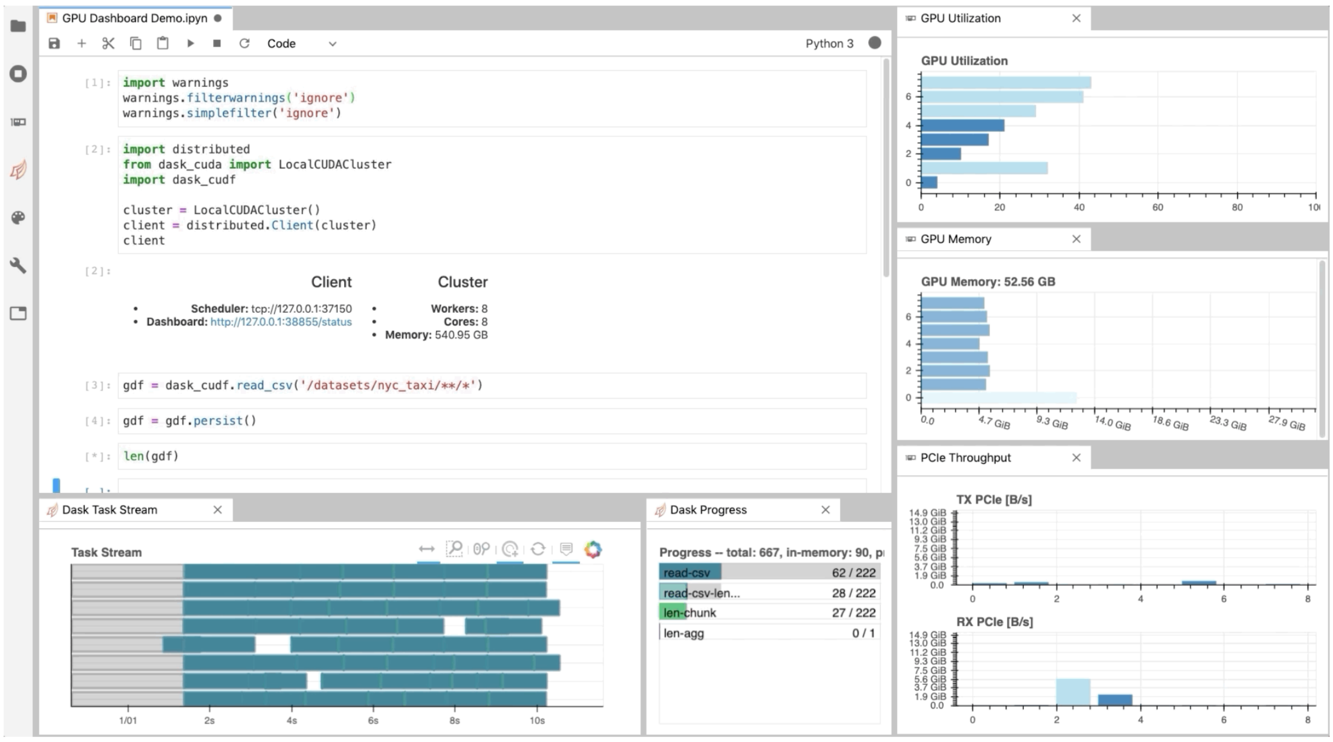
To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

GPU Dashboards in Jupyter Lab

Jacob Tomlinson
Dec 13, 2019 · 8 min read

Authors: Jacob Tomlinson, Rick Zamora and Ken Hester



```
[1]: import warnings
     warnings.filterwarnings('ignore')
     warnings.simplefilter('ignore')

[2]: import distributed
     from dask_cuda import LocalCUDACluster
     import dask_cudf

     cluster = LocalCUDACluster()
     client = distributed.Client(cluster)
     client

[2]:
          Client                                        Cluster
  • Scheduler: tcp://127.0.0.1:37150          • Workers: 8
  • Dashboard: http://127.0.0.1:38855/status  • Cores: 8
                                              • Memory: 540.95 GB

[3]: gdf = dask_cudf.read_csv('/datasets/nyc_taxi/**/*')

[4]: gdf = gdf.persist()

[*]: len(gdf)
```

