

GPU Programming Made Easy with CuPy

Bernardo Abreu Figueiredo
Konstantinos Iliakis

Acknowledgements: Simon Albright, Helga Timko, Heiko Damerau

Outline

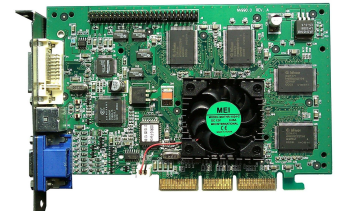
- Why GPUs? Why CuPy?
- Two real-world use cases
- Live Demo
- CuPy features and capabilities
- Speedup results for the presented use cases
- How to access CERN resources

Why GPUs?

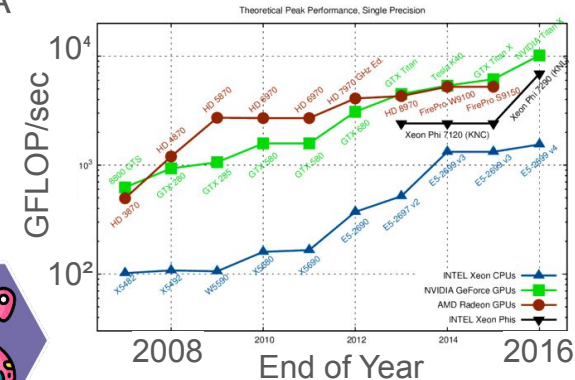
- Originally (80s-90s) built for graphics, called Video Graphics Arrays/ Adapters (**VGAs**)
- In 2007, Nvidia introduces **CUDA** to facilitate general-purpose application development
- Combination of computing-capacity and cost-efficiency → **dominant platform** for general-purpose acceleration
- Nowadays: **Widespread applicability** in every computing domain



2007: Initial CUDA release



1999: World's first GPU
GeForce 256



2008-2016: CPU- GPU,
peak performance
comparison [1]

Present: Widespread
Adoption of GPUs

[1] <https://www.karlsruhp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/>

GPU Challenges

1. GPUs are throughput oriented devices:

- GPUs implement SIMD: Single operation on multiple data points simultaneously
- Massive multi-threading and widely vectorized execution units

2. Cumbersome programming model:

- Implicit parallelism: Every code line executed by multiple threads
- Limited debugging tools

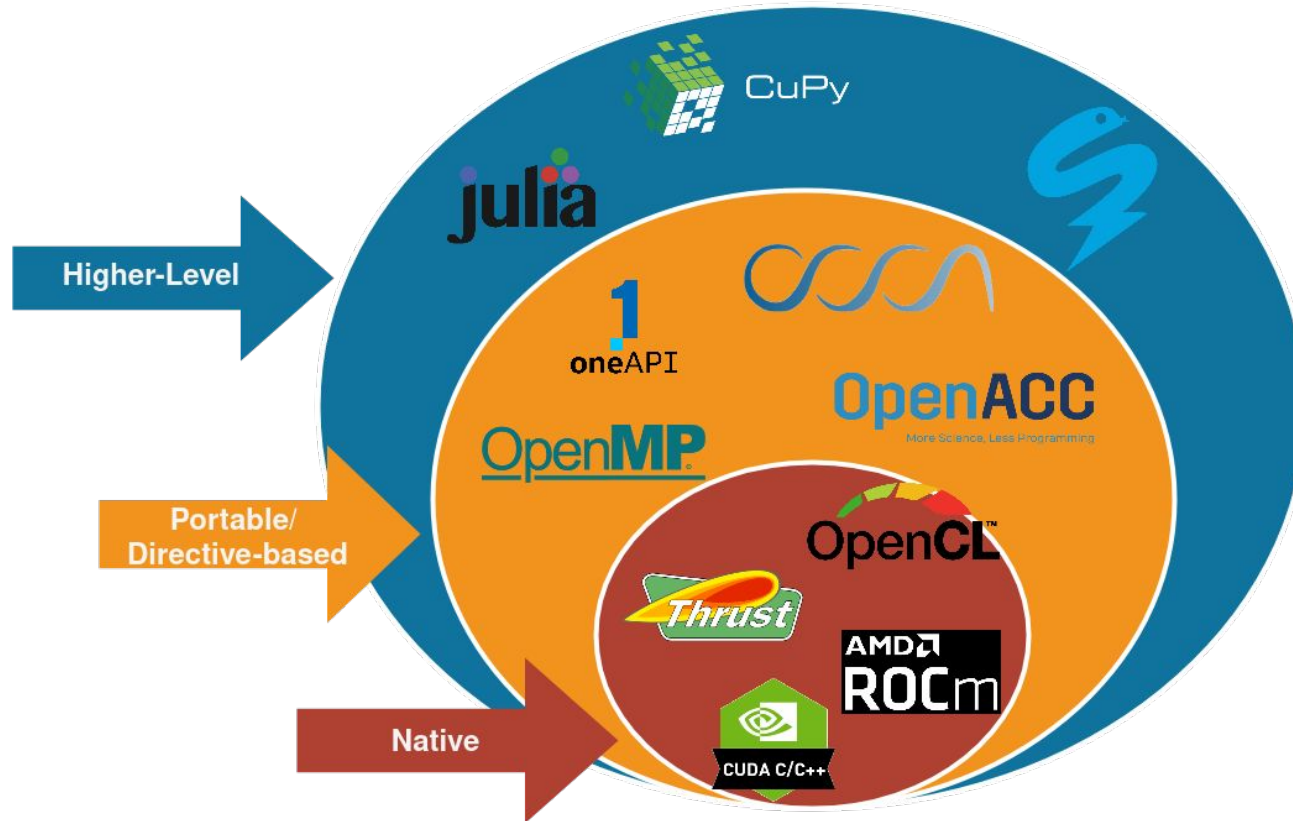
3. Performance bottlenecks: Can negate potential performance gains

