Python with NumPy on GPU

Pawel Pomorski ppomorsk@sharcnet.ca

January 29, 2025

Introduction

Python easy to use but not built for performance, slower than compiled languages.

NumPy provides performance by calling compiled libraries for numerical computations.

GPUs offer a great deal of computation power, so it's worth considering transferring your NumPy code to the GPU.

GPU availability

New alliance systems will come online in 2025.

system	GPUs
fir	640 H100
nibi	288 H100
rorqual	324 H100
trillium	240 H100
narval	636 A100

These GPUs will be in high demand.

Methods to carry over NumPy computations to the GPU

Aim is easy replacement

CuPy - drop in replacement but need memory management

https://docs.cupy.dev/en

cuPyNumeric - complete replacement, no code modification needed

https://docs.nvidia.com/cupynumeric/

Brief overview of GPUs

No GPU code will need to be written, but still need to know the basics.

Large number of cores of GPUs, so problem must be big enough to occupy them all.

GPU has a separate memory.

Prepare NumPy code for conversion to GPU

Optimize the NumPy code first, always a good idea

Must eliminate explicit loops are replace them with NumPy operations to enable porting to the gpu.

Profile the code to identify regions where the most time is spent, port those to the GPU to speed up the code.

Example: Poisson equation

Solve for u(x) where, for some given b(x)

$$\frac{\partial^2 u}{\partial x^2} = b(x)$$

Finite differencing scheme from discretizing solution on a grid

Pick units so that grid spacing is 1

Boundary conditions u=0 at edges

$u_{i+1} + u_{i-1} - 2u_i = b_i$

Now can express this as

$Au = b$

and solve for u.

Constructing the matrix with explicit loops - avoid!

$$\begin{bmatrix} -2 & 1 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 1 & -2 \end{bmatrix}$$

```python
a=np.zeros( (n,n) )
for i in range(n):
    a[i,i]=-2.0
for i in range(n-1):
    a[i,i+1]=1.0
for i in range(1,n):
    a[i,i-1]=1.0
```

Use NumPy array slicing

$$\begin{bmatrix} -2 & 1 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 1 & -2 \end{bmatrix}$$

	-2	1	0	0	1	-2	1	0	0	1	-2	1	0	0	1	-2	
	-2	0	0	0	0	-2	0	0	0	0	-2	0	0	0	0	-2	
		1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1
1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1		

```
n=4; n2=n*n
i1d=np.identity(n).reshape([n2,])
a=-2.0*i1d
a[1:n2-1]=a[1:n2-1]+i1d[:n2-2]+i1d[2:]
a=a.reshape([n,n])
```

NumPy code

```
import numpy as np
from time import time

n=16000; n2=n*n
i1d=np.identity(n).reshape([n2,])
a=-2.0*i1d
a[1:n2-1]=a[1:n2-1]+i1d[:n2-2]+i1d[2:]
a=a.reshape([n,n])

x=np.arange(n)-n/2.0
w=n/20.0
b=np.exp(-x*x/(w*w))

t0 = time()
x = np.linalg.solve(a,b)
t1 = time()
print("time(s) = ",(t1-t0))
```

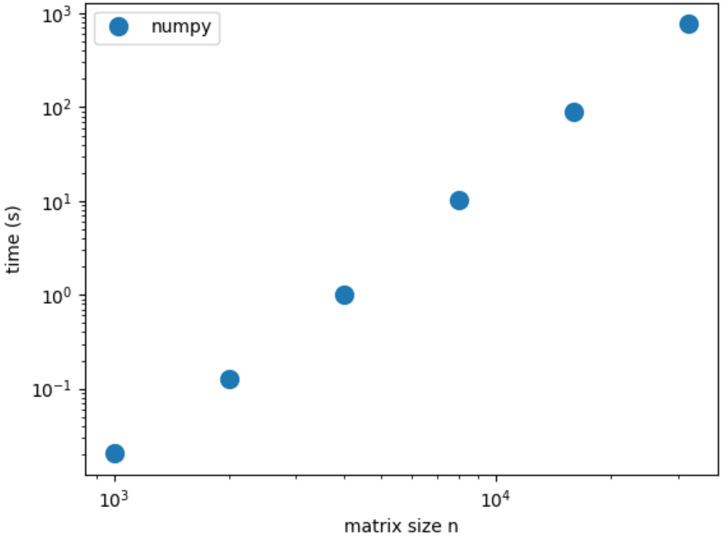
NumPy performance

CPU on Narval GPU node

AMD EPYC 7413 (Zen 3)

n	time	relative change
1000	0.0203	
2000	0.127	6.3
4000	1.01	7.9
8000	10.2	10.1
16000	89.0	8.7
32000	769	8.6

