RAPIDS & CUDA PYTHON FOR BLAZING FAST SCIENTIFIC COMPUTING

Abraham Stern, Ph.D., January 21, 2020
THE NEW HPC
Accelerating ML, DL, HPC and Visualization Workflows

MACHINE LEARNING
RAPIDS | H2O | more

DEEP LEARNING
TensorFlow | PyTorch | more

HPC
NAMD | GROMACS | +600 More

VISUALIZATION
ParaView | IndeX | more
NVIDIA ACCELERATED COMPUTING IS ACCELERATING

NVIDIA Share of New Top 500 Systems

- **ORNL Summit**
  - World’s Fastest
  - 27,648 GPUs | 149 PF

- **LLNL Sierra**
  - World’s 2nd Fastest
  - 17,280 GPUs | 95 PF

- **Piz Daint**
  - Europe’s Fastest
  - 5,704 GPUs | 21 PF

- **Total Pangea 3**
  - Fastest Industrial
  - 3,348 GPUs | 18 PF

- **ABCI**
  - Japan’s Fastest
  - 4,352 GPUs | 20 PF
INTERSECTION OF HPC & AI
TRANSFORMING SCIENCE

HPC
> Algorithms based on first principles theory
> Proven models for accurate results

AI
> Neural networks that learn patterns from large data sets
> Improve predictive accuracy and faster response time

IDENTIFYING CHEMICAL COMPOUNDS
EXASCALE WEATHER MODELING
SPEEDING PATH TO FUSION ENERGY
O&G FAULT INTERPRETATION

90% Prediction Accuracy
Published in Nature April 2019
Tensor Cores Achieved 1.13 EF
2018 Gordon Bell Winner
Orders Of Magnitude Speedup
3M New Compounds In 1 Day
Time-to-solution Reduced From Weeks To 2 Hours
AI FOR SCIENCE
BRINGING HPC AND AI TOGETHER

Combining simulation with the power of AI to
Dramatically accelerate time to insight

Design Space Exploration
ICF + MERLIN — Fusion

Inverse Problems
LIGO — Gravitational Waves

Faster Prediction
ANI + MD - Chemistry

Real-time Steering
ITER - Fusion Energy
BEYOND MOORE’S LAW
Progress Of Stack In 6 Years

- cuBLAS: 5.0
- cuFFT: 5.0
- cuRAND: 5.0
- cuSPARSE: 5.0
- NPP: 5.0
- Thrust: 1.5.3
- CUDA: 5.0
- Resource Mgr: r304
- Base OS: CentOS 6.2

- cuBLAS: 10.0
- cuFFT: 10.0
- cuRAND: 10.0
- cuSOLVER: 10.0
- cuSPARSE: 10.0
- NPP: 10.0
- Thrust: 1.9.0
- CUDA: 10.0
- Resource Mgr: r384
- Base OS: Ubuntu 16.04

Accelerated Server with Volta

Measured performance of Amber, CHROMA, GTC, LAMMPS, MILC, NAMD, Quantum Espresso, SPECFEM3D
# NVIDIA DATA CENTER PLATFORM

Single Platform Drives Utilization and Productivity

## CUSTOMER USE CASES

<table>
<thead>
<tr>
<th>Speech</th>
<th>Translate</th>
<th>Recommender</th>
<th>Healthcare</th>
<th>Manufacturing</th>
<th>Finance</th>
<th>Molecular Simulations</th>
<th>Weather Forecasting</th>
<th>Seismic Mapping</th>
<th>Creative &amp; Technical</th>
<th>Knowledge Workers</th>
</tr>
</thead>
</table>

## CONSUMER INTERNET & INDUSTRY APPLICATIONS

## SCIENTIFIC APPLICATIONS

## VIRTUAL GRAPHICS

## APPS & FRAMEWORKS

- **Python**
- **RAPIDS**
- **TensorFlow**
- **PyTorch**
- **MXNet**
- **Chainer**
- **ONNX**
- **Amber**
- **NAMD**
- Additional **+600 Applications**