- Data transfers
- Memory management
- Thread divergence



CuPy

Zoo of GPU Programming Solutions



Why CuPy?

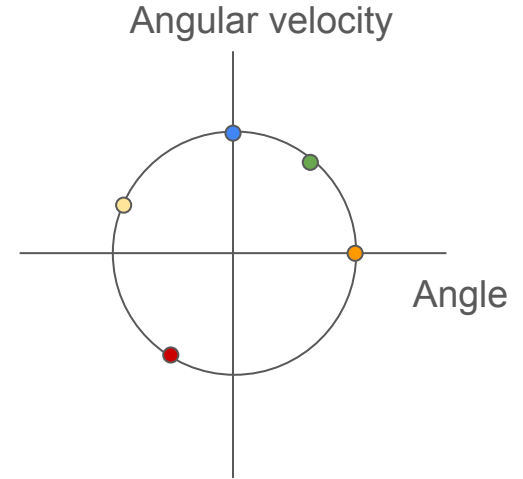
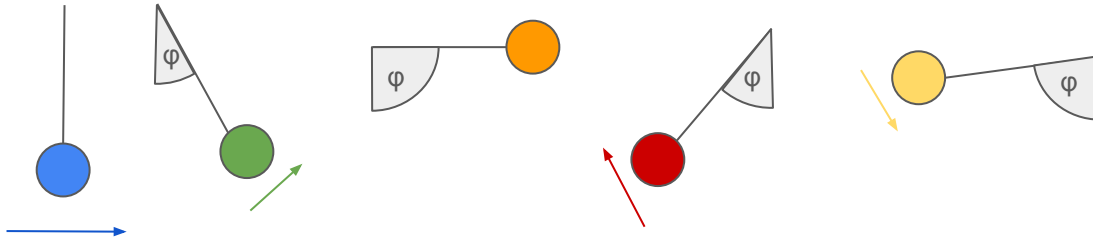


- Beginner friendly:
 - Requires **minimal knowledge of GPU** programming model and architecture
 - Easy-to-install (pip, conda)
- Flexibility and applicability:
 - Drop-in replacement for NumPy & SciPy (equivalent API)
 - Complete list:
<https://docs.cupy.dev/en/stable/reference/comparison.html>
 - Multiple ways to implement GPU kernels
 - NVIDIA + AMD platforms
- Efficiency:
 - Most modern features, optimized libraries
 - Extremely low-overhead
 - Low-level support

