Sailfish: Lattice Boltzmann Fluid Simulations with GPUs and Python

Michał Januszewski

Institute of Physics
University of Silesia in Katowice, Poland

Google

GTC 2012
Let’s go back to 2009...

- Was working with stochastic differential equations on GPUs (google sdepy if you’re interested).
- Some previous experience with Smoothed Particle Hydrodynamics on CPUs.
- No prior knowledge of the lattice Boltzmann method.
- Started with a simple implementation in C and quickly rewrote it in Python & CUDA.
My adventure with Computational Fluid Dynamics on GPUs

Let’s go back to 2009...

- Was working with stochastic differential equations on GPUs (google sdepy if you’re interested).

Some previous experience with Smoothed Particle Hydrodynamics on CPUs.

- No prior knowledge of the lattice Boltzmann method.
- Started with a simple implementation in C and quickly rewrote it in Python & CUDA.

credit: "Piled Higher and Deeper" by Jorge Cham, www.phdcomics.com
Let’s go back to 2009...

- Was working with stochastic differential equations on GPUs (google sdepy if you’re interested).

- Some previous experience with Smoothed Particle Hydrodynamics on CPUs.
- No prior knowledge of the **lattice Boltzmann method**.
- Started with a simple implementation in C and quickly re wrote it in Python & CUDA.
Let’s go back to 2009...

- Was working with stochastic differential equations on GPUs (google sdepy if you’re interested).

Some previous experience with Smoothed Particle Hydrodynamics on CPUs.

- No prior knowledge of the lattice Boltzmann method.
- Started with a simple implementation in C and quickly rewrote it in Python & CUDA.

credit: "Piled Higher and Deeper" by Jorge Cham, www.phdcomics.com
Why did I do that, how it worked out and can you do something similar?

... with some technical details ...
1 Macroscopic scale: continuum, velocity ($\vec{v}$), pressure ($p$), Navier-Stokes equation:

$$
\rho \left( \frac{\partial \vec{v}}{\partial t} + \vec{v} \cdot \nabla \vec{v} \right) = -\nabla p + \mu \nabla^2 \vec{v} + \vec{f}
$$

2 Mesoscopic scale: particle ensemble, the lattice Boltzmann method.

$$
\frac{\partial f}{\partial t} + \vec{v} \cdot \frac{\partial f}{\partial \vec{x}} m + \frac{\partial f}{\partial \vec{p}} \cdot \vec{F} = \left. \frac{\partial f}{\partial t} \right|_{\text{coll}}
$$

3 Microscopic scale: individual molecules and atoms, molecular dynamics.
1. Macroscopic scale: continuum, velocity ($\vec{v}$), pressure ($p$), Navier-Stokes equation:

$$\rho \left( \frac{\partial \vec{v}}{\partial t} + \vec{v} \cdot \nabla \vec{v} \right) = -\nabla p + \mu \nabla^2 \vec{v} + \vec{f}$$

2. Mesoscopic scale: particle ensemble, the lattice Boltzmann method.

$$\frac{\partial f}{\partial t} + \frac{\partial f}{\partial \vec{x}} \cdot \vec{p} + \frac{\partial f}{\partial \vec{p}} \cdot \vec{F} = \frac{\partial f}{\partial t} \bigg|_{\text{coll}}$$

3. Microscopic scale: individual molecules and atoms, molecular dynamics.
Fluid simulation

1. Macroscopic scale: continuum, velocity ($\vec{v}$), pressure ($p$), Navier-Stokes equation:

$$\rho \left( \frac{\partial \vec{v}}{\partial t} + \vec{v} \cdot \nabla \vec{v} \right) = -\nabla p + \mu \nabla^2 \vec{v} + \vec{f}$$

