ParticleGrid: Enabling Deep Learning using 3D Representation of Materials

Ethan Ferguson*, Shehtab Zaman*, Kenneth Chiu*, Denis Akhiyarov↑, Cécile Pereira↑, Mauricio Araya↑

* Binghamton University

↑Total Energies SE

ParticleGrid

- Is a library for generating 3D grids to represent molecules
- Is designed for deep learning applications and to seamlessly integrate with deep learning frameworks
  - Generative and predictive models are the goal
  - Low overhead and simple to install
- Is highly optimized
  - Orders of magnitude faster than NumPy and Numba
  - Up to 9000x, 79.5x, and 14x over NumPy, Numba, and baseline C++ implementations

Representations of Molecules

$C_9H_8O_4$ or $CH_3COOC_6H_4COOH$ or Aspirin

(a) SMILES:

CC(=O)OC1=CC=CC=C1C(=O)O

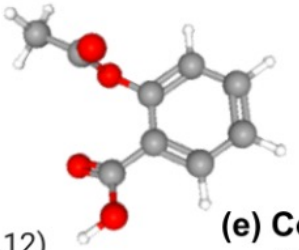
(b) InChI:

1S/C9H8O4/c1-6(10)13-8-5-3-2-4-7(8)9(11)12/h2-5H,1H3,(H,11,12)

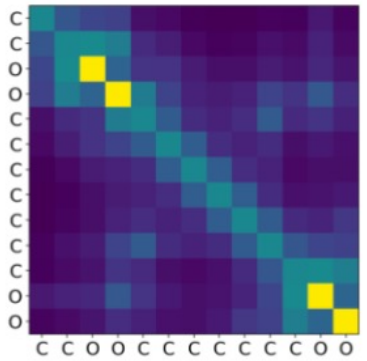
(c) SELFIES:

[C][C][=Branch1][C][=O][O][C][=C][C][=C][C][=C][Ring1][=Branch1][C][=Branch1][C][=O][O]

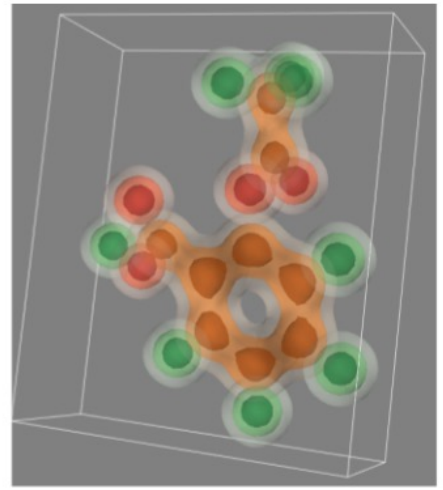
(d) Graph:



(e) Coulomb Matrix:



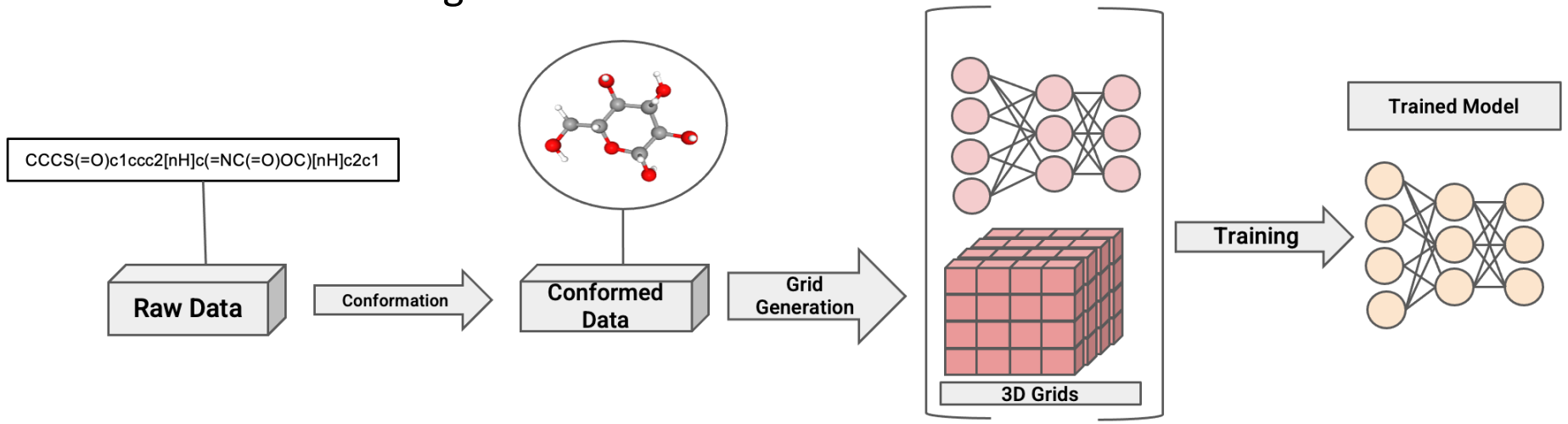
(f) 3D Grid:



- String based representations are compact and expressive
  - But they don’t generalize for crystals and other materials
  - Symmetries are hidden
- Graph based representations are a natural choice for molecules
  - Graph generation is difficult

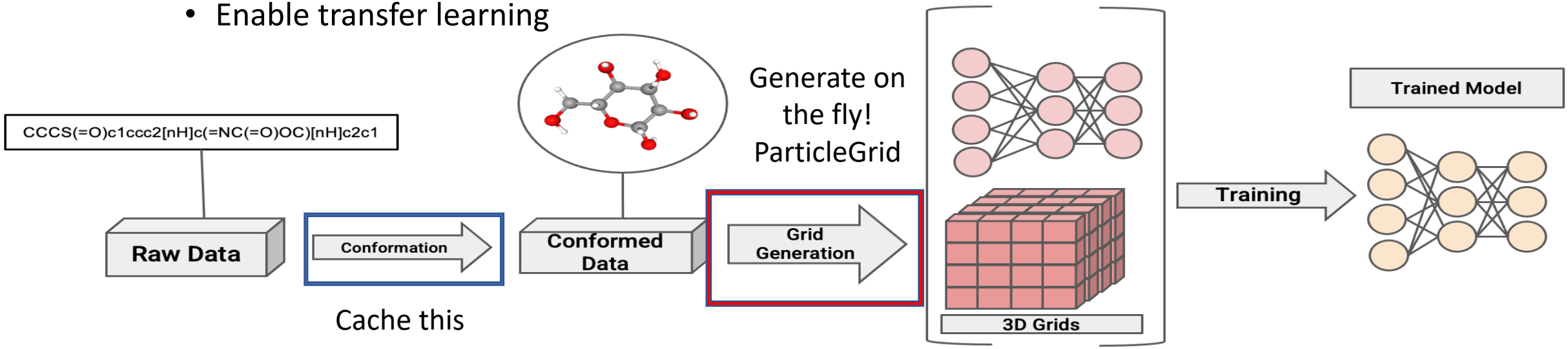
Design Goals

- Why use 3D grids?
  - Learning on tensors is convenient and well-studied
  - Deep learning generative models on grids
  - Convenient, reusable representation
    - Enable transfer learning
- Design Goals:
  - Application in deep learning
  - Ease of use
  - Flexible
  - Fast
  - Invertible



Design Goals

- Why use 3D grids?
  - Learning on tensors is convenient and well-studied
  - Deep learning generative models on grids
  - Convenient, reusable representation
    - Enable transfer learning
- Design Goals:
  - Application in deep learning
  - Ease of use
  - Flexible
  - Fast
  - Invertible