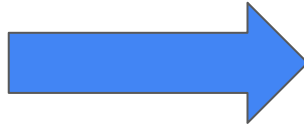
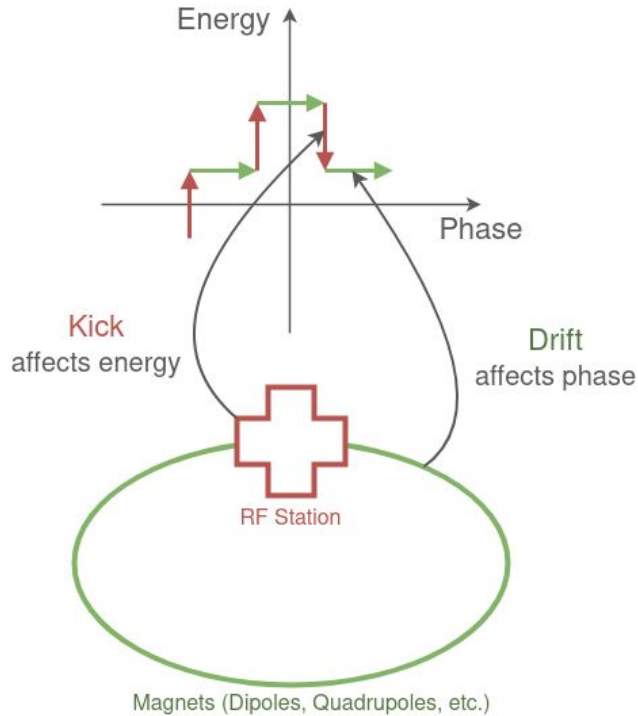


Use Case: Synchrotron Motion

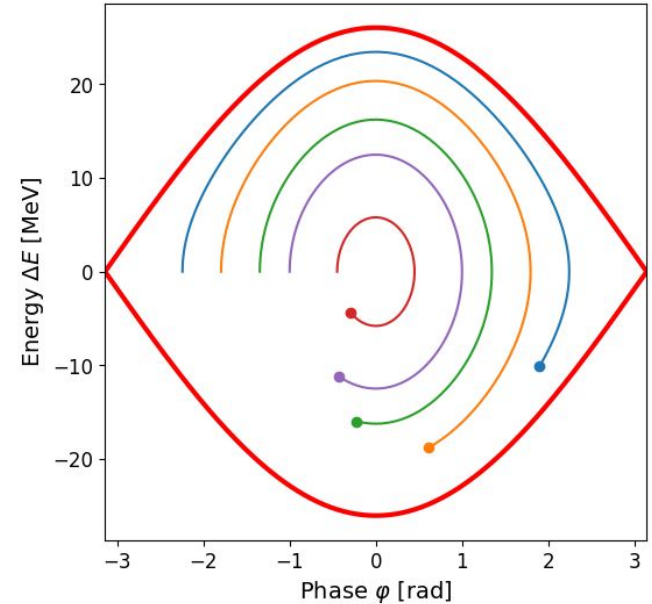
- Phase space describes state of a physical system
- Analogous to pendulum motion
 - Described by angle and angular velocity (change in angle)



Use Case: Synchrotron Motion

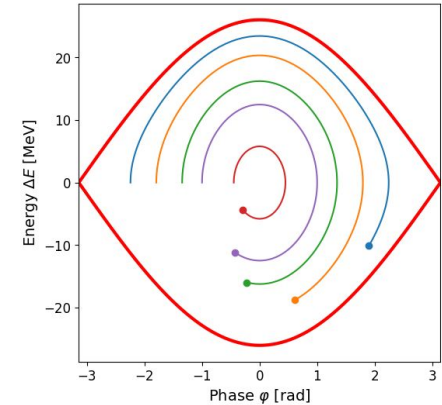


Tracking particles



Use Case: Synchrotron Motion

- Distribution of macroparticles in phase space
 - Given in phase and energy (or equivalent) coordinates
- Can be described by alternating kick and drift
 - Kick affects energy coordinates (Particle traversing RF station)
 - Drift affects phase coordinates (Trajectory bent by magnetic field)
- Calculation does not depend on other particles
 - Highly parallelizable