2. Mesoscopic scale: particle ensemble, the lattice Boltzmann method.

$$\frac{\partial f}{\partial t} + \frac{\partial f}{\partial \vec{x}} \cdot \frac{\vec{p}}{m} + \frac{\partial f}{\partial \vec{p}} \cdot \vec{F} = \frac{\partial f}{\partial t} \bigg|_{\text{coll}}$$

3. Microscopic scale: individual molecules and atoms, molecular dynamics.
Lattice Boltzmann: the basics

- Discrete, regular, Cartesian grid \((i)\) is a node index).
- Mass fractions: \(f_\alpha\):
  \[ f_C, f_E, f_W, f_S, f_N, f_{NE}, f_{NW}, f_{SE}, f_{SW} \]
- Macroscopic quantities:
  \[
  \rho_i = \sum_\alpha f_\alpha(\vec{x}_i, t)
  \]
  \[
  \rho_i \vec{v}_i = \sum_\alpha \vec{c}_\alpha f_\alpha(\vec{x}_i, t)
  \]
Collision:

\[ f_\alpha^*(\vec{x}_i, t) = f_\alpha(\vec{x}_i, t) - \frac{f_\alpha(\vec{x}_i, t) - f_\alpha^{(eq)}(\rho_i, \vec{v}_i)}{\tau} \]

Streaming:

\[ f_\alpha(\vec{x}_i + \vec{c}_\alpha, t + 1) = f_\alpha^*(\vec{x}_i, t) \]
Lattice Boltzmann: the algorithm

1. Collision:

\[ f_\alpha^*(\vec{x}_i, t) = f_\alpha(\vec{x}_i, t) - \frac{f_\alpha(\vec{x}_i, t) - f_\alpha^{(eq)}(\rho_i, \vec{v}_i)}{\tau} \]

2. Streaming:

\[ f_\alpha(\vec{x}_i + \vec{c}_\alpha, t + 1) = f_\alpha^*(\vec{x}_i, t) \]
Lattice Boltzmann: the algorithm

1. Collision:

\[ f^*_\alpha(\vec{x}_i, t) = f_\alpha(\vec{x}_i, t) - \frac{f_\alpha(\vec{x}_i, t) - f^{(eq)}_\alpha(\rho_i, \vec{v}_i)}{\tau} \]

2. Streaming:

\[ f_\alpha(\vec{x}_i + \vec{c}_\alpha, t + 1) = f^*_\alpha(\vec{x}_i, t) \]
Why lattice Boltzmann?

- Applicable for low Mach number flows.
- Good for flows in complex domains (e.g. porous materials).
- Extremely well parallelizable (nearest-neighbour interactions).
- Easy to implement.

Image credit: EXA Corp.
Figure: Papers with "lattice Boltzmann" in the title (source: Scopus)
What is Sailfish?

- GPU-based implementation of the lattice Boltzmann method.
- Open source (LGPL v3).
- Implemented using Python and CUDA C / OpenCL.
- Written from scratch.
- Under development for approximately 3 years.

Image credit: Wikimedia

http://sailfish.us.edu.pl
Why Python?

- Easy to understand.
- Very expressive (get stuff done quickly).
- Great support for GPU programming (via PyCUDA/PyOpenCL).
- Bindings with many system libraries.

... but also **too slow** for large-scale numerical work.
"The boring stuff" (initialization, I/O, etc) becomes essentially free.

Use metaprogramming ("programs which write other programs") to:
  * generate optimized code on a case-by-case basis,
  * explore parameter spaces to find optimal solutions,
  * provide isolation from hardware details.

Possible realizations:
  * Abstract Syntax Trees.
  * Domain-specific languages.
  * Template-based code generation.
Metaprogramming one step further: computer algebra systems

- Numerical code initially described as formulas on paper.
- Computer code often repetitive.
- Write formulas directly in your program and generate code automatically.

New possibilities:
- Consistency checks at the level of mathematics.
- Code is documentation.
- Transform formulas prior to generating compilable code.
Many lattice Boltzmann models which differ in:
- lattice connectivity / dimension
- collision operator
- equilibrium function
- turbulence models
- ...

