Emmanuel Tchoumkeu

GPU Computing with Python

Scalable Computing Systems and Applications in AI, BigData and HPC

with coordinator Prof. Dr. Julian Kunkel

Table of contents

- GPU was first invented by NVidia in 1999.
  Originally GPUs were purely fixed-function devices, meaning that they were designed to **specifically process stages of graphics pipeline** such as vertex and pixel shaders.
- A Central Processing Unit (**CPU**) is a latency-optimized general purpose processor that is designed to handle a wide range of distinct tasks sequentially,
- The GPU consists of an **array of Streaming Multiprocessors (SM)**, each of which is capable of supporting thousands of co-resident concurrent hardware threads, up to $2048 = 2^{11}$ on modern architecture GPUs.

The GWDG Scientific Compute Cluster

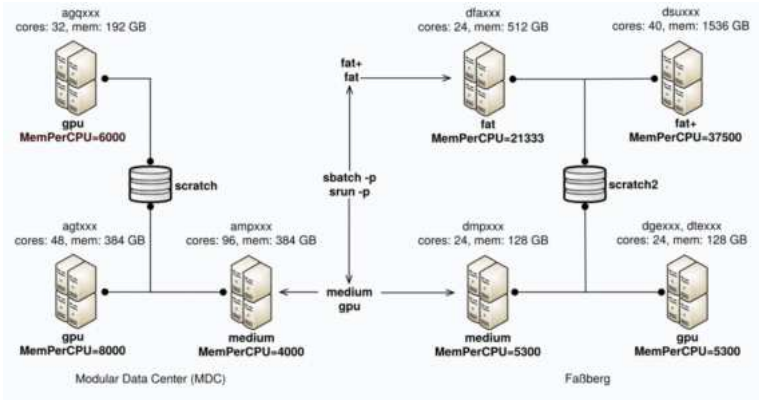


Image source: https://docs.gwdg.de

Available Partitions on the SCC, "EMMY" (2022)

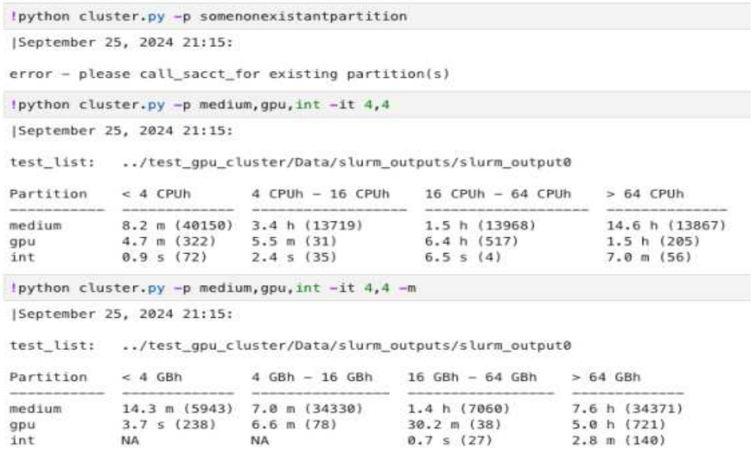
- on the GWDG Scientific Compute Cluster (SCC)
  - General Partitions
    - medium general purpose partition, well suited for most jobs. Up to 1024 cores per job.
    - fat up to 512 GB in one host.
    - fat+ for extreme memory requirements. Up to 2048GB per host.
  - Special Purpose Partitions
    - gpu for jobs using GPU acceleration.
    - int for interactive jobs, i.e. jobs which require a shell or a GUI.

Goals

Question: How many flops per transferred value each partition would provide so that running the code in parallel on GPU will increase performance?

- Here are the flop rates and testing assumptions:
  1. Data transfer from CPU to GPU is worth it at a rate of 15 **GB/s**.
  2. GPU is able to perform at 1 **Terabytes/second** (1000 GB).
  3. CPU is able to perform at 1 **Gigabytes/second**.
  4. Data is originally better placed in the CPU.
- How would a problem like this be solved step by step?