## CUDA-X & NVIDIA SDKs

### MACHINE LEARNING

- **cuDF**
- **cuML**
- **cuGRAPH**

### DEEP LEARNING

- **cuDNN**
- **CUTLASS**
- **TensorRT**

### HPC

- **OpenACC**
- **cuFFT**

### VIRTUAL GPU

- **vDWS**
- **vPC**
- **vAPPS**

## CUDA & CORE LIBRARIES - cuBLAS | NCCL

## TESLA GPUs & SYSTEMS

- **TESLA GPU**
- **NVIDIA DGX FAMILY**
- **NVIDIA HGX**
- **EVERY OEM**
- **EVERY MAJOR CLOUD**
GPU ACCELERATED LIBRARIES

“Drop-in” Acceleration for Your Applications

DEEP LEARNING
- cuDNN
- TensorFlow
- DeepStream SDK

SIGNAL, IMAGE & VIDEO
- cuFFT
- NVIDIA NPP
- CODEC SDK

LINEAR ALGEBRA
- cuBLAS
- CUDA Math library
- cuSOLVER
- cuSPARSE
- cuRAND

PARALLEL ALGORITHMS
- nvGRAPH
- NCCL
- Thrust
NVIDIA CUDA-X UPDATES
Software To Deliver Acceleration For HPC & AI Apps; 500+ New Updates

CUDA-X HPC & AI
40+ GPU Acceleration Libraries

CUDA
Desktop Development
Data Center
Supercomputers
GPU-Accelerated Cloud
WHAT IS OPENACC
Open Specification Developed by OpenACC.org Consortium

Directives-based programming model for parallel computing

Add Simple Compiler Directive

```c
main()
{
  <serial code>
  #pragma acc kernels
  {
    <parallel code>
  }
}
```

Designed for performance and portability on CPUs and GPUs

Read more at www.openacc.org/about
Using OpenACC allowed us to continue development of our fundamental algorithms and software capabilities simultaneously with the GPU-related work. In the end, we could use the same code base for SMP, cluster/network and GPU parallelism. PGI's compilers were essential to the success of our efforts.
For VASP, OpenACC is the way forward for GPU acceleration. Performance is similar and in some cases better than CUDA C, and OpenACC dramatically decreases GPU development and maintenance efforts. We’re excited to collaborate with NVIDIA and PGI as an early adopter of CUDA Unified Memory.
SCIENTIFIC COMPUTING WORKFLOW
Running on the big machine is only one part of the workflow

- Data preprocessing
- Algorithm development or prototyping
- Test runs or small trial runs
- Post-run analysis
REPERCUSSIONS OF WAITING

Running on the big machine is only one part of the workflow

- Disrupts train of thought
- Slows the innovation/discovery cycle
- Kills momentum
- Hurts creativity
- Ultimately, hurts the science
GPU ACCELERATED COMPUTING IN PYTHON

- Write GPU-accelerated code in Python
- High-level API with Numpy syntax
- Low-level API with CUDA syntax
- Automatic data movement
- Or manual data movement
GPU ACCELERATED COMPUTING IN PYTHON WITH NUMBA

```python
@cuda.jit
def distance_mat(crds, natoms, edge_feats):
    tx = cuda threadIdx.x
    ty = cuda threadIdx.y
    bx = cuda blockIdx.x

    edge_feats[bx, tx, ty] = 0.0

    # bounds check
    N = natoms[bx]
    if tx >= N: return
    if ty >= N: return

    for i in range(3):
        edge_feats[bx, tx, ty] += ( crds[bx, tx, i] - crds[bx, ty, i] )**2

    edge_feats[bx, tx, ty] = math.sqrt(edge_feats[bx, tx, ty])
```

- **Decorator to just-in-time compile CUDA**
- **CUDA-provided variables for thread indices**
- **Map our thread indices to some work**
INTRODUCTION TO CUDA

GPUs do work in parallel

performWork<<<2, 4>>>()
INTRODUCTION TO CUDA

GPU work is done in a thread

performWork<<<2, 4>>>()

GPU
INTRODUCTION TO CUDA

Many threads run in parallel

performWork<<<2, 4>>>()
INTRODUCTION TO CUDA

A collection of threads is a block

performWork<<<2, 4>>>(())
INTRODUCTION TO CUDA

There are many blocks

performWork<<<2, 4>>>()
INTRODUCTION TO CUDA

A collection of blocks associated with a given kernel launch is a grid

```
performWork<<<2, 4>>>()
```
INTRODUCTION TO CUDA