Many formulas are independent of (some of) these details.

RTCG makes it possible to easily experiment with all of these.
LBM as a framework

- Many lattice Boltzmann models which differ in:
  - lattice connectivity / dimension
  - collision operator
  - equilibrium function
  - turbulence models
  - ...
- Many formulas are independent of (some of) these details.
- RTCG makes it possible to easily experiment with all of these.
LBM as a framework

- Many lattice Boltzmann models which differ in:
  - lattice connectivity / dimension
  - collision operator
  - equilibrium function
  - turbulence models
  - ...
  - Many formulas are independent of (some of) these details.
  - RTCG makes it possible to easily experiment with all of these.
LBM as a framework

- Many lattice Boltzmann models which differ in:
  - lattice connectivity / dimension
  - collision operator

\[
\text{BGK: } \frac{|f_i \rangle - |f_i^{eq} \rangle}{\tau}
\]

- equilibrium function
- turbulence models
- ...

- Many formulas are independent of (some of) these details.
- RTCG makes it possible to easily experiment with all of these.
LBM as a framework

- Many lattice Boltzmann models which differ in:
  - lattice connectivity / dimension
  - collision operator

\[
\text{MRT: } M^{-1} S \left( M |f_i\rangle - |m_{eq_i}\rangle \right)
\]

- equilibrium function
- turbulence models
- ...

- Many formulas are independent of (some of) these details.
- RTCG makes it possible to easily experiment with all of these.
Many lattice Boltzmann models which differ in:
- lattice connectivity / dimension
- collision operator
- equilibrium function
- turbulence models
- ...

Many formulas are independent of (some of) these details.
RTCG makes it possible to easily experiment with all of these.
LBM as a framework

- Many lattice Boltzmann models which differ in:
  - lattice connectivity / dimension
  - collision operator
  - equilibrium function
  - turbulence models
  - ...

- Many formulas are independent of (some of) these details.

- RTCG makes it possible to easily experiment with all of these.
Many lattice Boltzmann models which differ in:
- lattice connectivity / dimension
- collision operator
- equilibrium function
- turbulence models
- ...

Many formulas are independent of (some of) these details.

RTCG makes it possible to easily experiment with all of these.
LBM as a framework

- Many lattice Boltzmann models which differ in:
  - lattice connectivity / dimension
  - collision operator
  - equilibrium function
  - turbulence models
  - ...

- Many formulas are independent of (some of) these details.

- RTCG makes it possible to easily experiment with all of these.
Many lattice Boltzmann models which differ in:
- lattice connectivity / dimension
- collision operator
- equilibrium function
- turbulence models
- ...

Many formulas are independent of (some of) these details.

**RTCG makes it possible to easily experiment with all of these.**
Mako code:

```python
${device_func} inline void bounce_back(Dist *fi)
{
    float t;

    %for i in sym.bb_swap_pairs(grid):
        t = fi->${grid.idx_name[i]};
        fi->${grid.idx_name[i]} = fi->${grid.idx_name[grid.idx_opposite[i]]};
        fi->${grid.idx_name[grid.idx_opposite[i]]} = t;
    %endfor
}
```
CUDA C code, D2Q9 grid:

```c
__device__ inline void bounce_back(Dist * fi)
{
    float t;
    t = fi->fE;
    fi->fE = fi->fW;
    fi->fW = t;
    t = fi->fN;
    fi->fN = fi->fS;
    fi->fS = t;
    t = fi->fNE;
    fi->fNE = fi->fSW;
    fi->fSW = t;
    t = fi->fNW;
    fi->fNW = fi->fSE;
    fi->fSE = t;
}
```
CUDA C code, D2Q9 grid:

```c
__device__ inline void bounce_back(Dist * fi)
{
    float t;
    t = fi->fE;
    fi->fE = fi->fW;
    fi->fW = t;
    t = fi->fN;
    fi->fN = fi->fS;
    fi->fS = t;
    t = fi->fNE;
    fi->fNE = fi->fSW;
    fi->fSW = t;
    t = fi->fNW;
    fi->fNW = fi->fSE;
    fi->fSE = t;
}
```
CUDA C code, D3Q13 grid:

```c
__device__ inline void bounce_back(Dist * fi)
{
    float t;
    t = fi->fNE;
    fi->fNE = fi->fSW;
    fi->fSW = t;
    t = fi->fSE;
    fi->fSE = fi->fNW;
    fi->fNW = t;
    t = fi->fTE;
    fi->fTE = fi->fBW;
    fi->fBW = t;
    t = fi->fBE;
    fi->fBE = fi->fTW;
    fi->fTW = t;
    ...
    t = fi->fTN;
    fi->fTN = fi->fBS;
    fi->fBS = t;
    t = fi->fBN;
    fi->fBN = fi->fTS;
    fi->fTS = t;
}
```
Collision step of the LB algorithm:

\[
f^{*}_\alpha(x_i, t) = f_\alpha(x_i, t) - \frac{f_\alpha(x_i, t) - f^{(eq)}_\alpha(\rho_i, v_i)}{\tau}
\]

with

\[
f^{(eq)}_\alpha(\rho_i, v_i) = w_\alpha \rho \left( 1 + 3 \vec{c}_\alpha \cdot \vec{v}_i + \frac{9}{2} (\vec{c}_\alpha \cdot \vec{v}_i)^2 - \frac{3}{2} \vec{v}_i^2 \right)
\]
\[ f^{(eq)}_{\alpha}(\rho_i, \vec{v}_i) = w_{\alpha}\rho\left(1 + 3\vec{c}_\alpha \cdot \vec{v}_i + \frac{9}{2}(\vec{c}_\alpha \cdot \vec{v}_i)^2 - \frac{3}{2}\vec{v}_i^2\right) \]

```python
def bgk_equilibrium(grid, rho=None):
    out = []

    if rho is None:
        rho = S.rho

    for i, ei in enumerate(grid.basis):
        t = (grid.weights[i] * rho * (1 + 3*ei.dot(grid.v) + Rational(9, 2) * (ei.dot(grid.v))**2 - Rational(3, 2) * grid.v.dot(grid.v)))

        out.append(t)

    return out
```
\begin{align*}
  f_{eq}^{(eq)}(\rho_i, \vec{v}_i) &= w_\alpha \rho \left( 1 + 3 \vec{c}_\alpha \cdot \vec{v}_i + \frac{9}{2} (\vec{c}_\alpha \cdot \vec{v}_i)^2 - \frac{3}{2} \vec{v}_i^2 \right) \\
\end{align*}

```python
def bgk_equilibrium(grid, rho=None):
    out = []

    if rho is None:
        rho = S.rho

    for i, ei in enumerate(grid.basis):
        t = (grid.weights[i] * rho * (1 +
            3*ei.dot(grid.v) +
            Rational(9, 2) * (ei.dot(grid.v))**2 -
            Rational(3, 2) * grid.v.dot(grid.v)))

        out.append(t)

    return out
```
\[ f^{(eq)}_{\alpha}(\rho_i, \vec{v}_i) = w_{\alpha} \rho \left( 1 + 3 \vec{c}_{\alpha} \cdot \vec{v}_i + \frac{9}{2} (\vec{c}_{\alpha} \cdot \vec{v}_i)^2 - \frac{3}{2} \vec{v}_i^2 \right) \]

```python
def bgk_equilibrium(grid, rho=None):
    out = []
    if rho is None:
        rho = S.rho
    for i, ei in enumerate(grid.basis):
        t = (grid.weights[i] * rho * (1 +
            3*ei.dot(grid.v) +
            Rational(9, 2) * (ei.dot(grid.v))**2 -
            Rational(3, 2) * grid.v.dot(grid.v)))
        out.append(t)
    return out
```
\[ f^{(eq)}_{\alpha}(\rho_i, \vec{v}_i) = w_{\alpha} \rho \left( 1 + 3 \vec{c}_{\alpha} \cdot \vec{v}_i + \frac{9}{2} (\vec{c}_{\alpha} \cdot \vec{v}_i)^2 - \frac{3}{2} \vec{v}_i^2 \right) \]

```python
def bgk_equilibrium(grid, rho=None):
    out = []

    if rho is None:
        rho = S.rho

    for i, ei in enumerate(grid.basis):
        t = (grid.weights[i] * rho * (1 +
            3*ei.dot(grid.v) +
            Rational(9, 2) * (ei.dot(grid.v))**2 -
            Rational(3, 2) * grid.v.dot(grid.v)))

        out.append(t)

    return out
```
\[ f_{\alpha}^{(eq)}(\rho_i, \vec{v}_i) = w_\alpha \rho \left( 1 + 3 \vec{c}_\alpha \cdot \vec{v}_i + \frac{9}{2} (\vec{c}_\alpha \cdot \vec{v}_i)^2 - \frac{3}{2} \vec{v}_i^2 \right) \]

```python
def bgk_equilibrium(grid, rho=None):
    out = []

    if rho is None:
        rho = S.rho

    for i, ei in enumerate(grid.basis):
        t = (grid.weights[i] * rho * (1 +
            3*ei.dot(grid.v) +
            Rational(9, 2) * (ei.dot(grid.v))**2
            Rational(3, 2) * grid.v.dot(grid.v)))
        out.append(t)

    return out
```
Sailfish: symbolic run-time code generation example