Particle	Phase	Energy
1	-1	0
2	0.2	0.8

Kick

Particle	Phase	Energy
1	-1	0+0.05
2	0.2	0.8-0.02

Drift

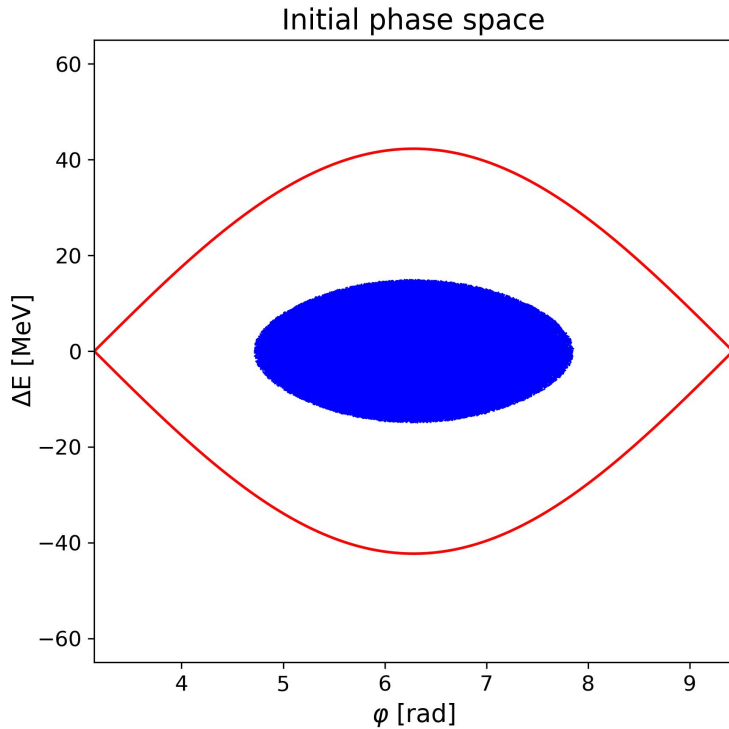
Particle	Phase	Energy
1	-1+0.04	0+0.05
2	0.2-0.06	0.8-0.02

Use Case: Beam Longitudinal Dynamics (BLonD) Code

- **Particle tracking simulator**, specializes on the longitudinal plane (δT , δE)
- **Modular structure**, can simulate a wide range of conditions
 - Energy regimes (MeV to TeV)
 - Particle types (electron, proton, muon, ...)
 - Actively used for PSB, PS, SPS, LHC, FCC, Muon Collider, etc
- **Indispensable tool** for:
 - Efficient operation
 - Accelerator upgrades
 - Future projects
- Written in Python, with **accelerated backends** (C++, Numba, CuPy, MPI)
- Well documented and benchmarked, recently **PRAB Editor's Suggestion** [1]

[1] H. Timko et al. "Beam longitudinal dynamics simulation studies", <https://journals.aps.org/prab/abstract/10.1103/PhysRevAccelBeams.26.114602>

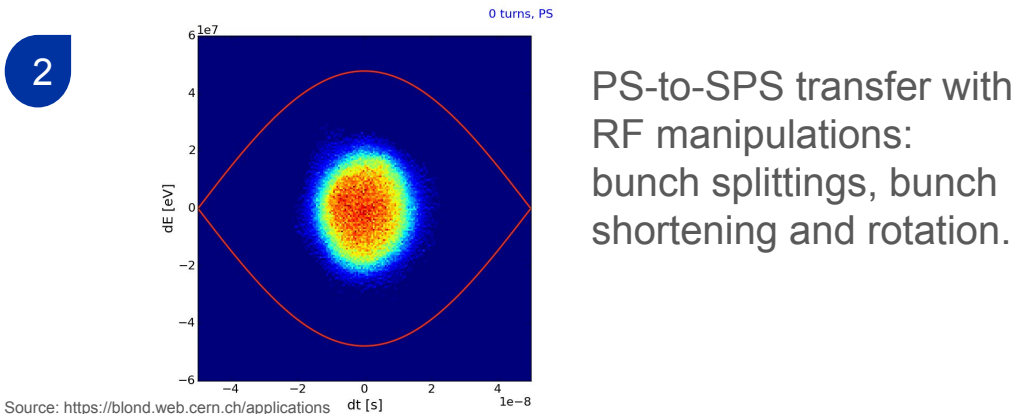
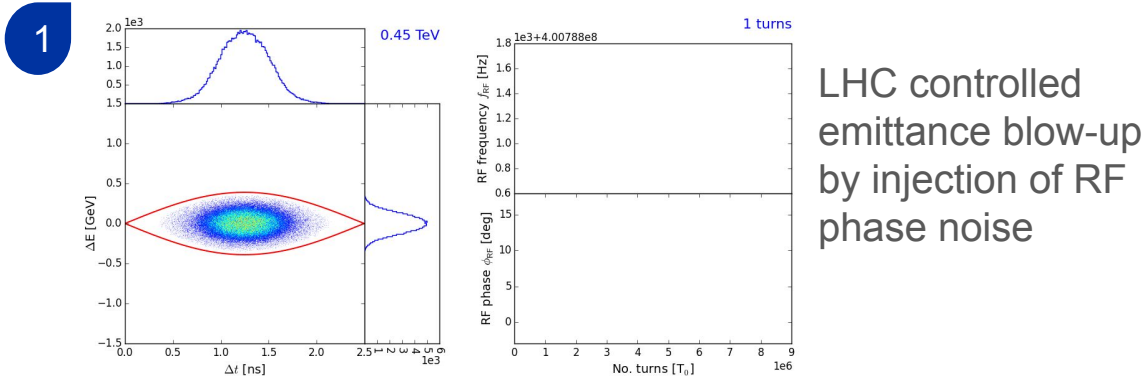
Use Case: BLoND Applications (simple)



