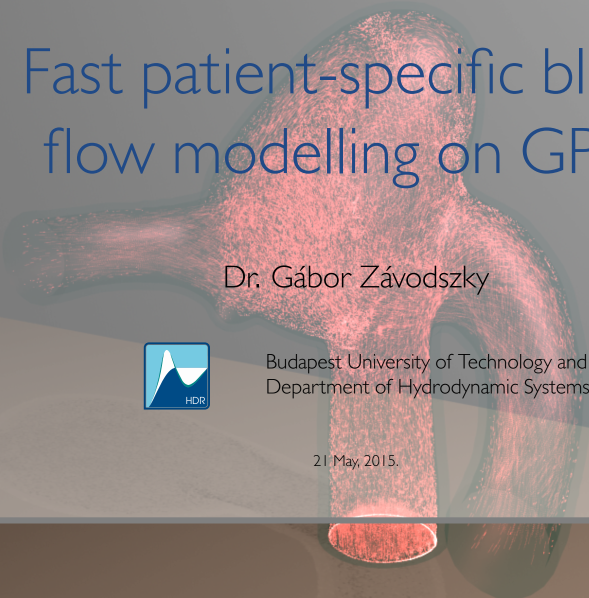
Fast patient-specific blood flow modelling on GPUs

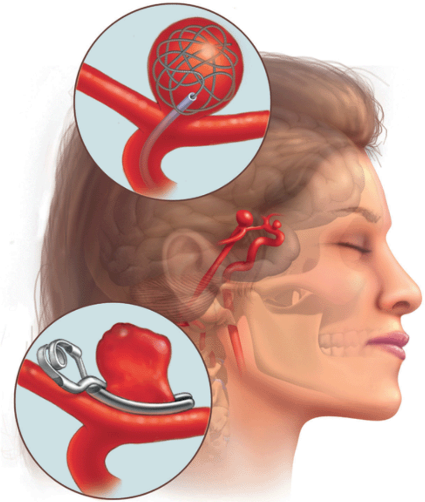


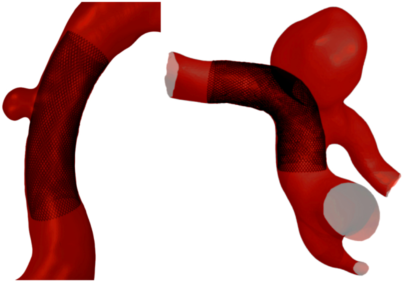
Dr. Gábor Závodszky

Budapest University of Technology and Economics
Department of Hydrodynamic Systems