Uses Cases: Reaching GPU (on CLOUD) with Python

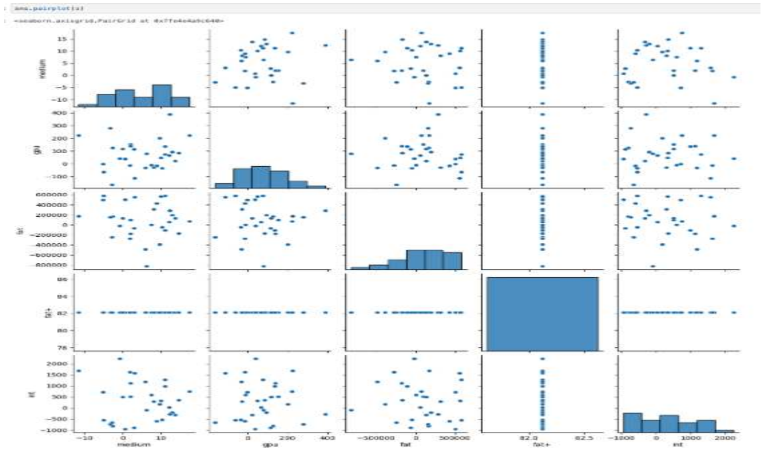


```
!python cluster.py -p somenonexistantpartition
|September 25, 2024 21:15:

error - please call_sacct_for existing partition(s)
```

```
!python cluster.py -p medium,gpu,int -it 4,4
|September 25, 2024 21:15:

test_list:   ../test_gpu_cluster/Data/slurm_outputs/slurm_output0

Partition    < 4 CPUh       4 CPUh - 16 CPUh    16 CPUh - 64 CPUh    > 64 CPUh
-----------  -------------  ------------------  -------------------  ---------------
medium       8.2 m (40150)  3.4 h (13719)       1.5 h (13968)        14.6 h (13867)
gpu          4.7 m (322)    5.5 m (31)          6.4 h (517)          1.5 h (205)
int          0.9 s (72)     2.4 s (35)          6.5 s (4)            7.0 m (56)
```

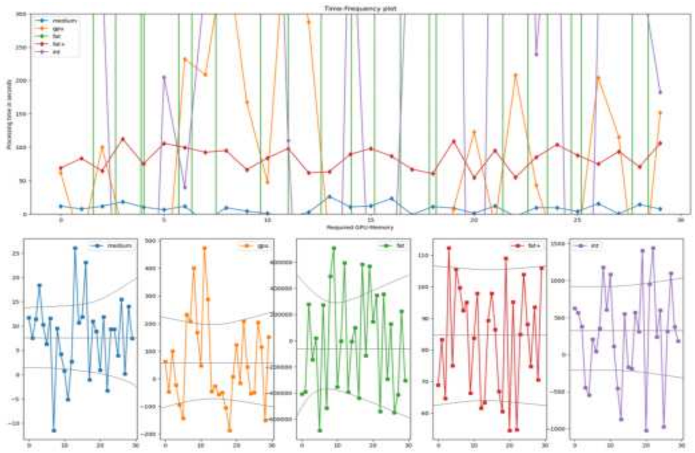
```
!python cluster.py -p medium,gpu,int -it 4,4 -m
|September 25, 2024 21:15:

test_list:   ../test_gpu_cluster/Data/slurm_outputs/slurm_output0

Partition    < 4 GBh        4 GBh - 16 GBh    16 GBh - 64 GBh    > 64 GBh
-----------  -------------  ----------------  -----------------  -------------
medium       14.3 m (5943)  7.0 m (34330)     1.4 h (7060)       7.6 h (34371)
gpu          3.7 s (238)    6.6 m (78)        30.2 m (38)        5.0 h (721)
int          NA             NA                0.7 s (27)         2.8 m (140)
```

Performance Analysis (Plotting distributions)