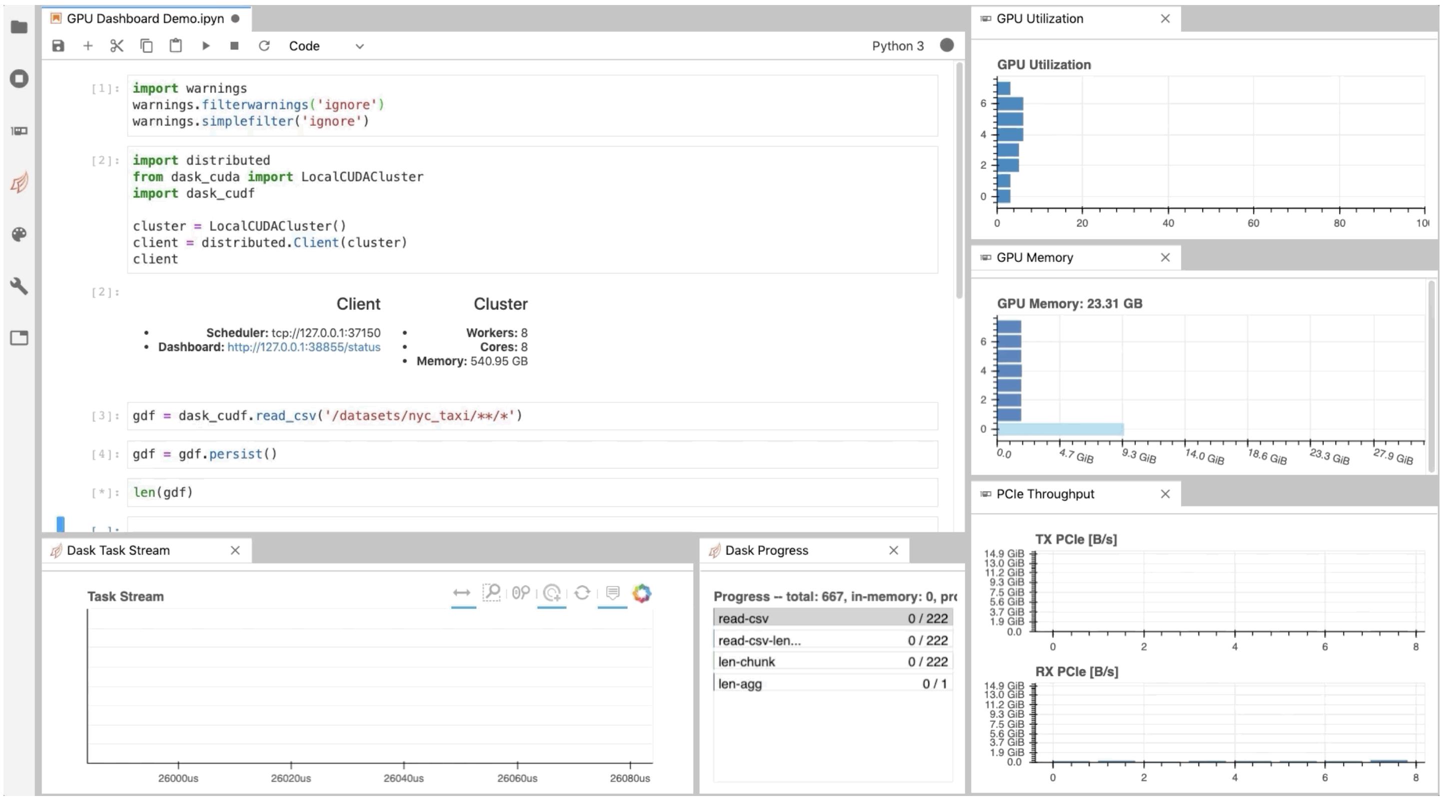
Introduction

We are excited to announce NVDashboard, an open-source package for the real-time visualization of NVIDIA GPU metrics in interactive Jupyter environments. NVDashboard is a great way for all GPU users to monitor system resources, but it is especially valuable for users of RAPIDS, NVIDIA's open-source suite of GPU-accelerated data-science software libraries.