GPU functions are called kernels

```
performWork<<<2, 4>>>()
```
Kernels are launched with an execution configuration

```
performWork<<<2, 4>>>()
```
INTRODUCTION TO CUDA

The execution configuration defines the number of blocks in the grid

```c
performWork<<<2, 4>>>();
```
INTRODUCTION TO CUDA

... as well as the number of threads in each block

performWork<<<2, 4>>>()
INTRODUCTION TO CUDA

Every block in the grid contains the same number of threads

```
performWork<<<2, 4>>>()
```
Inside kernels definitions, CUDA-provided variables describe its executing thread, block, and grid.

```
performWork<<<2, 4>>>()
```
INTRODUCTION TO CUDA

gridDim.x is the number of blocks in the grid, in this case 2

performWork<<<2, 4>>>()
blockIdx.x is the index of the current block within the grid, in this case 0

performWork<<<2, 4>>>()
blockIdx.x is the index of the current block within the grid, in this case 1.
Inside a kernel `blockDim.x` describes the number of threads in a block. In this case 4 threads are executed in parallel.
INTRODUCTION TO CUDA

All blocks in a grid contain the same number of threads

performWork<<<2, 4>>>()
Inside a kernel `threadIdx.x` describes the index of the thread within a block. In this case 0

```
performWork<<<2, 4>>>()
```
Inside a kernel `threadIdx.x` describes the index of the thread within a block. In this case 1
Inside a kernel `threadIdx.x` describes the index of the thread within a block. In this case 2
Inside a kernel `threadIdx.x` describes the index of the thread within a block. In this case 3

```
performWork<<<2, 4>>>()
```
INTRODUCTION TO CUDA

Inside a kernel `threadIdx.x` describes the index of the thread within a block. In this case 0

```
performWork<<<2, 4>>>()

GPU
```
Inside a kernel `threadIdx.x` describes the index of the thread within a block. In this case 1
INTRODUCTION TO CUDA

Inside a kernel `threadIdx.x` describes the index of the thread within a block. In this case 2
Inside a kernel `threadIdx.x` describes the index of the thread within a block. In this case 3...
performWork<<<2, 4>>>(

<table>
<thead>
<tr>
<th>threadIdx.x</th>
<th>blockIdx.x * blockDim.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

dataIndex

0
GPU DATA

performWork<<<2, 4>>>()

<table>
<thead>
<tr>
<th>threadIdx.x</th>
<th>blockIdx.x</th>
<th>blockDim.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

dataIndex

1
CUDA kernel function `performWork<<<2, 4>>>()`

<table>
<thead>
<tr>
<th>threadIdx.x + blockIdx.x * blockDim.x</th>
<th>dataIndex</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

Data index: 2

CUDA data layout: 0, 1, 2, 3
performWork<<<2, 4>>>()

threadIdx.x + blockIdx.x * blockDim.x

\[
\begin{array}{ccc}
0 & 1 & 4 \\
1 & 2 & 5 \\
2 & 3 & 6 \\
3 & 4 & 7 \\
\end{array}
\]

dataIndex

\[
\begin{array}{c}
4 \\
\end{array}
\]
performWork<<<2, 4>>>(

<table>
<thead>
<tr>
<th>threadIdx.x</th>
<th>blockIdx.x</th>
<th>blockDim.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

dataIndex = 5

GPU DATA

GPU
performWork<<<2, 4>>>(

dataIndex

threadIdx.x  +  blockIdx.x  *  blockDim.x

2  1  4

0  1  2  3

0  1  2  3
CUDA kernel call:

```c
performWork<<<2, 4>>>()
```

**Kernel Execution and Data Mapping**

- **Thread Index Calculation**:
  - `threadIdx.x`: 0, 4
  - `blockIdx.x`: 1
  - `blockDim.x`: 4
  - Calculation: `threadIdx.x + blockIdx.x * blockDim.x = 3 + 1 * 4 = 7`

