cresset

smarter chemistry | smarter decisions

# Boosting RDKit molecular simulations through OpenMM

Paolo Tosco

## Outline

> Background

↓

> Implementation

↓

> Results

↓

> Conclusions and outlook

## Background

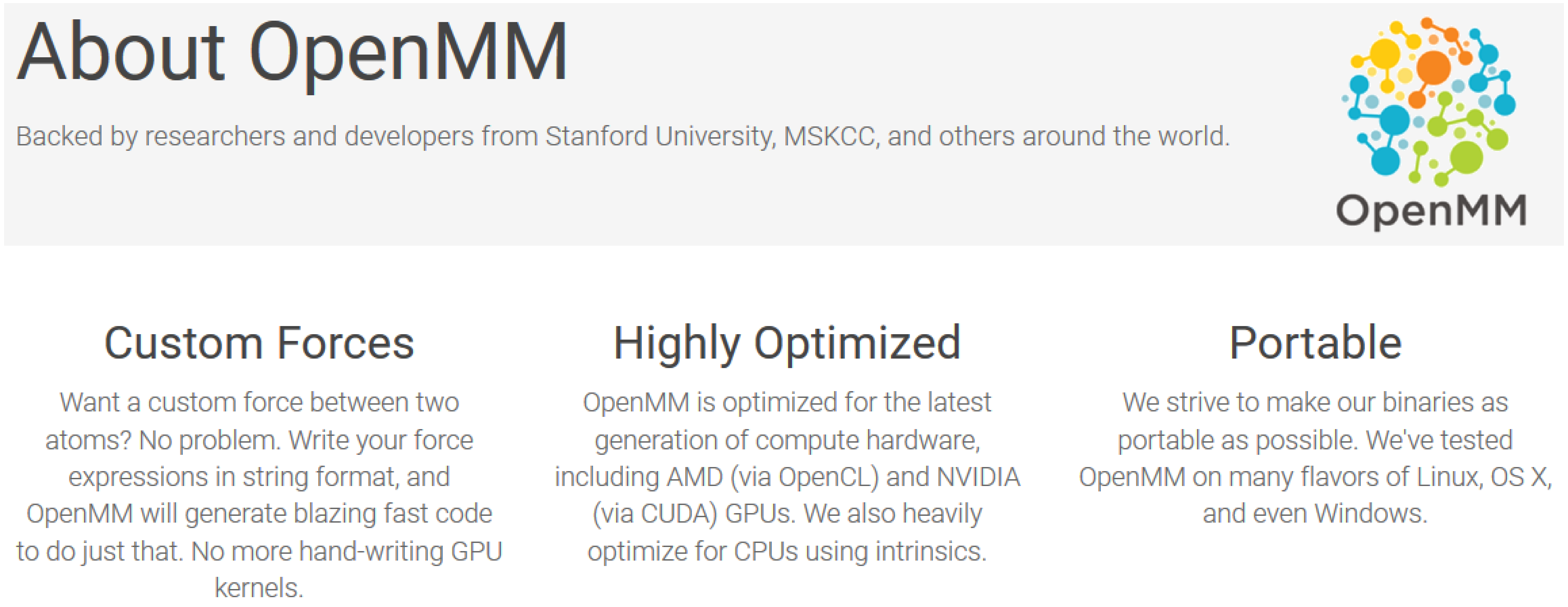
## Background

> OpenMM

> High performance toolkit for molecular simulation
> Python, C, C++, Fortran (!) bindings
> Open source
> Actively maintained on GitHub
> Licensed under MIT and LGPL

> http://openmm.org

# OpenMM



## About OpenMM

Backed by researchers and developers from Stanford University, MSKCC, and others around the world.

OpenMM

### Custom Forces

Want a custom force between two atoms? No problem. Write your force expressions in string format, and OpenMM will generate blazing fast code to do just that. No more hand-writing GPU kernels.

### Highly Optimized

OpenMM is optimized for the latest generation of compute hardware, including AMD (via OpenCL) and NVIDIA (via CUDA) GPUs. We also heavily optimize for CPUs using intrinsics.

### Portable

We strive to make our binaries as portable as possible. We've tested OpenMM on many flavors of Linux, OS X, and even Windows.

## What about…

> …spicing up the RDKit MM simulations with OpenMM?

- > Licenses are compatible
- > So are APIs
- > Initial proof-of-concept on MMFF94
- > If worthwhile, extend to UFF

OpenMM-accelerated
Open-Source Cheminformatics
and Machine Learning

## How much work is it?

OpenMM-accelerated
Open-Source Cheminformatics
and Machine Learning

- OpenMM does not implement MMFF94 natively
  - We need to implement it ourselves
  - Can we implement it using Custom Forces?
  - Can we borrow some Forces from already implemented force-fields?
  - Fortunately, atom types and force constants are already assigned by the RDKit