\[ f^{(eq)}_{\alpha}(\rho_i, \vec{v}_i) = w_\alpha \rho \left( 1 + 3 \vec{c}_\alpha \cdot \vec{v}_i + \frac{9}{2} (\vec{c}_\alpha \cdot \vec{v}_i)^2 - \frac{3}{2} \frac{v_i^2}{v_i} \right) \]

\[
\begin{align*}
feq0.fC &= 4 * \rho / 9 + 4 * \rho \times (-3 * v0[0] * v0[0] / 2 - 3 * v0[1] * v0[1] / 2) / 9; \\
feq0.fE &= \rho / 9 + \rho \times (3 * v0[0] \times (1 + v0[0]) - 3 * v0[1] * v0[1] / 2) / 9; \\
feq0.fN &= \rho / 9 + \rho \times (3 * v0[1] \times (1 + v0[1]) - 3 * v0[0] * v0[0] / 2) / 9; \\
feq0.fW &= \rho / 9 + \rho \times (-3 * v0[0] \times (1 - v0[0]) - 3 * v0[1] * v0[1] / 2) / 9; \\
feq0.fS &= \rho / 9 + \rho \times (-3 * v0[1] \times (1 - v0[1]) - 3 * v0[0] * v0[0] / 2) / 9; \\
feq0.fNE &= \rho / 36 + \rho \times (3 * v0[0] \times (1 + v0[0]) + 3 * v0[1] \times (1 + v0[1] + 3 * v0[0])) / 36; \\
feq0.fNW &= \rho / 36 + \rho \times (-3 * v0[0] \times (1 - v0[0]) + 3 * v0[1] \times (1 + v0[1] - 3 * v0[0])) / 36; \\
feq0.fSW &= \rho / 36 + \rho \times (-3 * v0[0] \times (1 - v0[0]) - 3 * v0[1] \times (1 - v0[1] - 3 * v0[0])) / 36; \\
feq0.fSE &= \rho / 36 + \rho \times (-3 * v0[1] \times (1 - v0[1] + 3 * v0[0]) + 3 * v0[0] \times (1 + v0[0])) / 36;
\end{align*}
\]
\[ f^{(eq)}_{\alpha}(\rho, \vec{v}) = w_{\alpha}^{\rho} \left( 1 + 3c_{\alpha} \cdot \vec{v} + \frac{9}{2}(c_{\alpha} \cdot \vec{v})^2 - \frac{3}{2}v^2 \right) \]