- **Data Index**: 7

**Data Layout**

- **Left Block**: `0 1 2 3`
- **Right Block**: `0 1 2 3`

**GPU Data Mapping**

- **Left Block**: Index 0 to 3
- **Right Block**: Index 4 to 7
We can also have a two- or three-dimensional thread block or grid:

```
performWork<<<2, (4, 4)>>>()
```

```
0 1 2 3
0 1 2 3
```

```
threadIdx.y
```

```
threadIdx.x
```

```
threadIdx.x
```
GPU ACCELERATED COMPUTING IN PYTHON WITH NUMBA

```python
c@cuda.jit
def distance_mat(crdx, natoms, edge_feats):
    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bx = cuda.blockIdx.x

    edge_feats[bx, tx, ty] = 0.0

    # bounds check
    N = natoms[bx]
    if tx >= N: return
    if ty >= N: return

    for i in range(3):
        edge_feats[bx, tx, ty] += (crdx[bx, tx, i] - crdx[bx, ty, i])**2

    edge_feats[bx, tx, ty] = math.sqrt(edge_feats[bx, tx, ty])
```

- **Decorator to just-in-time compile CUDA**
- **CUDA-provided variables for thread indices**
- **Map our thread indices to some work**
# create result array on the GPU
define edge_feats_gpu = cuda.device_array(((LAST_XYZ+1), ATOM_MAX, ATOM_MAX), dtype=np.float32)

# transfer structures data (the coordinates) to GPU
define structures_gpu = cuda.to_device(structures_crd_mat)
define natoms_gpu = cuda.to_device(structures_natoms_mat)

# launch kernel
define distance_mat[ LAST_XYZ+1 , (29, 29)](structures_gpu, natoms_gpu, edge_feats_gpu)

# copy the data back
define edge_feats_cpu = edge_feats_gpu.copy_to_host()

# make sure we're done
define cuda.synchronize()
@vectorize(['float32(float32, float32, float32)',
             'float64(float64, float64, float64)',
             target='cuda'])

def cu_discriminant(a, b, c):
    return math.sqrt(b ** 2 - 4 * a * c)

N = 10000
dtype = np.float32

# prepare the input
A = np.array(np.random.sample(N), dtype=dtype)
B = np.array(np.random.sample(N) + 10, dtype=dtype)
C = np.array(np.random.sample(N), dtype=dtype)

D = cu_discriminant(A, B, C)

print(D)  # print result
RAPIDS — OPEN GPU DATA SCIENCE
Software Stack

Data Preparation → Model Training → Visualization

Python

ARROW

CUDA

DASK

Deep Learning Frameworks

CUDNN

RAPIDS

CUML

CUDF

CUGRAPH

CUDA

Apache Arrow
cuDF for GPU-accelerated DataFrames

<table>
<thead>
<tr>
<th>molecule_name</th>
<th>atom_index</th>
<th>atom</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>atom_codes</th>
<th>oh_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>dsgdb9nsd_000001</td>
<td>0</td>
<td>C</td>
<td>-0.012698</td>
<td>1.085804</td>
<td>0.008001</td>
<td>0</td>
<td>1.0</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>1</td>
<td>H</td>
<td>0.002150</td>
<td>-0.006031</td>
<td>0.001976</td>
<td>2</td>
<td>0.0</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>2</td>
<td>H</td>
<td>1.011731</td>
<td>1.463751</td>
<td>0.000277</td>
<td>2</td>
<td>0.0</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>3</td>
<td>H</td>
<td>-0.540815</td>
<td>1.447527</td>
<td>-0.876644</td>
<td>2</td>
<td>0.0</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>4</td>
<td>H</td>
<td>-0.523814</td>
<td>1.437933</td>
<td>0.906397</td>
<td>2</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Reading in Data

```python
structs_gdf = gd.read_csv('/data/structures.csv')
```
DataFrame from Pandas

```python
structs_df = pd.read_csv('/data/structures.csv')
structs_gdf = gd.from_pandas(structs_df)
```

<table>
<thead>
<tr>
<th>molecule_name</th>
<th>atom_index</th>
<th>atom</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>atom_codes</th>
<th>oh_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>dsgdb9nsd_000001</td>
<td>0</td>
<td>C</td>
<td>-0.012698</td>
<td>1.085804</td>
<td>0.008001</td>
<td>0</td>
<td>1.0</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>1</td>
<td>H</td>
<td>0.002150</td>
<td>-0.006031</td>
<td>0.001976</td>
<td>2</td>
<td>0.0</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>2</td>
<td>H</td>
<td>1.011731</td>
<td>1.463751</td>
<td>0.000277</td>
<td>2</td>
<td>0.0</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>3</td>
<td>H</td>
<td>-0.540815</td>
<td>1.447527</td>
<td>-0.876644</td>
<td>2</td>
<td>0.0</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>4</td>
<td>H</td>
<td>-0.523814</td>
<td>1.437933</td>
<td>0.906397</td>
<td>2</td>
<td>0.0</td>
</tr>
</tbody>
</table>
DataFrames are PrettyPrinted

```python
structs_df.head()
```