21 May, 2015.

Introduction - Cerebral aneurysms

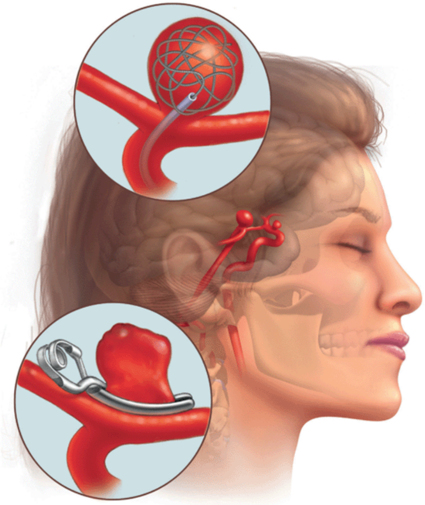


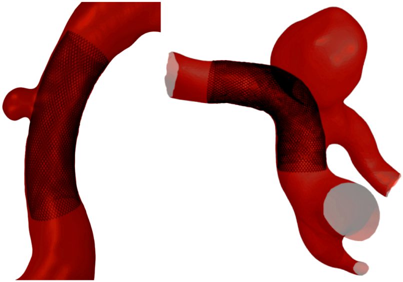


Treatment methods

- Clipping

Introduction - Cerebral aneurysms

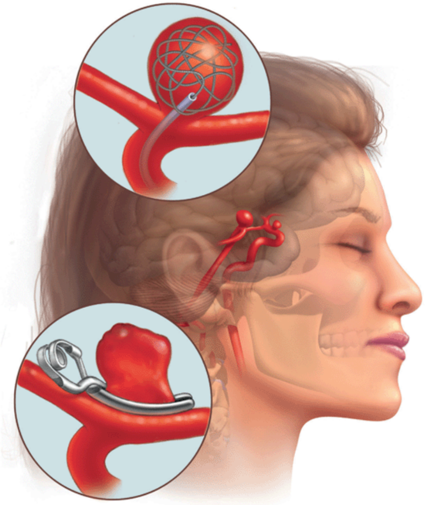


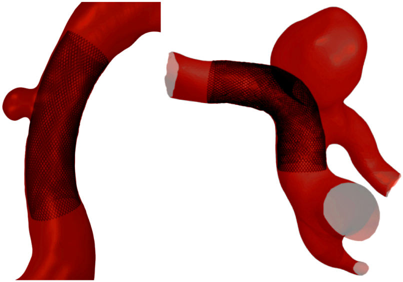


Treatment methods

- Clipping
- Coiling

Introduction - Cerebral aneurysms

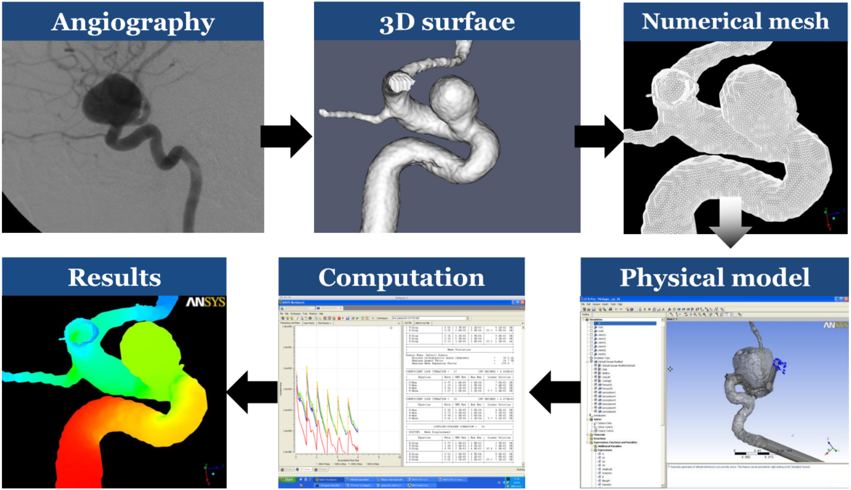




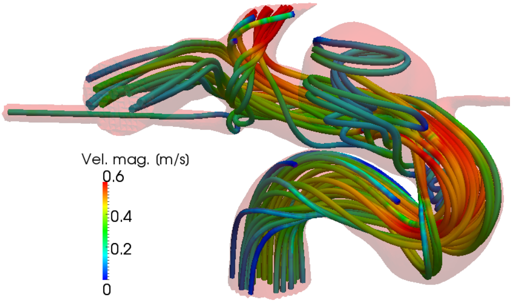
Treatment methods

- Clipping
- Coiling
- Flow diverting

Simulation toolsuite overview

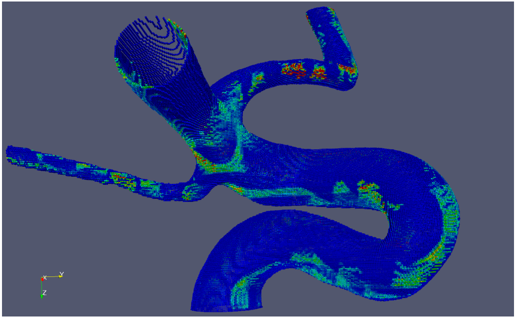


CFD simulation



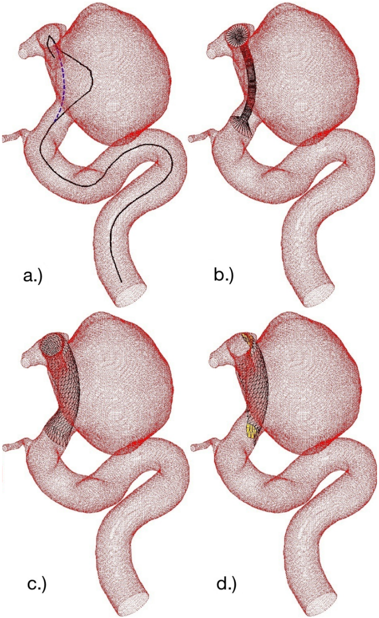
- Based on the lattice Boltzmann method.
- Highly parallel, explicit numerical scheme.
- Non-local steps are linear.
- Non-linear steps are local.

Calculation of derived quantities - PostProcess



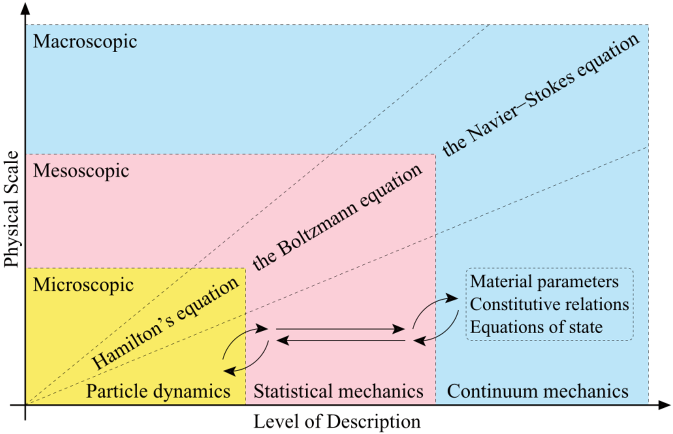
- Residence times for particles by tracing.
- Surface normals.
- Wall Shear Stress.