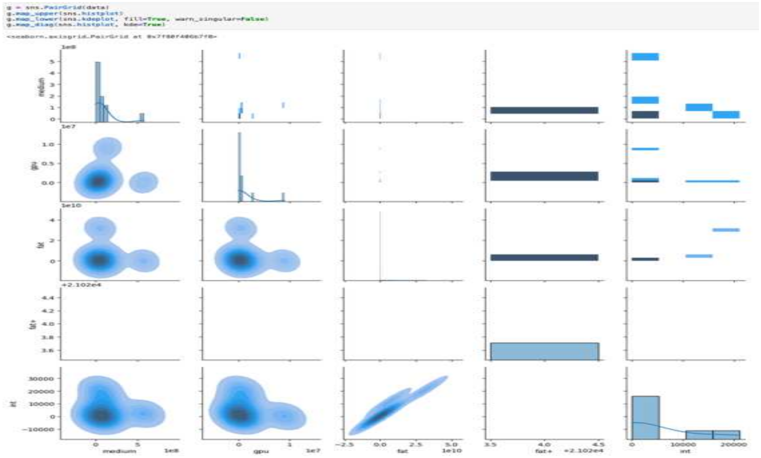


```
sns.pairplot(a)
<seaborn.axisgrid.PairGrid at 0x7fe4e4a5c640>
```

Performance Analysis (Forecasting within partitions)



Performance Analysis (Visualizing GPU & CPU Stop-Loss limits)



```
g = sns.PairGrid(data)
g.map_upper(sns.histplot)
g.map_lower(sns.kdeplot, fill=True, warn_singular=False)
g.map_diag(sns.histplot, kde=True)
```

```
<seaborn.axisgrid.PairGrid at 0x7f80f4806b7f0>
```

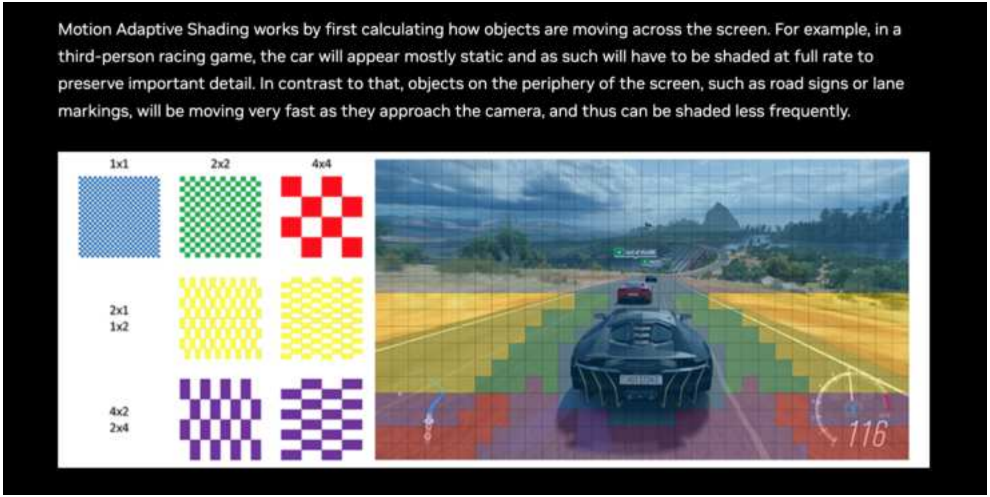
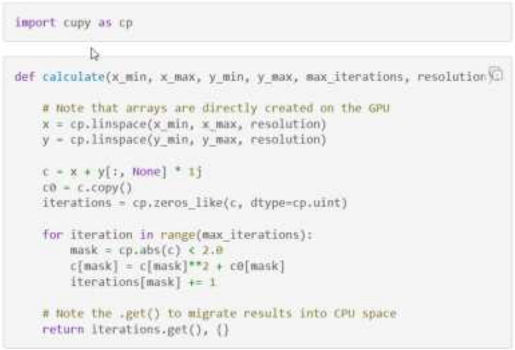

Motion Adaptive Shading works by first calculating how objects are moving across the screen. For example, in a third-person racing game, the car will appear mostly static and as such will have to be shaded at full rate to preserve important detail. In contrast to that, objects on the periphery of the screen, such as road signs or lane markings, will be moving very fast as they approach the camera, and thus can be shaded less frequently.