\begin{align*}
feq0.fC &= \frac{\rho}{3} + \rho \left( -3 * v_0[0] * v_0[0] / 2 - 3 * v_0[1] * v_0[1] / 2 - 3 * v_0[2] * v_0[2] / 2 \right) / 3; \\
feq0.fE &= \frac{\rho}{18} + \rho \left( 3 * v_0[0] * (1 + v_0[0]) - 3 * v_0[1] * v_0[1] / 2 - 3 * v_0[2] * v_0[2] / 2 \right) / 18; \\
feq0.fW &= \frac{\rho}{18} + \rho \left( -3 * v_0[0] * (1 - v_0[0]) - 3 * v_0[1] * v_0[1] / 2 - 3 * v_0[2] * v_0[2] / 2 \right) / 18; \\
feq0.fN &= \frac{\rho}{18} + \rho \left( 3 * v_0[1] * (1 + v_0[1]) - 3 * v_0[0] * v_0[0] / 2 - 3 * v_0[2] * v_0[2] / 2 \right) / 18; \\
feq0.fS &= \frac{\rho}{18} + \rho \left( -3 * v_0[1] * (1 - v_0[1]) - 3 * v_0[0] * v_0[0] / 2 - 3 * v_0[2] * v_0[2] / 2 \right) / 18; \\
feq0.fT &= \frac{\rho}{18} + \rho \left( 3 * v_0[2] * (1 + v_0[2]) - 3 * v_0[0] * v_0[0] / 2 - 3 * v_0[1] * v_0[1] / 2 \right) / 18; \\
feq0.fB &= \frac{\rho}{18} + \rho \left( -3 * v_0[2] * (1 - v_0[2]) - 3 * v_0[0] * v_0[0] / 2 - 3 * v_0[1] * v_0[1] / 2 \right) / 18; \\
feq0.fNE &= \frac{\rho}{36} + \rho \left( 3 * v_0[0] * (1 + v_0[0]) + 3 * v_0[1] * (1 + v_0[1] + 3 * v_0[0]) - 3 * v_0[2] + v_0[2] / 2 \right) / 36; \\
feq0.fNW &= \frac{\rho}{36} + \rho \left( -3 * v_0[0] * (1 - v_0[0]) + 3 * v_0[1] * (1 + v_0[1] - 3 * v_0[0]) - 3 * v_0[2] * v_0[2] / 2 \right) / 36; \\
feq0.fSE &= \frac{\rho}{36} + \rho \left( -3 * v_0[1] * (1 - v_0[1] + 3 * v_0[0]) + 3 * v_0[0] * (1 + v_0[0]) - 3 * v_0[2] * v_0[2] / 2 \right) / 36; \\
feq0.fSW &= \frac{\rho}{36} + \rho \left( -3 * v_0[0] * (1 - v_0[0]) - 3 * v_0[1] * (1 - v_0[1] - 3 * v_0[0]) - 3 * v_0[2] * v_0[2] / 2 \right) / 36;
\end{align*}
Sailfish: Run-Time Code Generation

Sailfish uses template-based Run-Time Code Generation (RTCG).

