
Outline

Physical nVidia GPU architecture ...



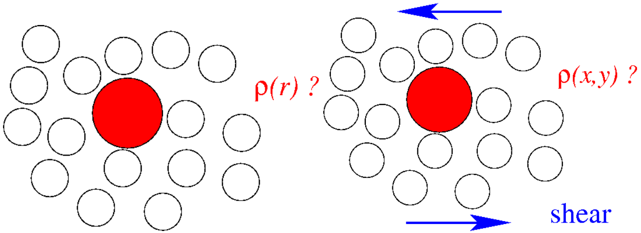
... and how it looks from a programmers perspective





What we want to do

- Calculate pair correlation functions and density profiles around a test particle in two dimensions.



- Equilibrium: $\rho(\mathbf{r}) = \rho_0 \; g(\mathbf{r})$.
- Under shear: $\rho(x,y) = \rho_0 \; g(x,y)$ (which gives us rheological information).

The basic concept of classical DFT

- Given the full free energy functional $F[\rho]$ of the system the density profile $\rho(\mathbf{r})$ is found by minimising

$$\Omega = F[\rho] - \int d\mathbf{r}\ \rho(\mathbf{r})(\mu - V(\mathbf{r})).$$

- $F[\rho]$ can be expressed by the sum of the ideal gas free energy and the excess free energy

$$F[\rho] = F^{id}[\rho] + F^{ex}[\rho].$$

- For $\rho = \rho_{eq}$ (equilibrium density) Ω is the grand potential.

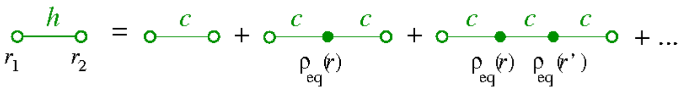
Density functional for hard discs

- Ideal gas and tailor expanded functional:

$$F^{id} = \int d\mathbf{r}\ \rho(\mathbf{r}) \cdot ln\left(\rho_0 - 1\right),$$

$$F^{ex} = \int d\mathbf{r}\ \mu\ \rho(\mathbf{r}) - \frac{1}{2}\int d\mathbf{r}\ d\mathbf{r'}\ c^{(2)}(\mathbf{r}-\mathbf{r'};\rho_0)\ \Delta\rho(\mathbf{r})\ \Delta\rho(\mathbf{r'}).$$

$c^{(2)}(\mathbf{r}-\mathbf{r'};\rho_0)$ is the direct correlation function.



- Differentiation of Ω leads to

$$\rho(\mathbf{r}) = \rho_0 \cdot \exp\left[-V(\mathbf{r}) + c^{(2)} * \Delta\rho\right].$$

Identifying the main computational steps

- Solution by iterating $\rho_{i+1}(\mathbf{r}) = \rho_0 e^{-V(\mathbf{r})+c^{(2)}*\Delta\rho_i}$.
- Calculation will take place on a 2D lattice of size $n \times n$.
- $V(\mathbf{r})$ constant potential, infinite in the area occupied by our test particle and 0 everywhere else.
- $c^{(2)}$ constant and the fourier transformed values can be stored in the graphic cards memory.
- $\Delta\rho = \rho(\mathbf{r}) - \rho_0$ has to be calculated each step.
- The convolution $c^{(2)} * \Delta\rho$ will be calculated in fourier space, making fourier transformations necessary. $\Delta\rho$ has to be calculated in each step. The fourier transformations will be the most time consuming part of the calculation.

The Fast Fourier Transformation (FFT)

- The direct evaluation of the fourier transformation $X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k \frac{n}{N}}$ runs in $\mathrm{O}(N^2)$.
- Cooley-Tukey FFT algorithm divides one fourier transformation of size N into two of size $N/2$

$$
\begin{aligned}
X_k &= \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-i2\pi k \frac{2m}{N}} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-i2\pi k \frac{2m+1}{N}} \\
&= \sum_{m=0}^{M-1} x_{2m} e^{-i2\pi k \frac{m}{M}} + e^{-\frac{i2\pi}{N}k} \sum_{m=0}^{M-1} x_{2m+1} e^{-i2\pi k \frac{m}{M}} \\
&= \begin{cases} E_k + e^{-\frac{i2\pi}{N}k} O_k & k < M, \\ E_{k-M} - e^{-\frac{i2\pi}{N}(k-M)} O_{k-M} & k \geq M. \end{cases}
\end{aligned}
$$

using $E_{k+M} = E_k, O_{k+M} = O_k$ even and odd FT.

- Recursive application leads to runtime of $\mathrm{O}(N \log N)$

The hardware and software

- CPU: Intel Core2 Quad CPU Q6700 @ 2.66GHz
- GPU: NVidia GeForce GTX 280
- NVidia Driver Version: 185.18.14
- X Server Version: 1.5.2 (11)
- Operating System: SUSE Linux 11.1, 64 bit
- Kernel Version: 2.6.27.29

Implementation (1)

- The CUFFT library is used for the fourier transformations.

```
#include <cufft.h>

__device__ Complex* d_drho;
__device__ Complex* d_fold;
__device__ cufftHandle plan;

cufftPlan2d(&plan, size_x, size_y, CUFFT_C2C);

cufftExecC2C(plan, (cufftComplex *)d_drho, (cufftComplex *)d_drho, CUFFT_FORWARD);
cufftExecC2C(plan, (cufftComplex *)d_fold, (cufftComplex *)d_fold, CUFFT_INVERSE);

cufftDestroy(plan);
cudaFree(d_fold);
cudaFree(d_drho);
```

- CUFFT is inspired by the popular FFTW C library.
- CUBLAS (inspired by the popular BLAS C library) is also available but not shown here.