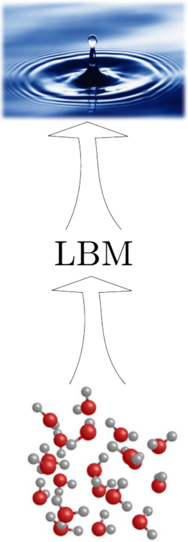
Virtual stenting



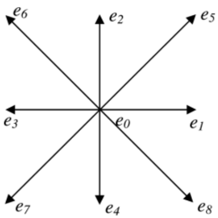
- Mass-Spring-Damper
- The dynamics is computed on the GPU
- The resulting surface is modelled as a porous layer in the CFD step.

CFD - in more details

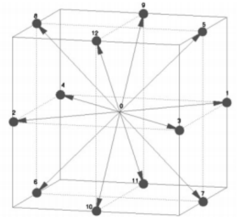




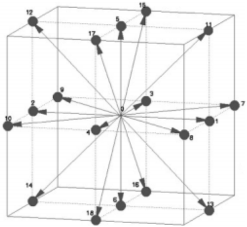
CFD - in more details II.



D2Q9



D3Q13



D3Q19

CFD - Python, Sailfish

Technologies

- (Python)

CFD - Python, Sailfish

Technologies

- (Python)
- (Mako)

CFD - Python, Sailfish

Technologies

- (Python)
- (Mako)
- (Sympy)

CFD - Python, Sailfish

Technologies

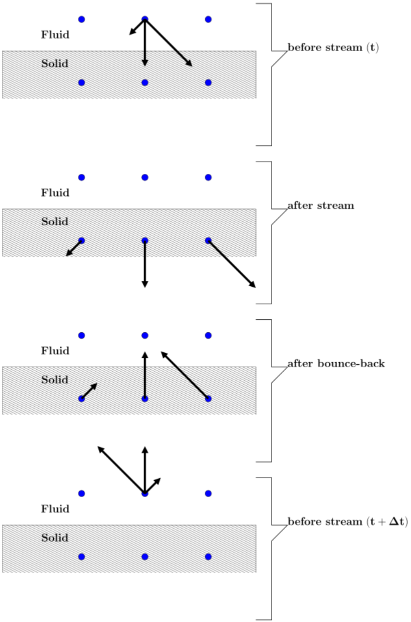
- (Python)
- (Mako)
- (Sympy)
- pyCUDA, pyOpenCL

CFD - Python, Sailfish

Technologies

- (Python)
- (Mako)
- (Sympy)
- pyCUDA, pyOpenCL
- Sailfish: Metaprogramming on GPUs!
  `https://github.com/sailfish-team/sailfish`

Bounce-back rule



- No-slip wall boundary

Bounce-back rule - Template

Mako code:

```
${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
}
```

Bounce-back rule - D2Q9

CUDA C code, D2Q9 grid:

```
__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;
}
```

Bounce-back rule - D2Q13

CUDA C code, D3Q13 grid:

```
__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;
}
```

Can we take it further?

Symbolic algebra!

Equilibrium function - Symbolic formalism

$$f_i^{eq}(\vec{x},t) = w_i\rho[1 + 3(\vec{e_i}\cdot\vec{u}) + \frac{9}{2}(\vec{e_i}\cdot\vec{u})^2 - \frac{3}{2}\vec{u}^2]$$

```
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
```

Equilibrium function - D3Q13

$$f_i^{eq}(\vec{x},t) = w_i\rho[1 + 3(\vec{e_i}\cdot\vec{u}) + \frac{9}{2}(\vec{e_i}\cdot\vec{u})^2 - \frac{3}{2}\vec{u}^2]$$

```
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
```

Equilibrium function - D3Q13

$$f_i^{eq}(\vec{x},t) = w_i\rho[1 + 3(\vec{e_i}\cdot\vec{u}) + \frac{9}{2}(\vec{e_i}\cdot\vec{u})^2 - \frac{3}{2}\vec{u}^2]$$

```
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
```

Equilibrium function - D3Q13

$$f_i^{eq}(\vec{x},t) = w_i\rho[1 + 3(\vec{e_i}\cdot\vec{u}) + \frac{9}{2}(\vec{e_i}\cdot\vec{u})^2 - \frac{3}{2}\vec{u}^2]$$

```
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
```