NumPy Performance



NumPy multithreaded performance

NumPy offers automatic multithreading of some routines

Run in a job requesting multiple cpu cores

Example:

```
OMP_NUM_THREADS=4 OMP_PROC_BIND=TRUE python test.py
```

Results for n=16000, narval CPU node AMD EPYC 7532 (Zen 2)

threads	time(s)	speedup
1	70.39	1.0
2	36.0	1.95
4	18.8	3.74
8	10.2	6.85
16	6.77	**10.4**
32	5.04	**13.9**
64	4.82	**14.6**

Basics of CuPy

Install cupy and numpy in virtual environment.

Get a GPU via sbatch or salloc

import cupy as cp

replace np. with cp. where calculation moved to GPU

Define arrays on the GPU

Move data between GPU and CPU.

For efficient programs, the time spend moving data must be significantly less than the time spent computing.

Memory management

```
# create Numpy array on CPU
x_cpu = np.array([1, 2, 3])

# create CuPy array on GPU
x_gpu = cp.array([4, 5, 6])

# move data from CPU to GPU
x_gpu=cp.asarray(x_cpu)

# move data from GPU to CPU
x_cpu = cp.asnumpy(x_gpu)

# NumPy on CPU data only
l2_cpu = np.linalg.norm(x_cpu)

# CuPy funtions operate on GPU data only
l2_gpu = cp.linalg.norm(x_gpu)
```

Example CuPy code

```
import numpy as np
import cupy as cp
from time import time
from cupyx.profiler import benchmark

n=16000
n2=n*n
i1d=np.identity(n).reshape([n2,])

a=-2.0*i1d
a[1:n2-1]=a[1:n2-1]+i1d[:n2-2]+i1d[2:]
a=a.reshape([n,n])

x=np.arange(n)-n/2.0
w=n/20.0
b=np.exp(-x*x/(w*w))
```

```
with cp.cuda.Device(0):
    a_gpu=cp.asarray(a)
    b_gpu=cp.asarray(b)

print( \
benchmark(cp.linalg.solve,(a_gpu,b_gpu),n_repeat=10))

t0 = time()
x_gpu = cp.linalg.solve(a_gpu,b_gpu)
t1 = time()
print("time(s) is ",(t1-t0))

x = cp.asnumpy(x_gpu)
```