## Implementation

OpenMM side, C++

# OpenMM implementation of MMFF94 Forces

| | | | |
|---|---|---|---|
| $E_{MMFF} = \sum EB_{ij}$ | bond stretching | $EB_{ij} = 143.9325 \frac{kb_{ij}}{2} \Delta r_{ij}^2 \left(1 + cs\,\Delta r_{ij} + \frac{7}{12} cs^2 \Delta r_{ij}^2\right)$ | `OpenMM::CustomBondForce` |
| $+\sum EA_{ijk}$ | angle bending | $EA_{ijk} = 0.043844 \frac{ka_{IJK}}{2} \Delta \vartheta_{ijk}^2 \left(1 + cb\,\Delta \vartheta_{ijk}\right)$ | `OpenMM::CustomAngleForce` |
| $+\sum EBA_{ijk}$ | stretch-bend | $EBA_{ijk} = 2.51210 \left(kba_{IJK} \Delta r_{ij} + kba_{KJI} \Delta r_{kj}\right) \Delta \vartheta_{ijk}$ | `OpenMM::AmoebaStretchBendForce` |
| $+\sum EOOP_{ijk;l}$ | out-of-plane bending | $EOOP_{ijk;l} = 0.043844 \frac{koop_{IJK:L}}{2} \chi_{ijk;l}^2$ | `OpenMM::AmoebaOutOfPlaneBendForce` |
| $+\sum ET_{ijkl}$ | torsion | $ET_{ijkl} = 0.5\left[V_1 (1 + \cos\phi) + V_2 (1 - \cos 2\phi) + V_3 (1 + \cos 3\phi)\right]$ | `OpenMM::CustomTorsionForce` |
| $+\sum EvdW_{ij}$ | van der Waals | $EvdW_{ij} = \varepsilon_{IJ} \left(\frac{1.07 R_{IJ}^*}{R_{ij} + 0.07 R_{IJ}^*}\right)^7 \left(\frac{1.12 R_{IJ}^{*7}}{R_{ij}^7 + 0.12 R_{IJ}^{*7}} - 2\right)$ | `OpenMM::AmoebaVdwForce` |
| $+\sum EQ_{ij}$ | electrostatic | $EQ_{ij} = 332.0716 \frac{q_i q_j}{D\left(R_{ij} + \delta\right)^n}$ | `OpenMM::CustomNonbondedForce` |

## The devil is in the details

OpenMM side, C++/C

## The AMOEBA vdW term

`OpenMM::AmoebaVdwForce`

$$\mathrm{EvdW}_{ij} = \varepsilon_{IJ}\left(\frac{1.07R^{*}_{IJ}}{R_{ij}+0.07R^{*}_{IJ}}\right)^{7}\left(\frac{1.12R^{*7}_{IJ}}{R^{7}_{ij}+0.12R^{*7}_{IJ}}-2\right)$$

Unfortunately, none of the supported combination rules for $R^{*}_{IJ}$ and $\varepsilon_{IJ}$ applies to MMFF94

- $R_{IJ}$ is the distance between particles $I$ and $J$
- $R^{*}_{IJ}$ is a combination of particle-specific $R^{*}_{I}$, $R^{*}_{J}$ constants
- $\varepsilon_{IJ}$ is a combination of particle-specific $\varepsilon_{I}$, $\varepsilon_{J}$ constants
  - OpenMM supports three different combination rules for $R^{*}_{IJ}$ and four for $\varepsilon_{IJ}$

# The MMFF94 vdW term

## MMFF94 combination rules are complex

$$\mathrm{EvdW}_{ij} = \varepsilon_{IJ}\left(\frac{1.07R^{*}_{IJ}}{R_{ij}+0.07R^{*}_{IJ}}\right)^{7}\left(\frac{1.12R^{*7}_{IJ}}{R^{7}_{ij}+0.12R^{*7}_{IJ}}-2\right)$$

$$R^{*}_{IJ} = 0.5\left(R^{*}_{II}+R^{*}_{JJ}\right)\left(1+B\left(1-e^{-12\gamma^{2}_{IJ}}\right)\right) \qquad (1) \qquad \gamma_{IJ} = \frac{R^{*}_{II}-R^{*}_{JJ}}{R^{*}_{II}+R^{*}_{JJ}}$$

$$\varepsilon_{IJ} = \frac{181.16G_{I}G_{J}\alpha_{I}\alpha_{J}}{\sqrt{\frac{\alpha_{I}}{N_{I}}}+\sqrt{\frac{\alpha_{J}}{N_{J}}}}\cdot\frac{1}{R^{*6}_{IJ}} \qquad (2)$$

- > HBA/HBD particle pairs need their $R^{*}_{IJ}$ and $\varepsilon_{IJ}$ to be scaled down
- > In equation (1), B is 0 if one of the particles is a HBD, otherwise it is 0.2
- > The dependence of $\varepsilon_{IJ}$ from $R^{*}_{IJ}$, and the need for scaling would make `OpenMM::CustomNonbondedForces` computationally inefficient

## (AMOEBA + MMFF94)$^{\text{hurry}}$ = BIG HACK!

- I hacked the `OpenMM::AmoebaVdwForce` implementation adding a “MMFF” combination rule
- I abused the per-particle $\sigma_I$, $\varepsilon_I$ and reduction (not used by MMFF94) parameters passed to the `OpenMM::AmoebaVdwForce::addParticle()` method…
- …to pass $R_I$, $G_I a_I$ and $a_I / N_I$ instead, using sign combinations to encode HBA/HBD features
- I modified the platform-specific kernels accordingly

Yuk! But it works!