3D Guassian Grids

- 3D grids based on Gaussian spreads of atoms:
- Each grid point is the sum over the integral of the 3D Guassian function
  - Calculate 6 erf per grid point per molecule

$$f_{\vec{\mu}}(\vec{p}) = \frac{1}{\sigma^3 (2\pi)^{\frac{3}{2}}} e^{-\frac{d(\vec{(}\mu),\vec{p})}{2\sigma^2}}$$

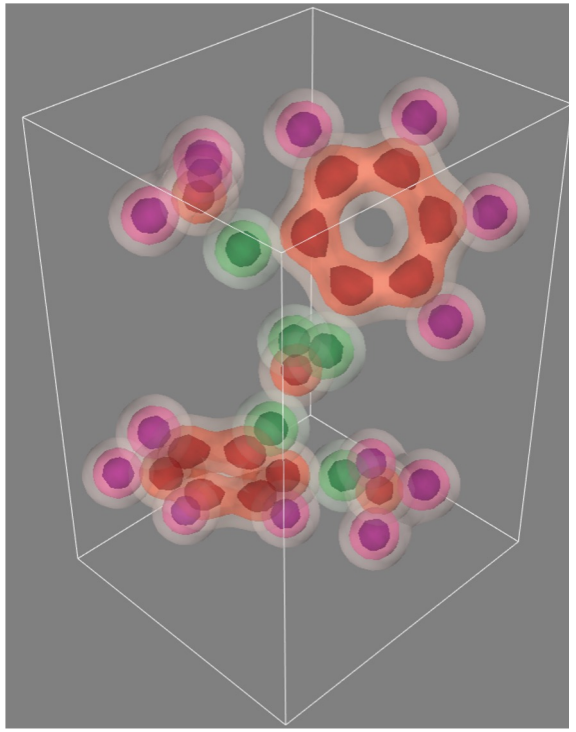
$$F_{\vec{\mu}}(\vec{p}) = \left(\frac{1}{2}\right)^3 G_{\mu_x}(x_a) H_{\mu_y}(y_b) L_{\mu_z}(z_c)$$

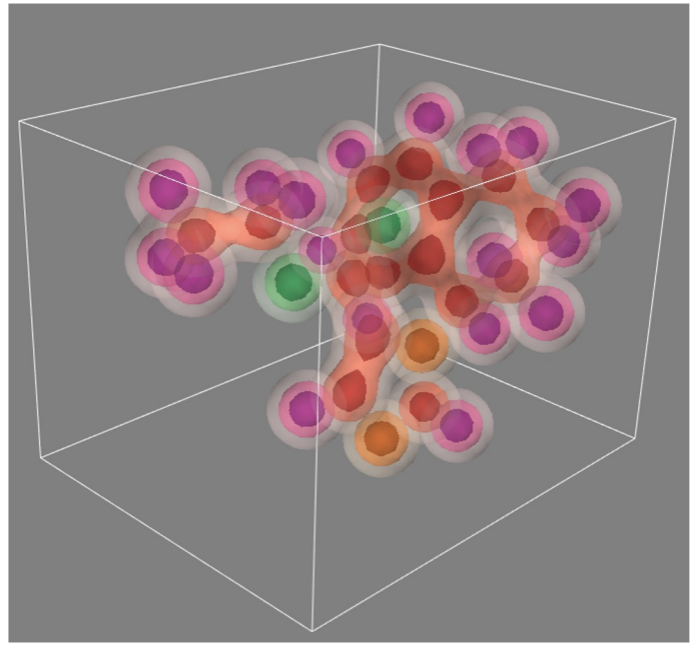
$$G_{\mu_x}(x) = \operatorname{erf}\left(\frac{\sqrt{2}}{2\sigma}(\mu_i - x)\right)\Bigg|_{x_i}^{x_i+\delta_x}$$

$$H_{\mu_y}(y) = \operatorname{erf}\left(\frac{\sqrt{2}}{2\sigma}(\mu_j - y)\right)\Bigg|_{y_j}^{y_j+\delta_y}$$

$$L_{\mu_z}(z) = \operatorname{erf}\left(\frac{\sqrt{2}}{2\sigma}(\mu_k - z)\right)\Bigg|_{z_k}^{z_k+\delta_z}$$