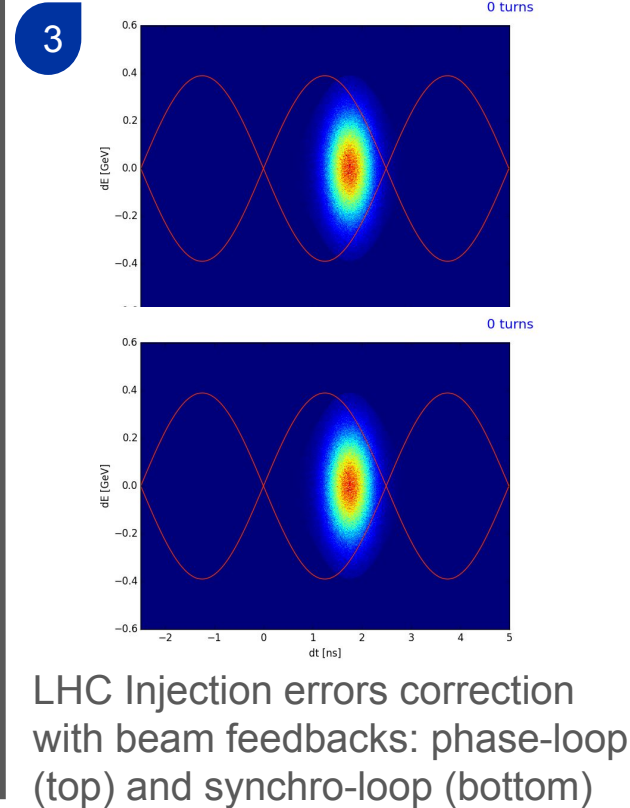
Example simulation of a bunch undergoing oscillations at injection

Here, the tracking can be completely parallelized

Use Case: BLonD Applications (more complicated)

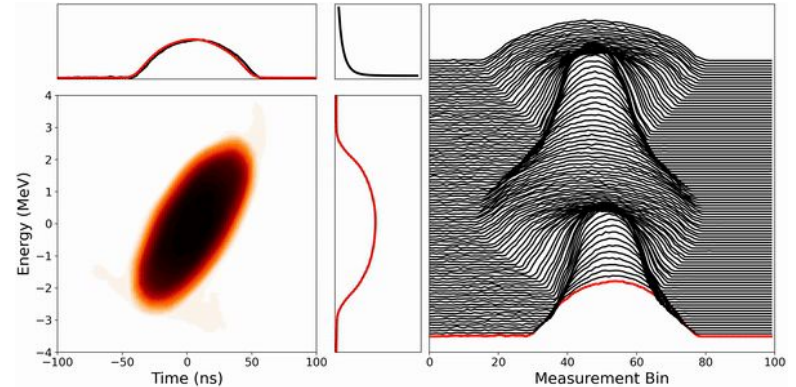


Source: <https://blond.web.cern.ch/applications>



Use Case: Longitudinal Phase Space Tomography

- Goal: Reconstruct the distribution of a particle bunch in longitudinal phase space
- Analogous to medical Tomography
 - Breathing patient [2]
 - Bunch rotating in phase space
- Input:
 - Accelerator and beam parameters
 - Measured (or generated) 1D bunch profiles
- Output:
 - Reconstructed 2D phase space distribution



