Distributed Computing with GPUs in Python

Getting Started Cheat Sheet

High-Performance Jupyter Notebooks – Managed GPU environment.
Scale up as needed. Get started in minutes with no setup required.
Try the free notebook with examples here: https://cutt.ly/rapids-cheatsheets-dask4beginners

For additional cheat sheets go to: **nvidia.com/rapids-kit/**

CLUSTER SETUP

Start the Dask cluster.

```
from dask_cuda import LocalCUDACluster
```

cluster = LocalCUDACluster()" - Create a Dask cluster with a single GPU worker. This is the easiest way to start developing with distributed RAPIDS using Dask on a single-GPU.

```
cluster = LocalCUDACluster(
   n_workers=2
   , threads_per_worker=1
   , CUDA_VISIBLE_DEVICES="0,1"
   , rmm_managed_memory=True
   , rmm_pool_size="20GB"
)
```

- Create a Dask cluster with 2 GPU workers if the machine has at least 2 GPUs available. Starting the Dask cluster this way uses the RAPIDS Memory Manager with an initial pool of 20GB.

bash: dask-scheduler - Start the Dask scheduler from a command line interface with default parameters.

bash: dask-scheduler --host 127.0.0.1 --port 8786 --dashboard-address 8787 --idle-timeout 600 - Start the Dask scheduler from a command line interface with specifying host IP (or hostname), its port, the port of the Dask dashboard, and the auto-suspend of the Dask cluster.

bash: dask-cuda-worker <scheduler-ip>:8786 - Start the Dask worker and register it with the running Dask scheduler on port 8786.

bash: dask-cuda-worker --worker-port=3000:3001 --death-timeout 600 <scheduler-ip>:8786 - Start the Dask worker and register it with the running Dask scheduler on port 8786. This call also specifies a range of ports the worker should use as well the auto-suspend time of the Dask worker.

CLIENT

Manage the Dask client.

```
from dask.distributed import Client
```

client = Client(cluster) - Create a Dask client and connect it to the cluster by passing a cluster object (cluster can be either LocalCUDACluster or a remote cluster).

client = Client('<scheduler>:<scheduler-port>') - Create a Dask client and connect it to the cluster by passing IP, hostname, or address of a scheduler and port as string.

client.restart() - Restart a Dask cluster and clear memory.

client.scheduler_info() - Extract all the cluster information as a dictionary. The information includes the scheduler and dashboard address and port, as well as a dictionary of all workers with the related information.

client.upload_file('<file>.egg') - Use a Python package on a Dask cluster. If a custom package is used in the code, every worker in the cluster needs to be able to import that package. This function puts the uploaded file in the PATH so the worker Python process can find it and import it.

client.shutdown() - Shut down the connected scheduler and workers.

DATAFRAME

Perform computations using Dask cuDF DataFrames.

ddf = dask_cudf.from_cudf(df, npartitions=2) - Create a Dask cuDF DataFrame from a cuDF DataFrame with two partitions.

ddf = dask_cudf.from_cudf(df, chunksize=1000) - Create a Dask cuDF DataFrame from a cuDF Dataframe with each partitions containing 1,000 rows.

```
def process_frame(df):
   df['num_inc'] = df['number'] + 10
   return df
```

ddf.map_partitions(process_frame) - Apply a function to each frame in the distributed DataFrame and receive modified frames back.

```
def multiply(a, b, mult):
   for i, (aa, bb) in enumerate(zip(a, b)):
      mult[i] = aa * bb

def process_frame_mul(df):
   df = df.apply_rows(
      multiply
      , incols = {'number': 'a', 'float_number': 'b'}
      , outcols = {'mult': np.float64}
      , kwargs = {}
   )
   return df['mult']
```

ddf.map_partitions(process_frame_mul) - Apply a function to each frame in the distributed DataFrame that calls a custom RAPIDS kernel on data and receive modified frames back.

```
def divide(a, div, b):
   for i, aa in enumerate(a):
      div[i] = aa / b

def process_frame_div(df, col_a, val_divide):
   df = df.apply_rows(
      divide
      , incols = {col_a: 'a'}
      , outcols = {'div': np.float64}
      , kwargs = {'b': val_divide}
   )
   return df['div']
```

ddf['div_number'] = ddf.map_partitions(process_frame_div, 'number', 10.0)
ddf['div_float'] = ddf.map_partitions(process_frame_div, 'float_number', 5.0)
- Apply a parametrized function to each frame in the distributed DataFrame that calls a custom RAPIDS kernel on the data and receive modified frames back.

ddf['div_number'] = ddf.map_partitions(lambda df: process_frame_div(df, 'number', 10.0))
ddf['div_float'] = ddf.map_partitions(lambda df: process_frame_div(df, 'float_number', 5.0)) - Apply a parametrized function using lambda to each frame in the distributed DataFrame that calls a custom RAPIDS kernel on data and receive modified frames back.

DATAFRAME

Perform computations using Dask cuDF DataFrames.

`ddf.compute()` - Run the computations and coalesce the results into a cuDF DataFrame.

`ddf.persist()` - Execute the Dask task graph and cache the results in memory on workers but not return the results to the headnode. Persisting the data intelligently in memory can have a profound impact on performance.

DELAYED

Build a delayed execution graph to parallelize code.

```
from dask.delayed import delayed

import cupy as cp
def delayed_task(n):
    df = cudf.DataFrame({'random': cp.random.rand(n)})
    df['rand_scaled'] = df['random'] * 3
    return df
```

`tasks = [delayed(delayed_task)(10) for _ in range(2)]` - Parallelize code and build the task graph directly with Dask. If your solution does not fit the DataFrame framework directly, this approach can still utilize Dask to parallize computations. These tasks are executed lazily.

```
@delayed
def delayed_task(n):
    df = cudf.DataFrame({'random': cp.random.rand(n)})
    df['rand_scaled'] = df['random'] * 3
    return df
```

`tasks = [delayed_task(10) for _ in range(2)]` - Parallelize code and build the task graph directly with Dask with a function decorator.

```
def process_frame_delayed(df):
    return df['number'] + 10

ddf_delayed_add = dask_cudf.from_delayed([
    process_frame_delayed(df)
    for df
    in ddf.to_delayed()
])
```

- Recreate the Dask cuDF DataFrame from delayed objects.

FUTURE

Control immediate execution of parallel code.

`client.persist(ddf)` - Execute the Dask task graph immediately and persist the intermediate result in memory.

```
def first_computation(df):
    return df['number'] + 10

def second_computation(result):
    return result / 10.0

computation_1 = client.submit(first_computation, ddf)
computation_2 = client.submit(second_computation, computation_1)
```

- Submit tasks to run in parallel with dependency between tasks. Dask will wait for the results of the first computation to finish before running the second task.

`computation.result().compute()` - Process the data and return the distributed result onto the headnode as a cuDF DataFrame.

```
computation = client.compute(tasks, optimize_graph=True)
dask.distributed.wait(computation)
```

- Await for the set of tasks to finish before progressing.

```
computation = client.compute(tasks, optimize_graph=True)

for part in dask.distributed.as_completed(computation):
    print(part.result())
```

- Process the data as the individual tasks get finished.

`computation.result()` - Retrieve the result of a finished task.

`computation.done()` - Check if a Dask future has finished executing the task.

`client.gather(computation)` - Collect the future objects from distributed memory.

`client.scatter(data)` - Send the futures to workers.

`computation.cancel()` - Cancel the computation.