Image source : https://www.nvidia.com//geforce/news/geforce-rtx-graphics-reinvented/

Benchmarking Naive, Cupy, Numba-cuda, Torch-cuda



```python
import cupy as cp
```

```python
def calculate(x_min, x_max, y_min, y_max, max_iterations, resolution

    # Note that arrays are directly created on the GPU
    x = cp.linspace(x_min, x_max, resolution)
    y = cp.linspace(y_min, y_max, resolution)

    c = x + y[:, None] * 1j
    c0 = c.copy()
    iterations = cp.zeros_like(c, dtype=cp.uint)

    for iteration in range(max_iterations):
        mask = cp.abs(c) < 2.0
        c[mask] = c[mask]**2 + c0[mask]
        iterations[mask] += 1

    # Note the .get() to migrate results into CPU space
    return iterations.get(), {}
```

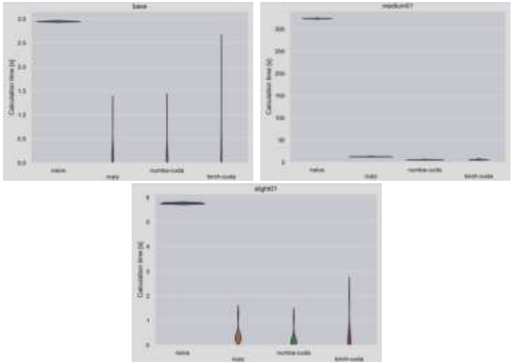
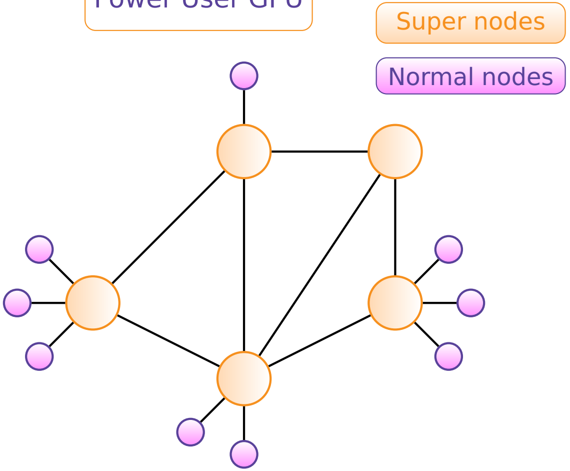
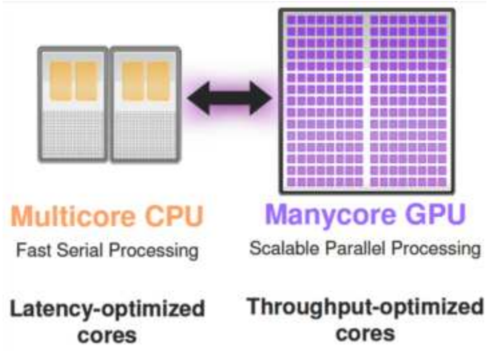


Image source:
https://jneines.github.io/accelerating_python_code/

Power User GPU





Conclusion

- As the GPU uses thousands of lightweight cores whose instruction sets are optimized for dimensional matrix arithmetic and floating point calculations, it is **extremely fast with linear algebra** and similar tasks that require a high degree of parallelism.
- As a rule of thumb,
  - if your algorithm accepts vectorized data, the job is probably well-suited for GPU computing (Nolte et al., "A Secure Workflow for Shared HPC Systems").
- GPU Limitations
  - Less Powerful Cores,
  - Less Memory,
  - Limited APIs The most popular GPU APIs are OpenCL and CUDA. **Just-in-time**, type-specializing, function compiler for accelerating numerically-focused Python for either a CPU or GPU

Thank You Very Much For Your Attention