<table>
<thead>
<tr>
<th>molecule_name</th>
<th>atom_index</th>
<th>atom</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>atom_codes</th>
<th>oh_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>dsgdb9nsd_000001</td>
<td>0</td>
<td>C</td>
<td>-0.012698</td>
<td>1.085804</td>
<td>0.008001</td>
<td>0</td>
<td>1.0</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>1</td>
<td>H</td>
<td>0.002150</td>
<td>-0.006031</td>
<td>0.001976</td>
<td>2</td>
<td>0.0</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>2</td>
<td>H</td>
<td>1.011731</td>
<td>1.463751</td>
<td>0.000277</td>
<td>2</td>
<td>0.0</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>3</td>
<td>H</td>
<td>-0.540815</td>
<td>1.447527</td>
<td>-0.876644</td>
<td>2</td>
<td>0.0</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>4</td>
<td>H</td>
<td>-0.523814</td>
<td>1.437933</td>
<td>0.906397</td>
<td>2</td>
<td>0.0</td>
</tr>
</tbody>
</table>
# Integer Slicing for Row Selection

<table>
<thead>
<tr>
<th>molecule_name</th>
<th>atom_index</th>
<th>atom</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>atom_codes</th>
<th>oh_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>dsgdb9nsd_000004</td>
<td>2</td>
<td>H</td>
<td>-1.661639</td>
<td>0.000000</td>
<td>1.000000</td>
<td>2</td>
<td>0.0</td>
</tr>
<tr>
<td>dsgdb9nsd_000004</td>
<td>3</td>
<td>H</td>
<td>1.661639</td>
<td>0.000000</td>
<td>1.000000</td>
<td>2</td>
<td>0.0</td>
</tr>
<tr>
<td>dsgdb9nsd_000005</td>
<td>0</td>
<td>C</td>
<td>-0.013324</td>
<td>1.132466</td>
<td>0.008276</td>
<td>0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
### Column Selection by Column Name

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>H</td>
</tr>
<tr>
<td>2</td>
<td>H</td>
</tr>
<tr>
<td>3</td>
<td>H</td>
</tr>
<tr>
<td>4</td>
<td>H</td>
</tr>
<tr>
<td>5</td>
<td>N</td>
</tr>
<tr>
<td>6</td>
<td>H</td>
</tr>
<tr>
<td>7</td>
<td>H</td>
</tr>
<tr>
<td>8</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>O</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
</tbody>
</table>
Column Selection by Column Name

<table>
<thead>
<tr>
<th></th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>H</td>
</tr>
<tr>
<td>2</td>
<td>H</td>
</tr>
<tr>
<td>3</td>
<td>H</td>
</tr>
<tr>
<td>4</td>
<td>H</td>
</tr>
<tr>
<td>5</td>
<td>N</td>
</tr>
<tr>
<td>6</td>
<td>H</td>
</tr>
<tr>
<td>7</td>
<td>H</td>
</tr>
<tr>
<td>8</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>O</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
</tbody>
</table>
Slicing by Row and Column Using loc

```
structs_gdf.loc[14:21, ['x', 'y', 'z']]
```