## The MMFF94 electrostatic term

$$EQ_{IJ} = s \cdot k \frac{q_I q_J}{D(R_{IJ} + \delta)^n}$$

- The problem here is that the scaling factor *s* has to be 0.75 for 1-4 interactions, and 1.0 for 1-*n* interactions (*n* > 4)

1-4 interactions

1-*n* interactions (*n* > 4)

- Unfortunately, only per-particle parameters are allowed, so the electrostatic term is split in two

- Per-particle and group exclusions do the rest

## Implementation

RDKit side, C++

## The OpenMMForceField class, or where the magic lies

The core of the new OpenMM-powered force field implementation is the ForceFields::OpenMMForceField class

```
class OpenMMForceField : public ForceField {
  public:
    [...]
    OpenMM::System *getSystem() const;
    OpenMM::Context *getContext() const;
    void setIntegrator(OpenMM::Integrator *integrator);
    OpenMM::Integrator *getIntegrator() const
    void initializeContext();
    void initializeContext(const std::string& platformName, const std::map<std::string, std::string> &prop);
    void initializeContext(OpenMM::Platform& platform, const std::map<std::string, std::string> &prop);
    double calcEnergy(std::vector<double> *contribs = NULL) const;
    double calcEnergy(double *pos);
    void calcGrad(double *forces) const;
    void calcGrad(double *pos, double *forces);
    int minimize(unsigned int maxIts = 200, double forceTol = 1e-4, double energyTol = 1e-6);
  protected:
    [...]
  private:
    [...]
}
```

Setters/getters for OpenMM::System, Platform and Integrator

OpenMM-enabled re-implementations of the base class methods

Before people get nervous: support for OpenMM can be *optionally* built into the RDKit ☺

cresset

## The MMFF::OpenMMForceField class, where more magic lies

The MMFF94-specific machinery is hosted by the MMFF::OpenMMForceField class

```
class OpenMMForceField : public ForceField {
  public:
    [...]
    void addBondStretchContrib(...);
    void addAngleBendContrib(...);
    void addStretchBendContrib(...);
    void addTorsionAngleContrib(...);
    void addOopBendContrib(...);
    void addVdWContrib(...);
    void addEleContrib(...);
    void addEleContrib1_4(...);
    const std::vector<std::string>& loadedPlugins();
    const std::vector<std::string>& failedPlugins();
  protected:
    [...]
  private:
    [...]
}
```

Before people get *even more* nervous: I haven't touched the base ForceField class, so ABI compatibility is guaranteed ☺

## Get me an OpenMM-enabled force field, now!

To construct an OpenMM-enabled force field, all you need is call the familiar

```
OpenMMForceField *constructOpenMMForceField(ROMol &mol,
  MMFFMolProperties *mmffMolProperties, double nonBondedThresh = 100.0,
  int confId = -1, bool ignoreInterfragInteractions = true);
```

The only difference from the well-known call are those two tiny `OpenMM` prefixes

> Once you have created it, you may do the usual things:
  > Calculate the potential energy: `ff->calcEnergy()`
  > Run a minimization: `ff->minimize()`
> Or more exotic ones:
  > Run *n* steps of MD: `ff->dynamics(n)`

## Implementation

RDKit side, Python

## I already know how to do this, don’t I?

Also in Python, you shouldn’t be too surprised by the new API:

```
MMFFGetMoleculeOpenMMForceField( (Mol)mol, (object)pyMMFFMolProperties [,
  (float)nonBondedThresh = 100.0 [, (int)confid = -1 [,
  (bool)ignoreInterfragInteractions = True]]])
```

Again, you’ll only need to add a tiny `OpenMM` to your existing scripts

> Once you have created it, you may do the usual things:
>
> > Calculate the potential energy: `ff.CalcEnergy()`
>
> > Run a minimization: `ff.Minimize()`

> Or more exotic ones:
>
> > Run *n* steps of MD: `ff.Dynamics(n)`

## Results

On to the Jupyter notebook

## Conclusions and outlook

A.k.a., the to-do list

## Conclusions

- The OpenMM implementation of MMFF94 within the RDKit
  - delivers impressive performance even on consumer GPU hardware
  - enables fast molecular mechanics simulations on both CPUs and GPUs
  - Can be accessed with minimal modifications to old scripts

## Outlook

> Before I can get all this to you I need:
  > To properly implement MMFF94 in OpenMM (no hacks!)
  > To extend the implementation to UFF
  > To add more APIs specific to molecular dynamics
  > To put some thought in the API design to give full access to present and future OpenMM functionality

cresset

smarter chemistry | smarter decisions

## Thank you for your attention

paolo@cresset-group.com

Acknwoledgments

> The OpenMM team for such a great open-source simulation toolkit

> The RDKit team for… well, you know

> Cresset for supporting this project

cressetgroup