- **Code is readable (education!)**
- Code is optimized for specific simulation cases.
- Many formulas stored in symbolic form (SymPy expressions) instead of executable code.
  - Prevents developers from making silly mistakes.
  - Easier to read.
  - Automated consistency checks.
- Possibility to auto-tune.
- Think: flexibility of Mathematica with the performance of C.
Primary ideas of the project:

- **Use Run-Time Code Generation to automatically generate optimized code.**
- **Allow for fast calculations using Python (no performance compromises!)**
- **Encourage experimentation.**
- **Eliminate sources of error.**
- **Minimize the use of programmer time.**
Sailfish currently supports:

- **Distributed multi-GPU** simulations.
- Single and double precision calculations.
- **Multiple LB models** (2D, 3D; BGK, MRT, entropic; single fluid, binary fluids, ...)
- Multiple output formats (NumPy, MatLab, VTK, ...)
- CUDA and OpenCL backends.
Sample simulations

M. Januszewski (IoP, US)
Sailfish: LBM with GPUs and Python
GTC 2012 19 / 26
How it all works: defining a simulation

```python
class RayleighTaylorDomain(Subdomain2D):
    def boundary_conditions(self, hx, hy):
        self.set_node(np.logical_or(hy == 0, hy == self.gy - 1),
                      self.NODE_WALL)

def initial_conditions(self, sim, hx, hy):
    sim.rho[:] = np.random.rand(*sim.rho.shape) / 100.0
    sim.phi[:] = np.random.rand(*sim.phi.shape) / 100.0
    sim.rho[(hy <= self.gy / 2)] += 1.0
    sim.phi[(hy <= self.gy / 2)] = 1e-4
    sim.rho[(hy > self.gy / 2)] = 1e-4
    sim.phi[(hy > self.gy / 2)] += 1.0

...```

```python
class RayleighTaylorSCSim(LBBinaryFluidShanChen, LBForcedSim):
    subdomain = RayleighTaylorDomain

    @classmethod
    def update_defaults(cls, defaults):
        defaults.update({'lat_nx': 640,
                         'lat_ny': 400,
                         'grid': 'D2Q9',
                         'G': 1.2,
                         'visc': 1.0 / 6.0,
                         'periodic_x': True})

    @classmethod
    def modify_config(cls, config):
        config.tau_phi = sym.relaxation_time(config.visc)

def __init__(self, config):
    super(RayleighTaylorSCSim, self).__init__(config)
    self.add_body_force((0.0, -0.15 / config.lat_ny), grid=1)