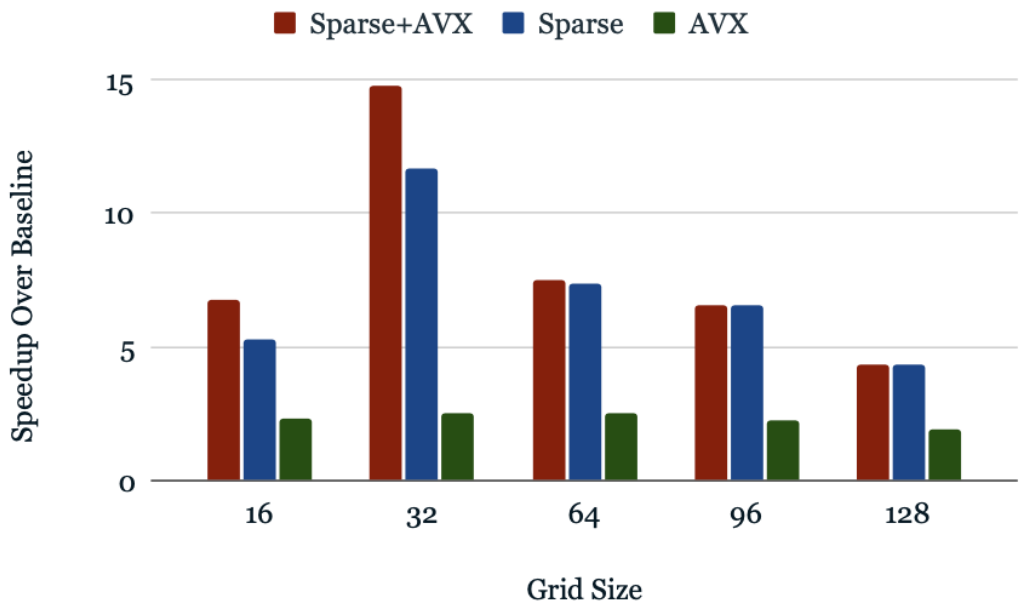
- Challenges:
  - Reduce the number of calculations
  - Exploit data parallelism





Performance Optimizations

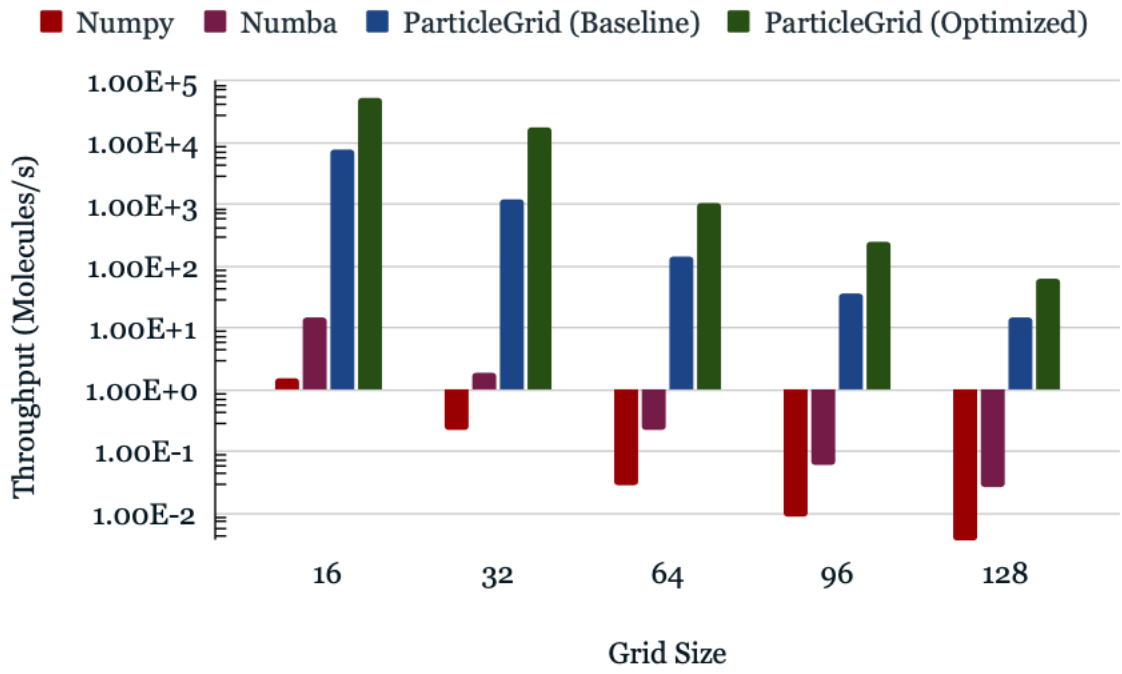
- Exploiting Truncation
  - Probe Gaussian values over grid in linear time over a single direction
  - Skip regions without signal in one dimension
    - No op for cells with little to no signal
  - Results in 7x speed up
- SIMD parallelization of main functions
  - Custom SIMD **erf** implementation using Bürman series approximation