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>-1.661639</td>
<td>0.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>15</td>
<td>1.661639</td>
<td>0.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>16</td>
<td>-0.013324</td>
<td>1.132486</td>
<td>0.008276</td>
</tr>
<tr>
<td>17</td>
<td>0.002311</td>
<td>-0.019159</td>
<td>0.001929</td>
</tr>
<tr>
<td>18</td>
<td>-0.027803</td>
<td>2.198949</td>
<td>0.014154</td>
</tr>
<tr>
<td>19</td>
<td>-0.018704</td>
<td>1.525582</td>
<td>0.010433</td>
</tr>
<tr>
<td>20</td>
<td>0.002104</td>
<td>-0.003882</td>
<td>0.001999</td>
</tr>
<tr>
<td>21</td>
<td>0.994873</td>
<td>1.939743</td>
<td>0.002941</td>
</tr>
</tbody>
</table>
Slicing by Row and Column Using loc

```python
structs_gdf.loc[14:21, ['x', 'y', 'z']]```

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>-1.661639</td>
<td>0.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>15</td>
<td>1.661639</td>
<td>0.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>16</td>
<td>-0.013324</td>
<td>1.132466</td>
<td>0.008276</td>
</tr>
<tr>
<td>17</td>
<td>0.002311</td>
<td>-0.019159</td>
<td>0.001929</td>
</tr>
<tr>
<td>18</td>
<td>-0.027803</td>
<td>2.198949</td>
<td>0.014154</td>
</tr>
<tr>
<td>19</td>
<td>-0.018704</td>
<td>1.525582</td>
<td>0.010433</td>
</tr>
<tr>
<td>20</td>
<td>0.002104</td>
<td>-0.003882</td>
<td>0.001999</td>
</tr>
<tr>
<td>21</td>
<td>0.994873</td>
<td>1.939743</td>
<td>0.002941</td>
</tr>
</tbody>
</table>
Compute a New Column “norm”

gdf['norm'] = gdf.x**2 + gdf.y**2 + gdf.z**2

gdf.head()

<table>
<thead>
<tr>
<th>molecule_name</th>
<th>atom_index</th>
<th>atom</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>atom_codes</th>
<th>norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>dsgdb9nsd_000001</td>
<td>0</td>
<td>C</td>
<td>-0.012698</td>
<td>1.085804</td>
<td>0.008001</td>
<td>0</td>
<td>1.179196</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>1</td>
<td>H</td>
<td>0.002150</td>
<td>-0.006031</td>
<td>0.001976</td>
<td>2</td>
<td>0.000045</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>2</td>
<td>H</td>
<td>1.011731</td>
<td>1.463751</td>
<td>0.000277</td>
<td>2</td>
<td>3.166167</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>3</td>
<td>H</td>
<td>-0.540815</td>
<td>1.447527</td>
<td>-0.876644</td>
<td>2</td>
<td>3.156318</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>4</td>
<td>H</td>
<td>-0.523814</td>
<td>1.437933</td>
<td>0.906397</td>
<td>2</td>
<td>3.163587</td>
</tr>
</tbody>
</table>
Compute a New Column “norm”

gdf[‘norm’] = gdf.x**2 + gdf.y**2 + gdf.z**2

gdf.head()

<table>
<thead>
<tr>
<th>molecule_name</th>
<th>atom_index</th>
<th>atom</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>atom_codes</th>
<th>norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>dsgdb9nsd_000001</td>
<td>0</td>
<td>C</td>
<td>-0.012698</td>
<td>1.085804</td>
<td>0.008001</td>
<td>0</td>
<td>1.179196</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>1</td>
<td>H</td>
<td>0.002150</td>
<td>-0.006031</td>
<td>0.001976</td>
<td>2</td>
<td>0.000045</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>2</td>
<td>H</td>
<td>1.011731</td>
<td>1.463751</td>
<td>0.000277</td>
<td>2</td>
<td>3.166167</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>3</td>
<td>H</td>
<td>-0.540815</td>
<td>1.447527</td>
<td>-0.876644</td>
<td>2</td>
<td>3.156318</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>4</td>
<td>H</td>
<td>-0.523814</td>
<td>1.437933</td>
<td>0.906397</td>
<td>2</td>
<td>3.163587</td>
</tr>
</tbody>
</table>
Construct a Boolean Series

```python
$\text{gdf.norm} > 3.0$
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>False</td>
</tr>
<tr>
<td>1</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>True</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
</tr>
<tr>
<td>4</td>
<td>True</td>
</tr>
<tr>
<td>5</td>
<td>False</td>
</tr>
<tr>
<td>6</td>
<td>False</td>
</tr>
<tr>
<td>7</td>
<td>False</td>
</tr>
<tr>
<td>8</td>
<td>False</td>
</tr>
<tr>
<td>9</td>
<td>False</td>
</tr>
<tr>
<td>10</td>
<td>False</td>
</tr>
<tr>
<td>11</td>
<td>False</td>
</tr>
</tbody>
</table>
Select Data With Boolean Series

```python
select_columns = ['molecule_name', 'atom']
gdf[select_columns][gdf.norm > 3.0]
```