Rotating bunch in phase space in the PSB [1]

[1] <https://tomograp.web.cern.ch>

[2] A. Biguri et al. "A General Method for Motion Compensation in X-ray Computed Tomography", <https://iopscience.iop.org/article/10.1088/1361-6560/aa7675>

Use Case: Longitudinal Tomography

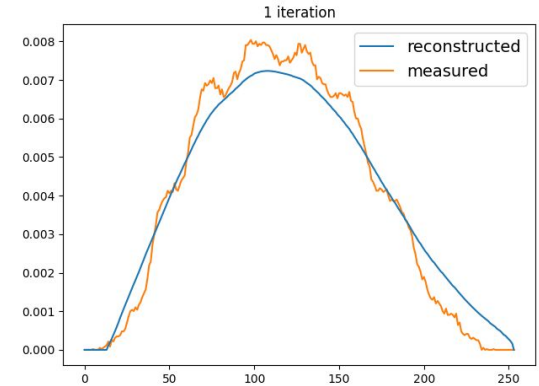
Two main parts of the application

1. Tracking

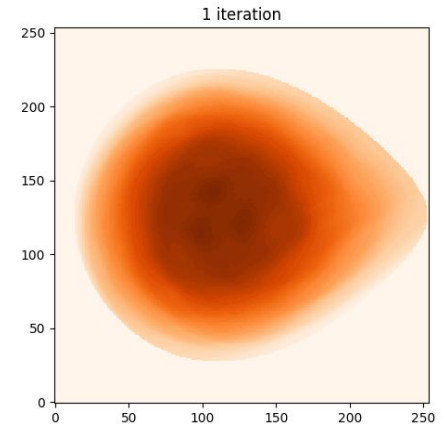
- Generate a **distribution** of particles
- Track **particles** for a number of turns (based on applying equations of motion)
- Store the phase/time and energy **coordinates** of the particles
- Massively parallelizable

2. Tomography reconstruction

- Initialize **weights** for particles based on their coordinates
- Reconstruct a **profile** based on initial weights
- Iteratively **adjust** both **weights** and **reconstructed profile** until convergence (based on difference)
- Partly parallelizable



Measured and reconstructed profile



Iterative reconstruction

How can BLoND and Tomography profit from GPUs?

- **Computationally intensive**
 - Tracking: trigonometric, exponential, etc
 - FFTs: Forward and backward FFTs
 - Linear algebra: Array and vector operations
- **Data parallel, mostly dependency-free**
- **Large input sets**
 - Number of simulated particles: 1 Million - 1 Billion
- **Infrequent need for CPU-GPU memory transfers**
 - Apart from periodic need for plotting/data storage, all other operations are GPU-accelerated

Interactive Session

- <https://gitlab.cern.ch/beams-and-rf-training/icsc-2024-cupy>
- First steps with CuPy
 - Creating CuPy arrays
 - Timing basic CuPy array operations
- When considering doing work on GPU, keep four things in mind
 - **Input size:** Large enough to keep GPU cores busy?
 - **Arithmetic Intensity:** Is the computation heavy enough?
 - **Data type length:** Is highest precision necessary or can it be reduced to achieve a better performance?
 - **Memory transfer:** Do we have to copy lots of data? Is there a way to keep them on one device to minimize transfers?
- There is no one-size-fits-all solution
 - Profile your code to see which device performs the best

CuPy Features: Supported functions

Complete list here: <https://docs.cupy.dev/en/stable/reference/comparison.html>

- Includes NumPy and SciPy routines
- CuPy behaves like a drop-in replacement for NumPy/SciPy
- NumPy and CuPy can be used interchangeably

```
import numpy as np
import cupy as cp

for xp in [np, cp]:
    x = xp.arange(10)
    W = xp.ones((10, 5))
    y = xp.dot(x, W)
    print(y)
```