The evaluations are performed on an Intel Xeon Silver 4114 CPU @ 2.20 Ghz, with 32k L1i and L1d cache, 1024K L2 cache, and 14080k L3 cache, 128 GB of system RAM and GCC 7.5.0

Performance Optimizations

- We achieve up to 9000x, 79.5x, and 14x over Numpy, Numba, and baseline C++ implementations
- This enables us to generate grids on the fly for deep learning training



The evaluations are performed on an Intel Xeon Silver 4114 CPU @ 2.20 Ghz, with 32k L1i and L1d cache, 1024K L2 cache, and 14080k L3 cache, 128 GB of system RAM and GCC 7.5.0

Using ParticleGrid

- ParticleGrid can be installed via pip
- Low dependency overhead:
  - C++17 enabled compiler
  - NumPy
- Smooth integration with deep learning frameworks
  - Zero-copy transfers to data containers
  - Integrates into existing data pipelines

```python
import numpy as np
import torch
import tensorflow as tf
from ParticleGrid import coord_to_grid

# Points are in the format (channel, x, y, z)
test_points = np.array([0, 0.5, 0.5, 0.5],
                       [1, 0.0, 0.1, 0.2])
# Generates a (2,32,32,32) grid
grid = coord_to_grid(test_points,
                     width=1,
                     height=1,
                     depth=1,
                     num_channels=2,
                     grid_size=32,
                     variance=0.05)

# Convert to PyTorch tensor
grid_torch_tensor = torch.from_numpy(grid)
# Convert to TensorFlow tensor
grid_tf_tensor = tf.convert_to_tensor(grid)
```

Using ParticleGrid

- The grid generation function ***coord_to_grid*** takes as input:
  - A set of of coordinates to transform
  - The dimensions of the extent (bounding box):
    - Width
    - Height
    - Depth
  - The size of the grid
  - The variance (amount of spread)

```
import numpy as np
import torch
import tensorflow as tf
from ParticleGrid import coord_to_grid

# Points are in the format (channel, x, y, z)
test_points = np.array([0, 0.5, 0.5, 0.5],
                       [1, 0.0, 0.1, 0.2])
# Generates a (2,32,32,32) grid
grid = coord_to_grid(test_points,
                     width=1,
                     height=1,
                     depth=1,
                     num_channels=2,
                     grid_size=32,
                     variance=0.05)

# Convert to PyTorch tensor
grid_torch_tensor = torch.from_numpy(grid)
# Convert to TensorFlow tensor
grid_tf_tensor = tf.convert_to_tensor(grid)
```