Given the computational intensity of modern data-science algorithms, there are many cases in which GPUs can offer game-changing workflow acceleration. To achieve optimal performance, it is absolutely critical for the underlying software to utilize system resources effectively. Although acceleration libraries (like cuDNN and RAPIDS)

To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

actually leveraging GPU resources as intended. While this can be accomplished with command-line tools like nvidia-smi, many professional data scientists prefer to use interactive Jupyter notebooks for day-to-day model and workflow development.



```
[1]: import warnings
     warnings.filterwarnings('ignore')
     warnings.simplefilter('ignore')

[2]: import distributed
     from dask_cuda import LocalCUDACluster
     import dask_cudf

     cluster = LocalCUDACluster()
     client = distributed.Client(cluster)
     client

[2]:
                     Client                              Cluster
  • Scheduler: tcp://127.0.0.1:37150         • Workers: 8
  • Dashboard: http://127.0.0.1:38855/status • Cores: 8
                                             • Memory: 540.95 GB

[3]: gdf = dask_cudf.read_csv('/datasets/nyc_taxi/**/*')

[4]: gdf = gdf.persist()

[*]: len(gdf)
```

```
Progress -- total: 667, in-memory: 0, pro
read-csv          0 / 222
read-csv-len...   0 / 222
len-chunk         0 / 222
len-agg           0 / 1
```

Fig. 1 The NVDashboard Jupyter-Lab extension in action. The GPU dashboards are shown along the right-hand side of the screen, while two dask-labextension dashboards are shown on the bottom left) [GIF].

As illustrated in **Fig. 1**, NVDashboard enables Jupyter notebook users to visualize system hardware metrics within the same interactive environment they use for development. Supported metrics include:

- GPU-compute utilization
- GPU-memory consumption
- PCIe throughput
- NVLink throughput

The package is built upon a Python-based dashboard server, which leverages the Bokeh visualization library to display and update figures in real time. An additional Jupyter-Lab extension embeds these dashboards as movable windows within an interactive

To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

For this reason, the available dashboards can be modified/extended to display any queryable GPU metrics accessible through NVML.

Using NVDashboard

The nvdashboard package is available on PyPI, and consists of two basic components:

- **Bokeh Server**: The server component leverages the wonderful Bokeh visualization library to display and update GPU-diagnostic dashboards in real time. The desired hardware metrics are accessed with PyNVML, an open-source python package composing wrappers for the NVIDIA Management Library (NVML). For this reason, NVDashboard can be modified/extended to display any queryable GPU metrics accessible through NVML, easily from Python.
- **Jupyter-Lab Extension**: The Jupyter-Lab extension allows embedding the GPU-diagnostic dashboards as movable windows within an interactive Jupyter-Lab environment.

The Jupyter Lab Extension

The practice of directly querying hardware metrics is often the best way to validate efficient run-time behavior, and this is especially true for interactive Jupyter-notebook users. In this case, the development process is often iterative, and improper GPU utilization results in huge productivity losses. As shown in **Fig. 1**, NVDashboard makes visualizing resource utilization easy for Jupyter-Lab users right alongside their code.

To install both the server and client-side components, run the following in a terminal:

```
$ pip install jupyterlab-nvdashboard

$ jupyter labextension install jupyterlab-nvdashboard
```

After NVDashborad is installed, a “GPU Dashboards” menu should be visible along the left-hand side of your Jupyter-Lab environment (see **Fig. 2**). Clicking on one of these buttons automatically adds a movable window, with a real-time display of the desired dashboard.