if __name__ == '__main__':
    ctrl = LBSimulationController(RayleighTaylorSCSim, LBGeometry2D)
    ctrl.run()
```
How it all works: simulation setup and code generation

1. Start a **controller** process.
2. Decompose domain into subdomains (cuboids).
3. Start a **master** process on each computational node.
4. Start **subdomain handlers** on each computational node (one process per domain).
5. Each handler:
   - sets initial conditions via macroscopic fields (numpy arrays),
   - generates CUDA code based on the features used in its subdomain,
   - executes the main loop.
Store mass fractions in a structure of arrays in global memory. Two lattices (A and B).

1 node – 1 GPU thread, arranged in 1D block:
- Aligned memory access as mass fractions are loaded into registers from lattice A.
- Relaxation fully local using registers.
- Write data to lattice B in global memory.

In the next iteration the role of A and B is reversed.
How it all works: LBM implementation on the GPU

- Store mass fractions in a structure of arrays in global memory. Two lattices (A and B).
- 1 node – 1 GPU thread, arranged in 1D block:
  - Aligned memory access as mass fractions are loaded into registers from lattice A.
  - Relaxation fully local using registers.
  - Write data to lattice B in global memory.
- In the next iteration the role of A and B is reversed.
Store mass fractions in a structure of arrays in global memory. Two lattices (A and B).

1 node – 1 GPU thread, arranged in 1D block:

- Aligned memory access as mass fractions are loaded into registers from lattice A.
- Relaxation fully local using registers.
- Write data to lattice B in global memory.

In the next iteration the role of A and B is reversed.
How it all works: main loop

Idea: Overlap network I/O and GPU computation.

- Split domain into **boundary** and **bulk**.
- Run simulation in the boundary first.
- Run kernels to collect data into a continuous memory block.
- Run simulation in the bulk area.
  - Copy data to be transferred from the GPU to the host.
  - Send data to remote nodes.
  - Receive data from remote nodes.
  - Copy data from the host to the GPU.
- Run kernels to distribute data from remote nodes to the correct locations in global memory.

GPU buffer in global mem.
How it all works: main loop

Idea: Overlap network I/O and GPU computation.

- Split domain into **boundary** and **bulk**.
- **Run simulation in the boundary first.**
  - Run kernels to collect data into a continuous memory block.
  - Run simulation in the bulk area.
    - Copy data to be transferred from the GPU to the host.
    - Send data to remote nodes.
    - Receive data from remote nodes.
    - Copy data from the host to the GPU.
- **Run kernels to distribute data from remote nodes to the correct locations in global memory.**
How it all works: main loop

Idea: Overlap network I/O and GPU computation.

- Split domain into **boundary** and **bulk**.
- Run simulation in the boundary first.
- Run kernels to collect data into a continuous memory block.
- Run simulation in the bulk area.
  - Copy data to be transferred from the GPU to the host.
  - Send data to remote nodes.
  - Receive data from remote nodes.
  - Copy data from the host to the GPU.
- Run kernels to distribute data from remote nodes to the correct locations in global memory.
How it all works: main loop

Idea: Overlap network I/O and GPU computation.
- Split domain into **boundary** and **bulk**.
- Run simulation in the boundary first.
- Run kernels to collect data into a continuous memory block.
- Run simulation in the bulk area.
  - Copy data to be transferred from the GPU to the host.
  - Send data to remote nodes.
  - Receive data from remote nodes.
  - Copy data from the host to the GPU.
- Run kernels to distribute data from remote nodes to the correct locations in global memory.
How it all works: main loop

Idea: Overlap network I/O and GPU computation.

- Split domain into **boundary** and **bulk**.
- Run simulation in the boundary first.
- Run kernels to collect data into a continuous memory block.
- Run simulation in the bulk area.
  - Copy data to be transferred from the GPU to the host.
  - Send data to remote nodes.
  - Receive data from remote nodes.
  - Copy data from the host to the GPU.
- Run kernels to distribute data from remote nodes to the correct locations in global memory.
How it all works: main loop

Idea: Overlap network I/O and GPU computation.

- Split domain into **boundary** and **bulk**.
- Run simulation in the boundary first.
- Run kernels to collect data into a continuous memory block.
- Run simulation in the bulk area.
  - Copy data to be transferred from the GPU to the host.
  - Send data to remote nodes.
  - Receive data from remote nodes.
  - Copy data from the host to the GPU.
- Run kernels to distribute data from remote nodes to the correct locations in global memory.

GPU buffer in global mem.
How it all works: node-node communication

Controller

execnet

Master

ØMQ

Subdomain handler

Subdomain handler

ØMQ

Master

ØMQ

Subdomain handler

Subdomain handler

ØMQ

ØMQ

ØMQ

M. Januszewski (IoP, US)

Sailfish: LBM with GPUs and Python

GTC 2012
Use the right tool for the job: Python + GPUs.

RTCG based on symbolic expressions is a powerful tool for building code quickly and reliably.

Programmer time more important than computer time.

With GPUs this does not necessarily mean a need to compromise on performance.
Summary

- Use the right tool for the job: Python + GPUs.
- RTCG based on symbolic expressions is a powerful tool for building code quickly and reliably.
- Programmer time more important than computer time.
- With GPUs this does not necessarily mean a need to compromise on performance.
Use the right tool for the job: Python + GPUs.

RTCG based on symbolic expressions is a powerful tool for building code quickly and reliably.

Programmer time more important than computer time.

With GPUs this does not necessarily mean a need to compromise on performance.
Use the right tool for the job: Python + GPUs.

RTCG based on symbolic expressions is a powerful tool for building code quickly and reliably.

Programmer time more important than computer time.

With GPUs this does not necessarily mean a need to compromise on performance.
Thanks for your attention. Questions?