Equilibrium function - D3Q13

$$f_i^{eq}(\vec{x}, t) = w_i \rho [1 + 3(\vec{e_i} \cdot \vec{u}) + \frac{9}{2}(\vec{e_i} \cdot \vec{u})^2 - \frac{3}{2}\vec{u}^2]$$

```
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
```

Equilibrium function - D3Q13

$$f_i^{eq}(\vec{x},t) = w_i\rho[1 + 3(\vec{e_i}\cdot\vec{u}) + \frac{9}{2}(\vec{e_i}\cdot\vec{u})^2 - \frac{3}{2}\vec{u}^2]$$

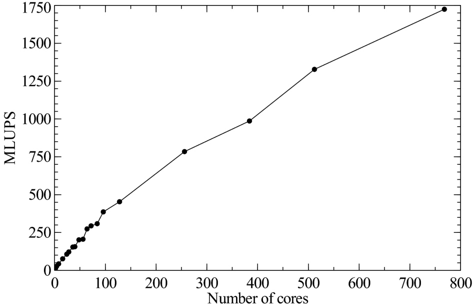
The generated code:

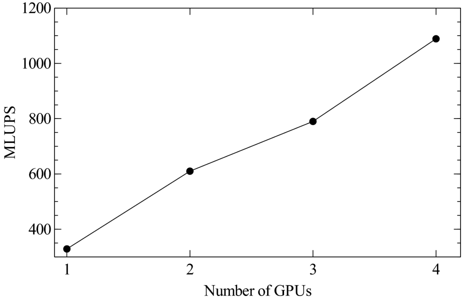
```
feq0.fC = rho / 3 + rho * (-3 * v0[0] * v0[0] / 2 - 3 * v0[1] * v0[1] / 2 - 3 * v0[2] * v0[2] / 2) / 3;
feq0.fE = rho / 18 + rho * (3 * v0[0] * (1 + v0[0]) - 3 * v0[1] * v0[1] / 2 - 3 * v0[2] * v0[2] / 2) / 18;
feq0.fW = rho / 18 + rho * (-3 * v0[0] * (1 - v0[0]) - 3 * v0[1] * v0[1] / 2 - 3 * v0[2] * v0[2] / 2) / 18;
feq0.fN = rho / 18 + rho * (3 * v0[1] * (1 + v0[1]) - 3 * v0[0] * v0[0] / 2 - 3 * v0[2] * v0[2] / 2) / 18;
feq0.fS = rho / 18 + rho * (-3 * v0[1] * (1 - v0[1]) - 3 * v0[0] * v0[0] / 2 - 3 * v0[2] * v0[2] / 2) / 18;
feq0.fT = rho / 18 + rho * (3 * v0[2] * (1 + v0[2]) - 3 * v0[0] * v0[0] / 2 - 3 * v0[1] * v0[1] / 2) / 18;
feq0.fB = rho / 18 + rho * (-3 * v0[2] * (1 - v0[2]) - 3 * v0[0] * v0[0] / 2 - 3 * v0[1] * v0[1] / 2) / 18;
feq0.fNE = rho / 36 + rho * (3 * v0[0] * (1 + v0[0]) + 3 * v0[1] * (1 + v0[1] + 3 * v0[0]) - 3 * v0[2] * v0[2] / 2) / 36;
feq0.fNW = rho / 36 + rho * (-3 * v0[0] * (1 - v0[0]) + 3 * v0[1] * (1 + v0[1] - 3 * v0[0]) - 3 * v0[2] * v0[2] / 2) / 36;
feq0.fSE = rho / 36 + rho * (-3 * v0[1] * (1 - v0[1] + 3 * v0[0]) + 3 * v0[0] * (1 + v0[0]) - 3 * v0[2] * v0[2] / 2) / 36;
feq0.fSW = rho / 36 + rho * (-3 * v0[0] * (1 - v0[0]) - 3 * v0[1] * (1 - v0[1] - 3 * v0[0]) - 3 * v0[2] * v0[2] / 2) / 36;
⋮
```

Advantages

- Closer to the mathematical formalism.
- Easier to read and modify.
- Encourages experimentation.
- Virtually no performance cost (apart from a small start-up overhead).

Scaling - CPU vs. GPU





- CPU - 768 cores
- GPU - 4 Tesla 2070
- Near-linear scaling
- Approximately one order of magnitude performance gain.
- (The fallback at the third point is due to primitive space partition).

Typical runtimes

Number of GPUs	$\sim 3M$	$\sim 6M$	$\sim 20M$
1	00:15:36	00:45:14	-
2	00:09:11	00:27:03	-
3	00:06:21	00:18:31	-
4	00:05:03	00:15:01	01:34: 22

Thank you for your attention!