<table>
<thead>
<tr>
<th>molecule_name</th>
<th>atom</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 dsgdb9nsd_000001</td>
<td>H</td>
</tr>
<tr>
<td>3 dsgdb9nsd_000001</td>
<td>H</td>
</tr>
<tr>
<td>4 dsgdb9nsd_000001</td>
<td>H</td>
</tr>
<tr>
<td>14 dsgdb9nsd_000004</td>
<td>H</td>
</tr>
<tr>
<td>15 dsgdb9nsd_000004</td>
<td>H</td>
</tr>
<tr>
<td>18 dsgdb9nsd_000005</td>
<td>H</td>
</tr>
<tr>
<td>21 dsgdb9nsd_000007</td>
<td>H</td>
</tr>
<tr>
<td>22 dsgdb9nsd_000007</td>
<td>H</td>
</tr>
</tbody>
</table>
Unique Elements

gdf.atom.unique()

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>H</td>
</tr>
<tr>
<td>3</td>
<td>N</td>
</tr>
<tr>
<td>4</td>
<td>O</td>
</tr>
</tbody>
</table>

Name: atom, dtype: category
Categories (5, object): [C, F, H, N, O]
User-Defined Functions

def label_mass(atom_codes, mass, kwarg1):
    for i, atom in enumerate(atom_codes):
        if atom == 0:
            mass[i] = 12.011
        elif atom == 3:
            mass[i] = 14.007
        elif atom == 4:
            mass[i] = 15.999
        elif atom == 1:
            mass[i] = 18.998
        elif atom == 2:
            mass[i] = 1.008
        else:
            mass[i] = -1.0
User-Defined Functions

def label_mass(atom_codes, mass, kwarg1):
    for i, atom in enumerate(atom_codes):
        if atom==0:
            mass[i]=12.011
        elif atom==3:
            mass[i]=14.007
        elif atom==4:
            mass[i]=15.999
        elif atom==1:
            mass[i]=18.998
        elif atom==2:
            mass[i]=1.008
        else:
            mass[i]=-1.0

gdf2 = gdf.apply_rows(label_mass,
                      incols=['atom_codes'],
                      outcols=dict(mass=np.float32),
                      kwargs=dict(kwarg1=1))
User-Defined Functions

gdf2 = gdf.apply_rows(label_mass,
    incols=[['atom_codes'],
    outcols=dict(mass=np.float32),
    kwargs=dict(kwarg1=1))

<table>
<thead>
<tr>
<th>molecule_name</th>
<th>atom_index</th>
<th>atom</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>atom_codes</th>
<th>norm</th>
<th>mass</th>
</tr>
</thead>
<tbody>
<tr>
<td>dsgdb9nsd_000001</td>
<td>0</td>
<td>C</td>
<td>-0.012698</td>
<td>1.085804</td>
<td>0.008001</td>
<td></td>
<td>1.179196</td>
<td>12.011</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>1</td>
<td>H</td>
<td>0.002150</td>
<td>-0.006031</td>
<td>0.001976</td>
<td></td>
<td>0.000045</td>
<td>1.008</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>2</td>
<td>H</td>
<td>1.011731</td>
<td>1.463751</td>
<td>0.000277</td>
<td></td>
<td>3.166167</td>
<td>1.008</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>3</td>
<td>H</td>
<td>-0.540815</td>
<td>1.447527</td>
<td>-0.876644</td>
<td></td>
<td>3.156318</td>
<td>1.008</td>
</tr>
<tr>
<td>dsgdb9nsd_000001</td>
<td>4</td>
<td>H</td>
<td>-0.523814</td>
<td>1.437933</td>
<td>0.906397</td>
<td></td>
<td>3.163587</td>
<td>1.008</td>
</tr>
<tr>
<td>dsgdb9nsd_000002</td>
<td>0</td>
<td>N</td>
<td>-0.040426</td>
<td>1.024108</td>
<td>0.062564</td>
<td></td>
<td>1.054345</td>
<td>14.007</td>
</tr>
</tbody>
</table>
Large set of vectorized operations are fast and convenient.