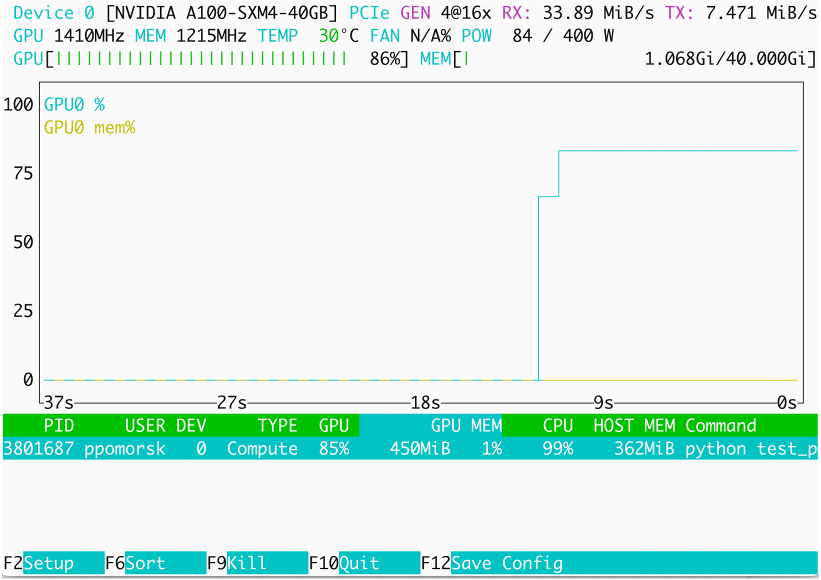
Output

inside job script or interactive job launched via salloc

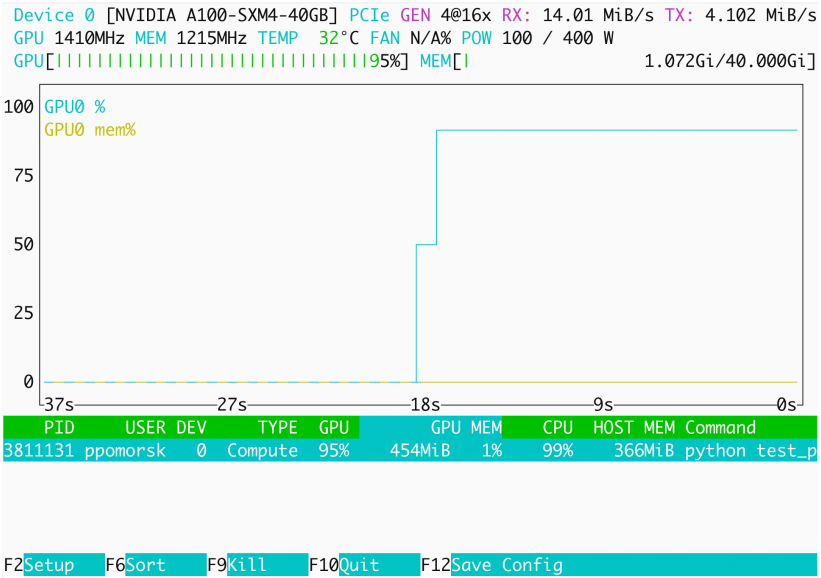
job has to request a gpu

```
$ source ~/ENV/bin/activate
$ module load cuda

$ python cupy_test.py

n =  1000, num_repeats=1000

solve:
CPU: 227.411 us +/-24.012(min:217.421/max:571.790)us
GPU-0: 3183.262 us +/-184.106(min:3122.176/max:4040.704)us
time(s) is  0.003096205711364746
```

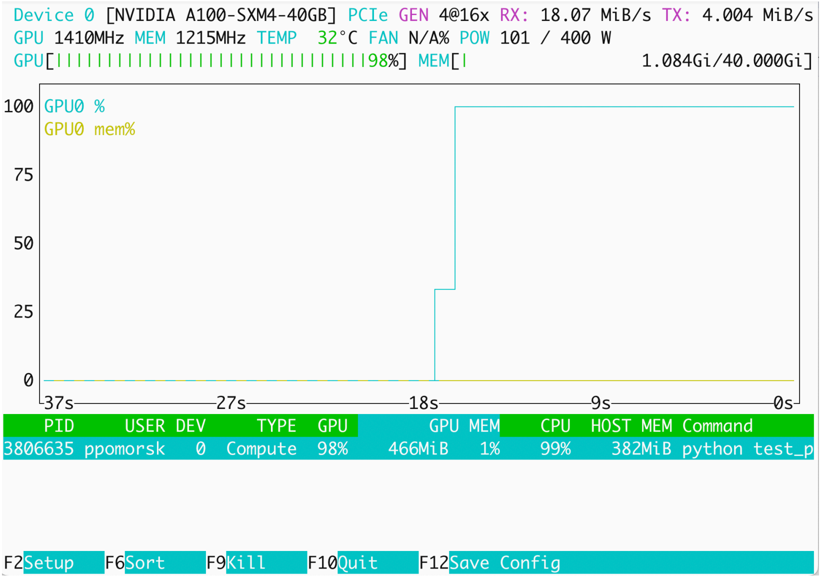
CuPy code GPU efficiency n=200 nvtop output



CuPy code GPU efficiency n=500 nvtop output



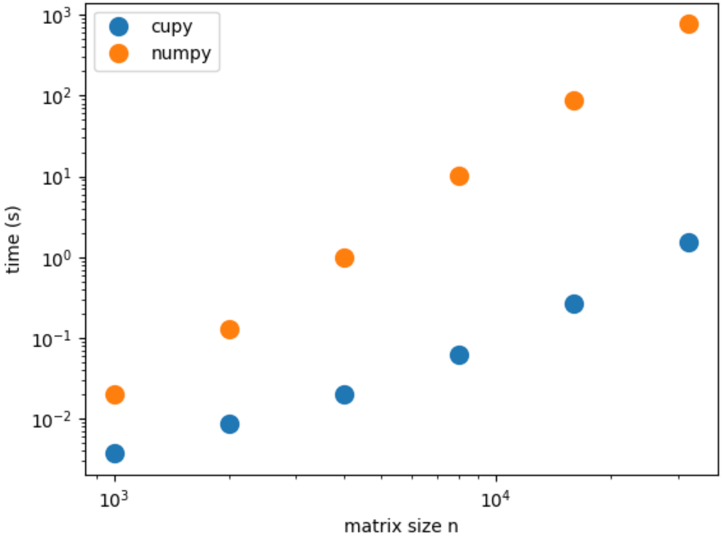
CuPy code GPU efficiency n=1000 nvtop output