To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

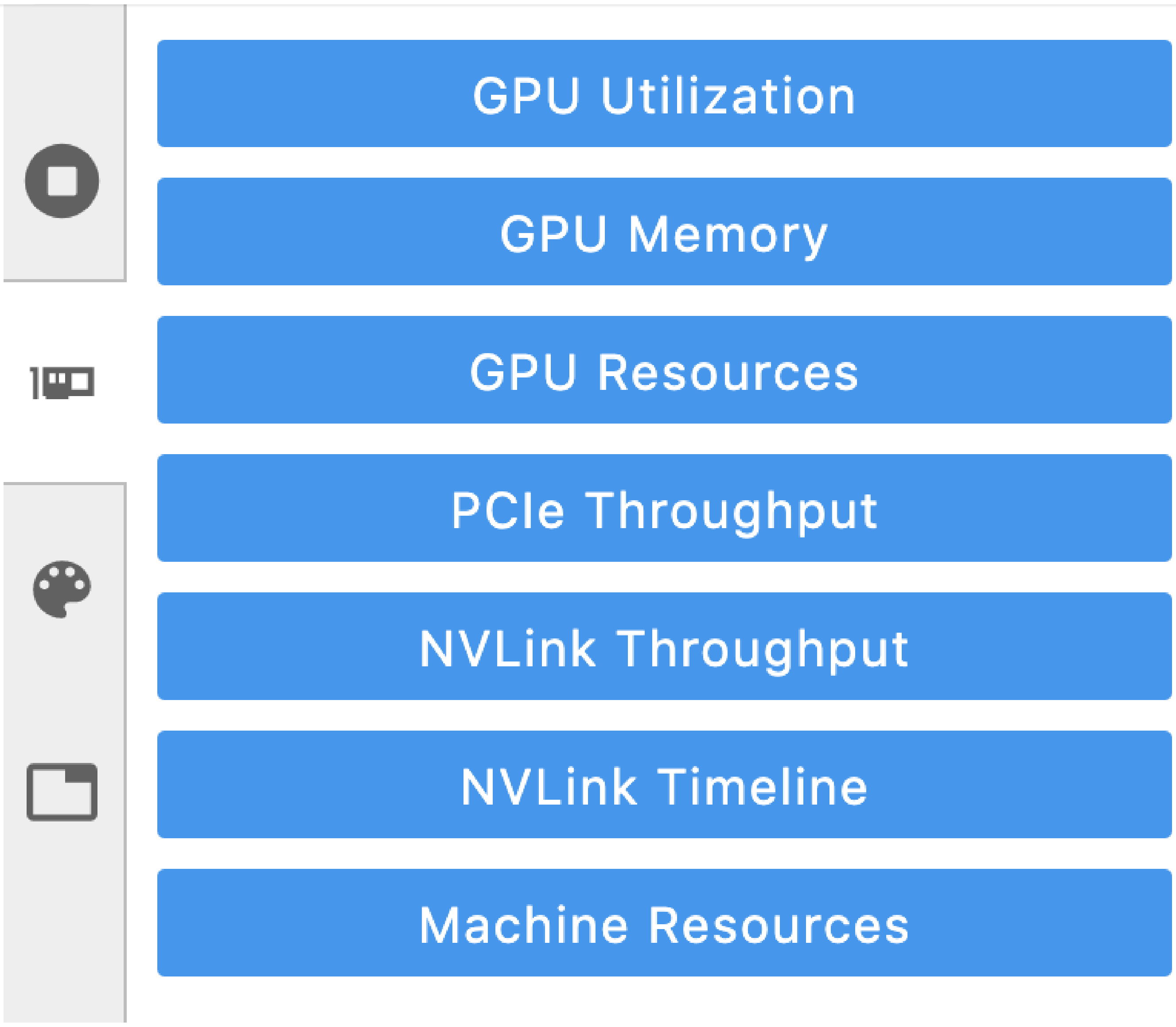
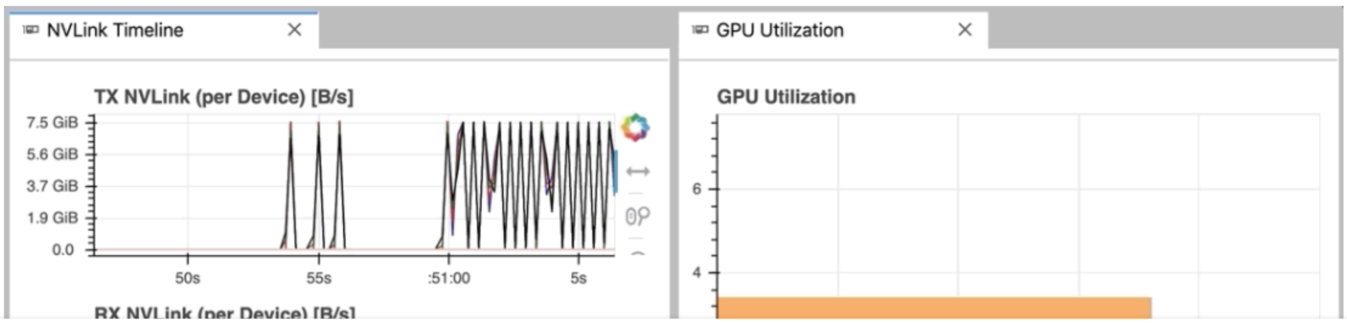