Kernels can be used to define custom vector functions and expose CUDA parallelism.

pandas provides a large set of **vector functions** that operate on all columns of a DataFrame or a single selected column (cuDF Series). These functions produce vectors of values for each of the columns, or a single Series for the individual Series. Examples:

- `max(axis=1)`
  - Element-wise max.

- `min(axis=1)`
  - Element-wise min.

- `clip(lower=-10, upper=10)`
  - Trim values at input thresholds

- `abs()`
  - Absolute value.

**Define a kernel function:**

```python
>>> def kernel(in1, in2, in3, out1, out2, extra1, extra2):
    for i, (x, y, z) in enumerate(zip(in1, in2, in3)):
        out1[i] = extra2 * x - extra1 * y
        out2[i] = y - extra1 * z
```

**Call the kernel with apply_rows:**

```python
>>> outdf = gdf.apply_rows(kernel,
                          incols=['in1', 'in2', 'in3'],
                          outcols=dict(out1=np.float64,
                                        out2=np.float64),
                          kwargs=dict(extra1=2.3, extra2=3.4))
```
# Read some data into a GPU DataFrame
train_gdf = gd.read_csv('~/data/train.csv')

# Do some string operations: split molecule_42 on '_', take int(42)
train_gdf['mol_id'] = train_gdf.molecule_name.str.split('_')[1].astype('int')

# hash by molecule name
train_gdf['hash'] = train_gdf['molecule_name'].hash_values()

# create bool mask on hash mod 4
mask = train_gdf['hash']%4==0

# split by mask
tr, va = train_gdf[~mask], train_gdf[mask]
Simple supervised ML algorithm that classifies a test point (green) based on the labels of its nearest neighbors within a radius, K. If the radius is 5 (solid circle) the test point will be labeled as a red triangle because there are two red triangles within the circle. If the chosen radius is 7 (dashed circle) the test point will be labeled as a blue square.
Density-based clustering algorithm: given a set of points in some space, it groups together points that are closely packed together (points with many nearby neighbors), marking as outliers points that lie alone in low-density regions (whose nearest neighbors are too far away). DBSCAN is one of the most common clustering algorithms and also most cited in scientific literature.
Unsupervised algorithm that assigns divides unlabeled data into K clusters. It is an iterative algorithm that assigns each data point to a cluster based on its proximity to the centroids of the current clusters. This (re)assignment perturbs the centroids of each cluster. Iteration continues until assignments stabilize.
A matrix factorization that expresses matrix $M$ as a product:

$$M = U \Sigma V^*$$

Truncated version is an approximate decomposition. Applications: Signal processing, data compression, image processing, satellite communication, wireless communication, least-squares fitting.
An orthogonal transformation that identifies the basis which contains the highest variance.

Applications: Compression, dimensionality reduction, signal processing, neuroscience, statistics, exploratory data analysis, and many disciplines of science and engineering.
Supervised decision tree ensemble algorithm that implements gradient boosting.

Can be used for regression or classification. Extremely popular, especially on Kaggle.
CURRENTLY SUPPORTED ALGORITHMS
v0.13.0a

- Density-Based Spatial Clustering of Applications with Noise (DBSCAN)
- K-Means
- Principal Components Analysis (PCA)
- Truncated Singular Value Decomposition (tSVD)
- Uniform Manifold Approximation and Projection (UMAP)
- t-Distributed Stochastic Neighbor Embedding (TSNE)
- Regression (OLS)
- Linear Regression with Lasso or Ridge Regularization
- ElasticNet Regression
- Logistic Regression
- Stochastic Gradient Descent (SGD), (CD), (QN) (including L-BFGS and OWL-QN) solvers for linear models
- Random Forest (RF) Classification
- Random Forest (RF) Regression
- K-Nearest Neighbors (KNN) Classification
- K-Nearest Neighbors (KNN) Regression
- Support Vector Machine Classifier (SVC)
- Linear Kalman Filter
- Holt-Winters Exponential Smoothing
- Auto-regressive Integrated Moving Average (ARIMA)
50+ GPU-OPTIMIZED SOFTWARE CONTAINERS

DEEP LEARNING
TensorFlow | PyTorch | more

MACHINE LEARNING
RAPIDS | H2O | more

INFERENCETensorRT | DeepStream | more

HPC
NAMD | GROMACS | more

GENOMICS
Parabricks

VISUALIZATION
ParaView | IndeX | more