Compare to cpu performance

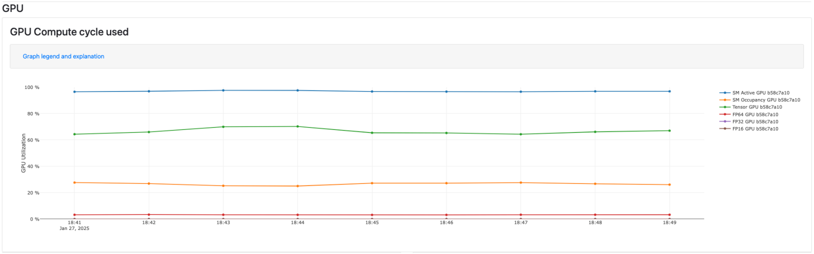
n	NumPy(s)	CuPy(s)	GPU speedup
1000	0.0203	0.00369	**5**
2000	0.127	0.00878	**14**
4000	1.01	0.0203	50
8000	10.2	0.0626	163
16000	89.0	0.265	335
32000	769	1.54	498

Scaling



Tensor cores use

View job data at https://portail.narval.calculquebec.ca/



cuPyNumeric

cuPyNumeric ports all of NumPy code in your program to the GPU.

That means all of your NumPy code must be efficient and cannot use loops.

Loops will slow down performance of cuPyNumeric drastically.

In contrast, CuPy allows to decide which parts of your calculation should run on the GPU and which should not, at the cost of having to manage the memory transfers.

cuPyNumeric code

```
import cupynumeric as np
from legate.timing import time

n=16000; n2=n*n
i1d=np.identity(n).reshape([n2,])
a=-2.0*i1d
a[1:n2-1]=a[1:n2-1]+i1d[:n2-2]+i1d[2:]
a=a.reshape([n,n])

x=np.arange(n)-n/2.0
w=n/20.0
b=np.exp(-x*x/(w*w))

t0 = time()
x = np.linalg.solve(a,b)
t1 = time()
print("time(s) = ",(t1-t0)/1000000.0)
```

Conda through Apptainer

cuPyNumeric requires conda for installation

We do not allow conda installations on our systems. See Anaconda page in our wiki for full justification.

Conda can be used through an Apptainer image

Apptainer is an HPC friendly analogue of Docker

Users can build their own Apptainer images on our clusters

image.def file

```
Bootstrap: docker
From: continuumio/miniconda3:latest

%post
    conda --version
    python --version
    CONDA_OVERRIDE_CUDA="12.2" \
    conda install -c conda-forge -c legate legate
    CONDA_OVERRIDE_CUDA="12.2" \
    conda install -c conda-forge -c legate cupynumeric
    conda install -c conda-forge matplotlib
```

Apptainer image build

```
# load module
module load apptainer

# build image
apptainer build --nv image.sif image.def

# verify legate configuration inside image
apptainer run --nv image.sif legate --info

# run shell inside image
  apptainer shell --nv image.sif
```

Legate

NVIDIA's framework for distributed accelerated computing.

Its goal is to take serial code and execute it without any changes on multple CPUs and GPUs.

Here only showing results for running on a single GPU.

Multi-GPU usage is only available when compiled with cusolverMP.
Only available when Legate built from source.

Running cuPyNumeric with Legate

```
# interactive salloc job
module load apptainer
apptainer shell --nv image.sif
export LEGATE_SHOW_CONFIG=1
legate --ompthreads=1 python cupynumeric_test.py

#job script
module load apptainer
apptainer run --nv image.sif \
legate --ompthreads=1 python test.py
```

Performance

n	cuPyNumeric(s)	CuPy(s)
1000	0.00392	0.00369
2000	0.00865	0.00878
4000	0.0205	0.0203
8000	0.0624	0.0626
16000	0.257	0.265
32000	OOM error	1.54

Conclusions

GPUs provide can provide huge performance boosts to your NumPy programs.

CuPy and cuPyNumeric offer two easy ways to convert your code to the GPU.

Problem size must be large enough before GPU use worthwhile.