Fig. 2 Main menu for the Jupyter-Lab extension.

It is important to clarify that NVDashboard automatically monitors the GPU resources for the entire machine, not only those being used by the local Jupyter environment. For this reason, the Jupyter-Lab Extension can certainly be used for non-iPython/notebook development. For example, in **Fig. 3**, the "NVLink Timeline" and "GPU Utilization" dashboards are being used within a Jupyter-Lab environment to monitor a multi-GPU deep-learning workflow executed from the command line.



To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

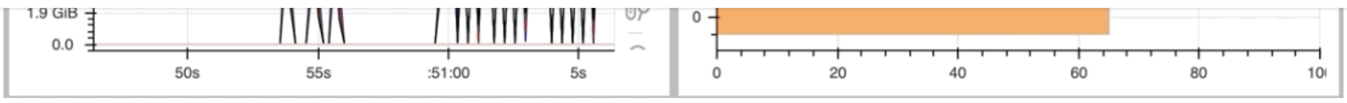


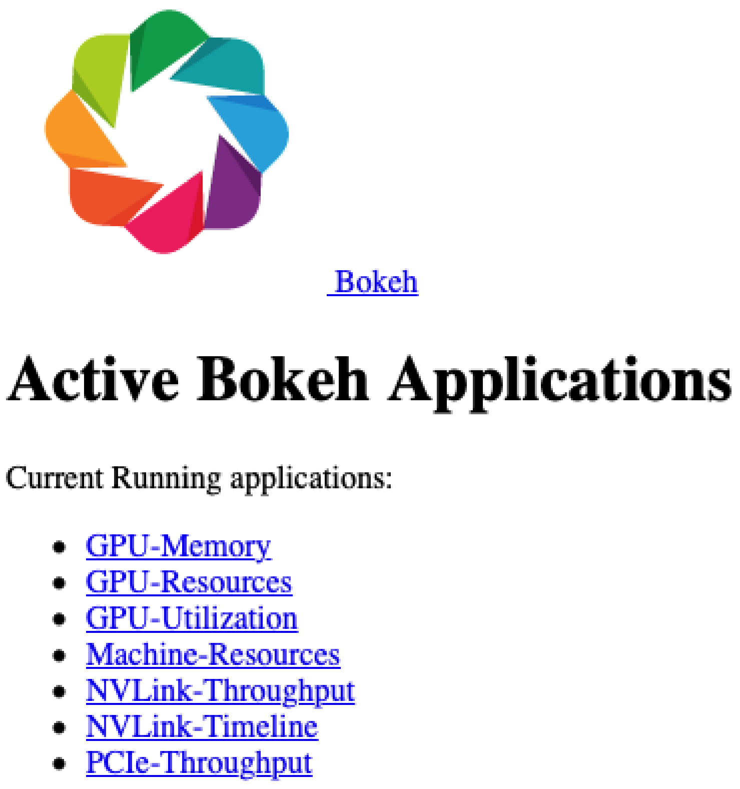
Fig. 3 The "NVLink Timeline" dashboard being used with Jupyter Lab [GIF].