CuPy Features: Drop-in replacement for NumPy

CuPy Arrays - Almost identical interface with NumPy arrays

```
import numpy as np
import cupy as cp

# Supports all array creation routines, like zeros, ones, empty, etc
dev_a = cp.arange(10, dtype=int)
dev_b = cp.array([1, 2, 3, 4])
print(type(dev_a)) ## Output: <class 'cupy.ndarray'>

# Can be printed out of the box, though this results in device to host memory copying
%time print(dev_a) ## Output: [...] Wall time: 2 ms

a = np.arange(10, dtype=int)
%time print(a) ## Output: [...] Wall time: 223 μs
```

CuPy Features: Drop-in replacement for NumPy

CuPy also supports all sorts of indexing

```
# strided with start stop index
print(dev_a[1:-1:2])
# using list of indices to gather
print(dev_a[[0,2,4]])
# or with boolean list
print(dev_a[dev_a % 3 == 0])
```

CuPy interoperable with NumPy arrays

```
# To get an array back to the host is simple:
b = cp.asnumpy(dev_a)
c = dev_a.get()
print(type(b), type(c))

# Cupy can (in exceptions) operate solely on numpy arrays
print(cp.allclose(b, c))
```

CuPy allows to write CPU/GPU agnostic code

```
# Easy to transfer arrays between device and the host
a = np.arange(0, 20, 2)
dev_a = cp.asarray(a)

# GPU/CPU agnostic code also works with CuPy
xp = cp.get_array_module(dev_a) # Returns cupy if any array is on the GPU, otherwise numpy
y = xp.sin(dev_a) + xp.cos(dev_a)
```

Flexibility for expressing and launching GPU kernels

1. Supported Numpy functions:

```
# Just a random compute intensive function
def saxpy_trig(x, y, a):
    return cp.exp(a * cp.sin(x) + cp.cos(y))

res = saxpy_trig(dev_x, dev_y, 0.5)
```

Automatic number of threads definition

Bonus: Fuse operations in a single kernel

```
@cp.fuse(kernel_name='saxpy_trig_fused')
def saxpy_trig_fused(x, y, a):
    return cp.exp(a * cp.sin(x) + cp.cos(y))

res = saxpy_trig_fused(dev_x, dev_y, 0.5)
```

Ease of use



Expressivity



Performance



2. Templated kernels for element-wise operations and reductions.

```
saxpy_trig_elemwise = cp.ElementwiseKernel(
    'float32 x, float32 y, float32 a', # Input Types
    'float32 z', # Output Types
    'z = exp(a * sin(x) + cos(y))', # Operation
    'saxpy_trig_elemwise' # Kernel name
)

res = saxpy_trig_elemwise(dev_x, dev_y, 0.5)
```

Automatic number of threads definition

Ease of use



Expressivity



Performance



3. With “raw” CUDA code

```
saxpy_trig_raw = cp.RawKernel(r'''
#include <cupy/complex.cuh>
extern "C" __global__
void saxpy_trig_raw(const float* x, const float* y,
                    float a, float*z, int n)
{
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid < n)
        z[tid] = exp(a * sin(x[tid]) + cos(y[tid]));
}
''', 'saxpy_trig_raw')

res = saxpy_trig_raw(args=(dev_x, dev_y, 0.5,
                           dev_out, len(dev_x)),
                    grid=((len(dev_x)+1023)//1024, ),
                    block=(1024, ))
```

- Manual number of threads definition
- Also supports loading pre-compiled kernels

Ease of use



Expressivity



Performance



CuPy Features: Access CUDA API

Exploring the available device and its attributes

```
import cupy as cp
device = cp.cuda.Device()
device.use()

print('Using device: ', cp.cuda.runtime.getDeviceProperties(device)['name'])
## Output: Using device: b'Tesla T4'

attributes = device.attributes
properties = cp.cuda.runtime.getDeviceProperties(device)
print('Number of multiprocessors:', attributes['MultiProcessorCount'])
## Output: Number of multiprocessors: 40

print('Global memory size (GB):', properties['totalGlobalMem'] / (1024**3))
## Output: Global memory size (GB): 14.58062744140625
```