Implementation (2)

- Evaluating the exponential function requires point-wise matrix manipulation and can easily be parallelized.

```
#include <cufft.h>

__global__ void function(cufftComplex* res, cufftComplex* v, cufftComplex* fold,
     float r, int sx, int sy) {
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (X < sx && Y < sy) {
        res[pos].x = r * exp(- v[pos].x + fold[pos].x);
        res[pos].y = 0;
    }
}


void iterate_gpu() {
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    int GRID_X = (int) ceil((float) size_x/(float) BLOCK_SIZE);
    int GRID_Y = (int) ceil((float) size_y/(float) BLOCK_SIZE);
    dim3 dimGrid(GRID_X, GRID_Y);
    [...]
    function<<<dimGrid,dimBlock>>>(d_fold, d_v, d_fold, rho0, size_x, size_y);
    [...]
}
```

Implementation (3)

- For convergence we need the difference between two iterations.

```
__global__ void compare_fast(cufftComplex* a, cufftComplex* b, int sx, int sy,
     float dx, float r)
{
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (pos < MAX_BLOCK)
    {
        __shared__ float c[MAX_BLOCK];
        c[pos] = a[pos + (int) (r/dx)].x - b[pos + (int) (r/dx)].x;
        c[pos] *= c[pos];
        c[pos] /= (float) MAX_BLOCK;
        __syncthreads();
        for (int s=MAX_BLOCK/2; s>0; s/=2)
        {
            if (pos < s)
            {
                c[pos] += c[pos + s];
            }
            __syncthreads();
        }
        if (pos == 0)
            {
                a[0].x = c[0];
                a[0].y = 0;
            }
    }
}
```

Radial distribution function $g(r)$

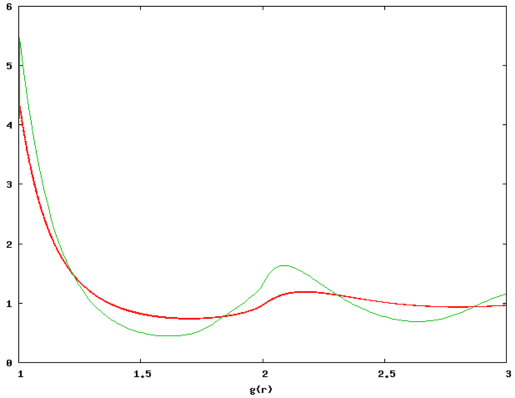


Figure: Comparison between DFT (red) and MC simulation (green) at a density of $\rho_0 = 0.65$. More accurate DFT results can be optained by using a better functional instead of the simple tailor expansion used in this example.

A complex model ...

- New functional with weighted densities (Rosenfeld functional):

$$F^{ex} = \int d^2r \left[-\mu_0 \cdot \ln(1-n_2) + \frac{1}{4\pi}\frac{n_1^2 - \mathbf{n}^2}{1-n_2} \right]$$

$$\text{with } n_i(\mathbf{x}) = \int \rho(\mathbf{x}') \cdot \omega_i(\mathbf{x}-\mathbf{x}') d\mathbf{x}'$$

$$\text{and } \omega_1 = \delta(R-r),\ \omega_2 = \Theta(R-r),\ \boldsymbol{\omega} = -\nabla\omega_2.$$

- New equation to iterate:

$$\rho = \rho_0 \cdot \exp\left[-\frac{\delta F}{\delta\rho} + \mu^{ex} - V^{ex} \right].$$

… leads to new challenges

- The ω_i are constant and need to be calculated once.
- The n_i have to be calculated in every time step which leads to an increase in the number of fourier transformations performed in each step.
- All these fields have to be available in the graphic cards memory, leading to a lot higher overall memory consumption.
- Maximum theoretical lattice size on currently available cards is 4096x4096 for 4GB cards, 2048x2048 has been achieved with our hardware, most runs are performed on a 1024x1024 grid.

Benchmarking

- A typical run on a 1024 x 1024 lattice:

```
#Initialization time: 208.862 ms
#FFT time: 2272.37 ms
#Function evaluation time: 277.565 ms
#loops: 377
#Overall time: 4101.56 ms
```

- Overall time is larger then the sum of the individual parts due to a "GPU boot" which has to occur before initialization.
- The ratio of fourier transformation to other calculations is about 8 : 1 which is similar to CPU calculations.
- The computation is a factor of 100 faster then a naive CPU implementation (using FFTW).
- The use of double precision has no significant impact on the result.

Outlook

- Calculations in 2D allow for the addition of shear and similar effects.
- Three body correlation functions can be efficiently calculated on the GPUs.
- Realistic results are expected using the Rosenfeld functional.
- 3D systems can be solved, memory constraints however will only allow for small grid sizes: Up to is possible 128^3 with the currently available cards while 512^3 is necessary to be able to compete with CPU calculations.