Using ParticleGrid

- Returns a NumPy array
  - By reference. Handles ownership of object over to Python
  - Most prominent DL libraries can ingest NumPy without copying the data

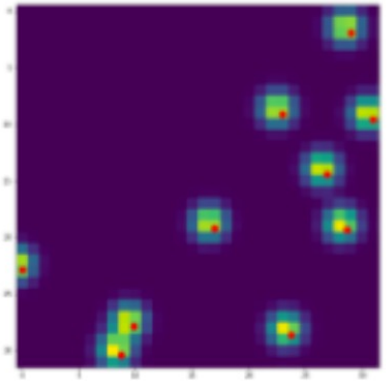
```python
import numpy as np
import torch
import tensorflow as tf
from ParticleGrid import coord_to_grid

# Points are in the format (channel, x, y, z)
test_points = np.array([0, 0.5, 0.5, 0.5],
                       [1, 0.0, 0.1, 0.2])
# Generates a (2,32,32,32) grid
grid = coord_to_grid(test_points,
                     width=1,
                     height=1,
                     depth=1,
                     num_channels=2,
                     grid_size=32,
                     variance=0.05)

# Convert to PyTorch tensor
grid_torch_tensor = torch.from_numpy(grid)
# Convert to TensorFlow tensor
grid_tf_tensor = tf.convert_to_tensor(grid)
```

Retrieving Coordinates from Grids

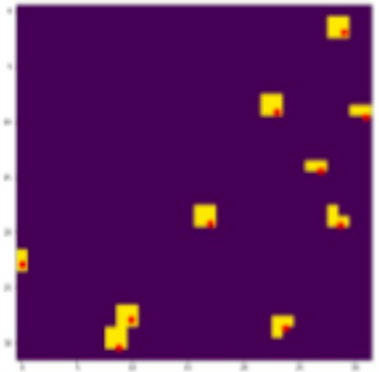
Input Grid Representation: The input to retrieve the generating coordinates of.

3D Peak Finding: Provides most likely location of atoms in the grid

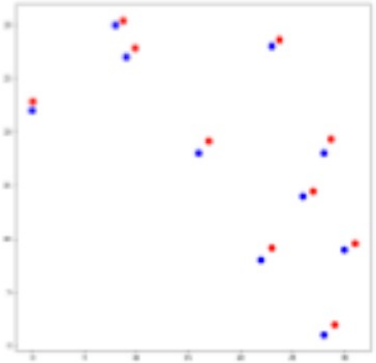
The true coordinates, generated from conformations of a molecules are never available to the discretizer. The approximated grid is optimized to match the input grid.



Cubic PH

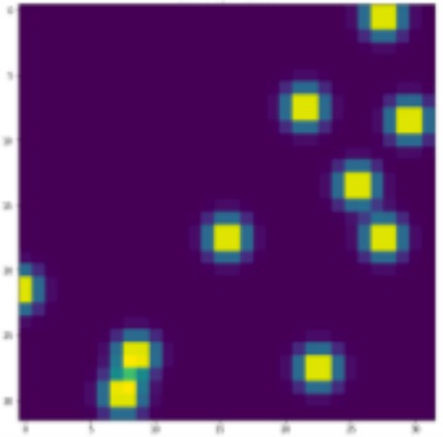


Estimate Coordinates

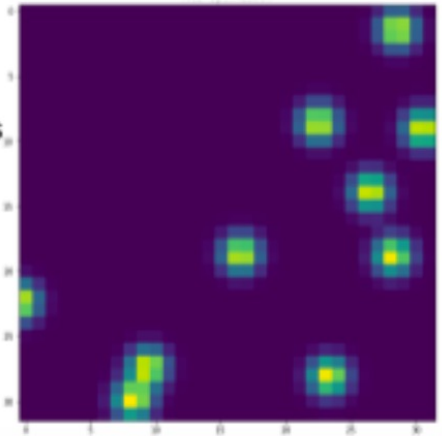


Atom Centers: Using likely locations infer the possible coordinates of the atoms. Here the blue is the first approximations and the red dots are the ground truth, The red dots are never available for comparison by the discretizer