The Bokeh Server

While the Jupyter-Lab extension is certainly ideal for fans of iPython/notebook-based development, other GPU users can also access the dashboards using a stand-alone Bokeh server. This is accomplished by running

```
$ python -m jupyterlab_nvdashboard.server <port-number>
```

After starting the Bokeh server, the GPU dashboards are accessed by opening the appropriate url in a standard web browser (e.g. http://<ip-address>:<port-number>). As shown in **Fig. 4**, the main menu lists all dashboards available in NVDashboard.



Bokeh

Active Bokeh Applications

Current Running applications:

- GPU-Memory
- GPU-Resources
- GPU-Utilization
- Machine-Resources
- NVLink-Throughput
- NVLink-Timeline
- PCIe-Throughput

Fig. 4 The main menu for the Bokeh-server component of NVDashboard.

To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

plots.

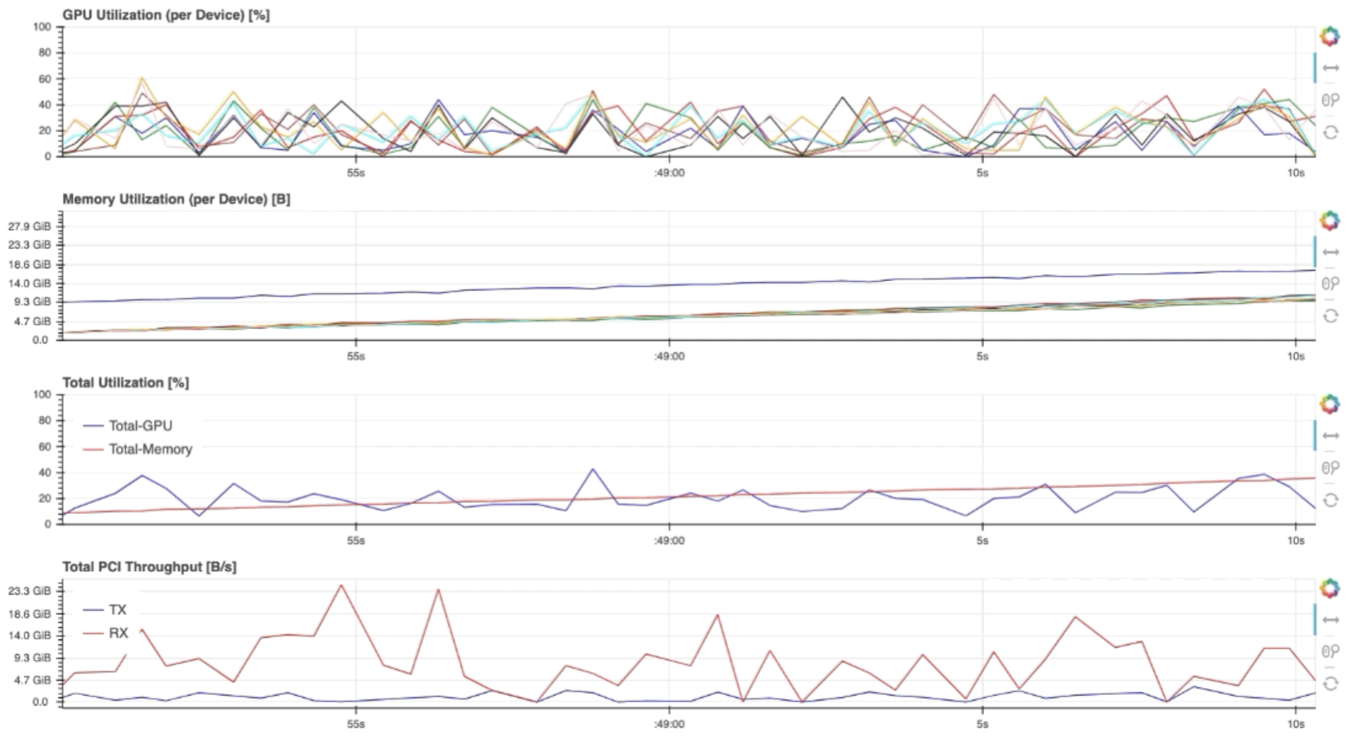


Fig. 5 The "GPU Resources" dashboard being used outside of Jupyter Lab [GIF]

To use NVDashboard in this way, only the pip-installation step is needed (the labextension installation step can be skipped):

```
$ pip install jupyterlab-nvdashboard
```

Alternatively, one can also clone the jupyterlab-nvdashboard repository, and simply execute the `server.py` script (e.g. `python jupyterlab_nvdashboard/server.py <port-number>`).

Implementation Details

The existing nvdashboard package provides a number of useful GPU-resource dashboards. However, it is fairly straightforward to modify existing dashboards and/or create completely new ones. In order to do this, you simply need to leverage PyNVML and Bokeh.

PyNVML Basics

To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

NVML is directly used by the better-known NVIDIA System Management Interface (nvidia-smi). According to the NVIDIA developer site, NVML provides access to the following query-able states (in additional to modifiable states not discussed here):

- **ECC error counts**: Both correctable single bit and detectable double bit errors are reported. Error counts are provided for both the current boot cycle and for the lifetime of the GPU.
- **GPU utilization**: Current utilization rates are reported for both the compute resources of the GPU and the memory interface.
- **Active compute process**: The list of active processes running on the GPU is reported, along with the corresponding process name/id and allocated GPU memory.