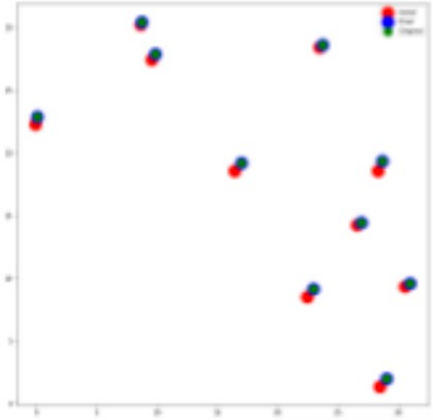
Grid



Optimize Grid



Optimized Coordinates

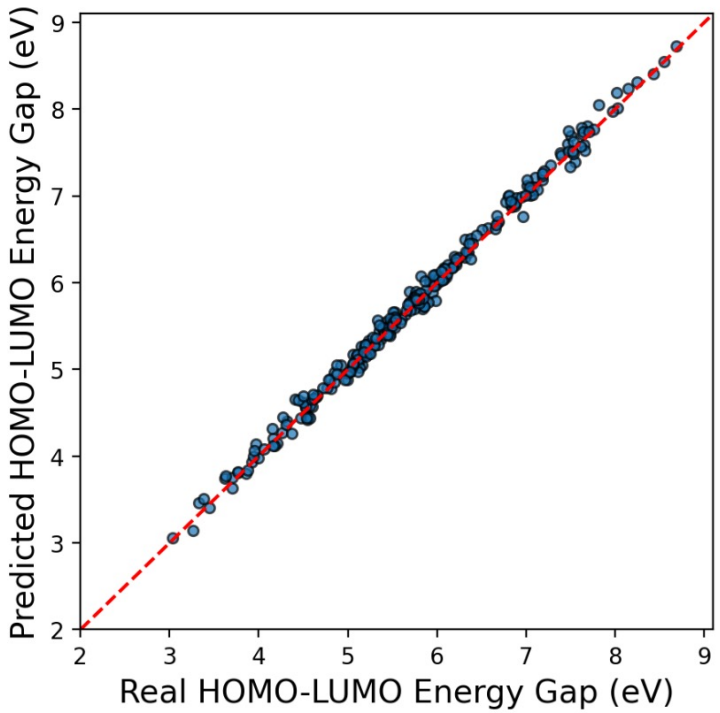


After optimization, the atomic coordinates will have the best possible coordinates. Red dots are updated via GD.

Approximated Grid: Using the inferred locations, generated grids again. Compare with the original grid. The better the atomic coordinates are approximated, the more similar the two grids will be.

Practical Application: Property Prediction

- Natural to consider the efficacy of this representation
- We use 3D structure for learning on molecular properties
- We train on the Open Graph Benchmarks Large-Scale Challenge PubChem Quantum Mechanics for Molecules (OGB-LSC PCQM4M)
  - Predict the highest occupied molecular orbital (HOMO) lowest unoccupied molecular orbital (LUMO) energy gap using 3D coordinates of molecules
- With a 10 layer 2D-Residual Network
  - We achieve a 0.006 MSE on the test set
  - A throughput of about 50,000 molecules a second!



Future Work

- Enable wider range of materials such as periodic crystals
- Support SIMD on ARM and Power architectures
- Integrate noise generation to enable on the fly diffusion data generation
- Support new use cases for practitioners
  - We are looking for users!
  - If you have a problem that could use molecular 3D information for deep learning, we're interested!
- The public repo is located at:

  https://github.com/ParticleGrid/ParticleGrid

Acknowledgements

We would like to thank TotalEnergies EP Research & Technology for supporting and allowing this material to be shared.