- **Clocks and PState**: Max and current clock rates are reported for several important clock domains, as well as the current GPU performance state.
- **Temperature and fan speed**: The current core GPU temperature is reported, along with fan speeds for non-passive products.
- **Power management**: For supported products, the current board power draw and power limits are reported.
- **Identification**: Various dynamic and static information is reported, including board serial numbers, PCI device ids, VBIOS/Inforom version numbers and product names.

Although several different python wrappers for NVML currently exist, we use the PyNVML package hosted by GoAi on GitHub. This version of PyNVML uses ctypes to wrap most of the NVML C API. NVDashboard utilizes only a small subset of the API needed to query real-time GPU-resource utilization, including:

- `nvmlInit()` : Initialize NVML. Upon successful initialization, the GPU handles are cached to lower the latency of data queries during active monitoring in a dashboard.
- `nvmlShutdown()` : Finalize NVML

To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

- `nvmlDeviceGetHandleByIndex()` : Get a handle for a device (given an integer index)
- `nvmlDeviceGetMemoryInfo()` : Get a memory-info object (given a device handle)
- `nvmlDeviceGetUtilizationRates()` : Get a utilization-rate object (given a device handle)
- `nvmlDeviceGetPcieThroughput()` : Get a PCIe-throughput object (given a device handle)
- `nvmlDeviceGetNvLinkUtilizationCounter()` : Get an NVLink utilization counter (given a device handle and link index)

In the current version of PyNVML, the python function names are usually chosen to exactly match the C API. For example, to query the current GPU-utilization rate on every available device, the code would look something like this:

```
In [1]: from pynvml import *

In [2]: nvmlInit()

In [3]: ngpus = nvmlDeviceGetCount()

In [4]: for i in range(ngpus):

…: handle = nvmlDeviceGetHandleByIndex(i)

…: gpu_util = nvmlDeviceGetUtilizationRates(handle).gpu

…: print('GPU %d Utilization = %d%%' % (i, gpu_util))

…:

GPU 0 Utilization = 43%

GPU 1 Utilization = 0%

GPU 2 Utilization = 15%

GPU 3 Utilization = 0%

GPU 4 Utilization = 36%

GPU 5 Utilization = 0%
```

To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

Note that, in addition to the GitHub repository, PyNVML is also hosted on PyPI and Conda Forge.

Dashboard Code

In order to modify/add a GPU dashboard, it is only necessary to work with two files (`jupyterlab_bokeh_server/server.py` and `jupyterlab_nvdashboard/apps/gpu.py`). Most of the PyNVML and bokeh code needed to add/modify a dashboard will be in `gpu.py` . It is only necessary to modify `server.py` in the case that you are adding or changing a menu/display name. In this case, the new/modified name must be specified in routes dictionary (with the key being the desired name, and the value being the corresponding dashboard definition):

```
routes = {
    "/GPU-Utilization": apps.gpu.gpu,
    "/GPU-Memory": apps.gpu.gpu_mem,
    "/GPU-Resources": apps.gpu.gpu_resource_timeline,
    "/PCIe-Throughput": apps.gpu.pci,
    "/NVLink-Throughput": apps.gpu.nvlink,
    "/NVLink-Timeline": apps.gpu.nvlink_timeline,
    "/Machine-Resources": apps.cpu.resource_timeline,
}
```

In order for the server to constantly refresh the PyNVML data used by the bokeh applications, we use bokeh's ColumnDataSource class to define the *source* of data in each of our plots. The ColumnDataSource class allows an update function to be passed for each type of data, which can be called within a dedicated callback function (cb) for each application. For example, the existing gpu application is defined like this:

```
def gpu(doc):
    fig = figure(title="GPU Utilization", sizing_mode="stretch_both", x_range=[0, 100]

    def get_utilization():
        return [
            pynvml.nvmlDeviceGetUtilizationRates(gpu_handles[i]).gpu
            for i in range(ngpus)
        ]

```

To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

```
    mapper = LinearColorMapper(palette=all_palettes["RdYlBu"][4], low=0, high=100)
    fig.hbar(
        source=source,
        y="right",
        right="gpu",
        height=0.8,
        color={"field": "gpu", "transform": mapper},
    )
    fig.toolbar_location = None
    doc.title = "GPU Utilization [%]"
    doc.add_root(fig)

    def cb():
        source.data.update({"gpu": get_utilization()})

    doc.add_periodic_callback(cb, 200)
```

nvdashboard_example.py hosted with ❤ by **GitHub** **view raw**

Note that the real-time update of PyNVML GPU-utilization data is performed within the `source.data.update()` call. With the necessary `ColumnDataSource` logic in place, the standard GPU definition (above) can be modified in many ways. For example, swapping the x and y axes, specifying a different color palette, or even changing the figure from an hbar to something else entirely.

Users should feel free to open a pull request to contribute valuable improvements/additions — Community engagement is certainly encouraged!

Jupyter Lab Extension Code

In order to package this up as a Jupyter Lab extension we need our bokeh server to be run when Jupyter Lab starts. We can do this by adding jupyter-server-proxy as a dependency and registering an entrypoint in our setup.py

```
entry_points={
    'jupyter_serverproxy_servers': [
        'nvdashboard = jupyterlab_nvdashboard:launch_server',
    ]
},
```

To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

Then we can write a JavaScript frontend extension for Jupyter Lab which adds a sidebar menu with a list of dashboards which are opened as IFrames in new tabs in the interface. This is installed by running jupyter labextension install jupyterlab-nvdashboard.

There is also a custom Bokeh server endpoint in the Python side of things which serves a list of all the available dashboards and their URLs as a json file at `/nvdashboard/index.json`. The front end can use this json file to populate the menu automatically. This means that if we want to add any new dashboards we can do everything on the Python side by adding new Bokeh apps and the Jupyter Lab extension will pick them up automatically.

Python Gpu Jupyterlab Dashboard Open Source Software

About Help Legal