CuPy Features: Access CUDA API to time functions

CUDA events to time GPU kernels

```
# It is trickier to time GPU kernels, because they behave asynchronously w.r.t the host
def benchmark(func, args, n_repeat=10, n_warmup=1):
    import cupy as cp
    gpu_start = cp.cuda.Event()
    gpu_end = cp.cuda.Event()
    for i in range(n_warmup):
        out = func(*args)

    gpu_start.record()
    for i in range(n_repeat):
        out = func(*args)


    gpu_end.record()
    gpu_end.synchronize()
    t_gpu = cp.cuda.get_elapsed_time(gpu_start, gpu_end)
    print('Average GPU time (ms): ', t_gpu / n_repeat)
```

CuPy Advanced Features: Streams

- Concurrency through pipelining
- Overlap memory transfers with kernel executions

Sequential: 

Time →

Concurrent: 

Time →

H2D: Host to Device, D2H: Device to host

CuPy Advanced Features: Streams

```
import cupy as cp
import numpy as np

rand = cp.random.RandomState(seed=1)
streams = []

for i in range(10):
    streams.append(cp.cuda.Stream()) # Create the streams

y_cpu = np.random.normal(size=(2**24, 1)) # Create one random matrix in CPU

for stream in streams: # Iterate over streams and execute operations asynchronously
    with stream:
        x = rand.normal(size=(1, 2**24)) # Create other random matrix on GPU
        y = cp.asarray(y_cpu) # Transfer CPU matrix to GPU
        z = cp.matmul(x, y) # Multiply matrices

for stream in streams:
    stream.synchronize()
```

Overlapping execution!



CuPy Advanced Features: Memory Pool

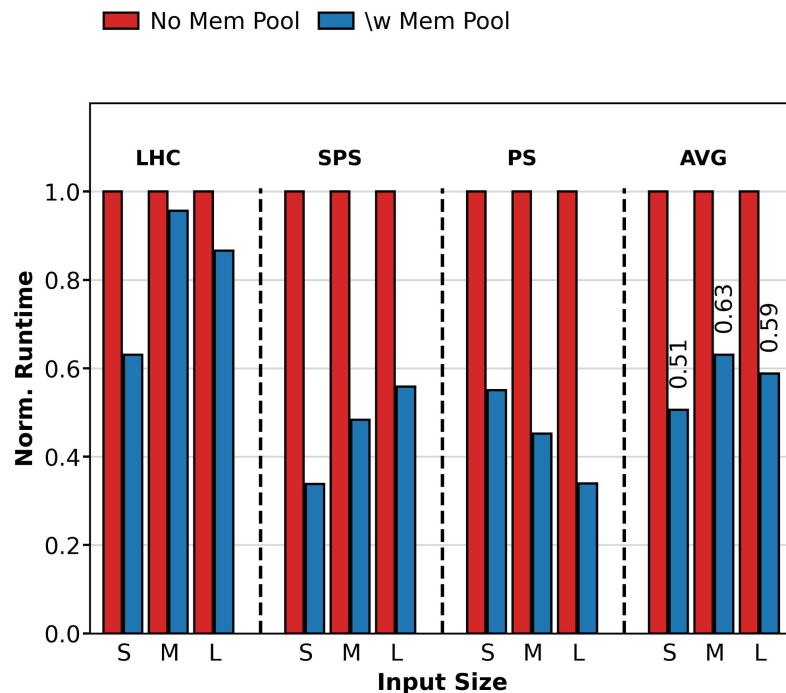
- Memory allocations (on the GPU) can be costly

Memory management

- Myth: Always free() your memory
- Reality: It's a tradeoff – leaking is sometimes cheaper

Introduction to Efficient Computing, A. Nowak, tCSC-22

- Memory pool: Software managed GPU memory region
- Instead of deallocating memory: Keeping it for future use
- Caches allocated memory blocks
- Reduce cost of alloc/free



GPU Models used at CERN

Model	Tesla T4	V100	A100
Generation	Turing (2019)	Volta (2018)	Ampere (2020)
Transistors	13.6 billion	21.1 billion	54.2 billion
RAM	16 GB	32 GB	40 GB
Bandwidth	320 GB/s	900 GB/s	1555 GB/s
Cores	40	80	108
Peak FP32 Perf.	8.1 TFLOPS	15.7 TFLOPS	19.5 TFLOPS
Peak FP64 Perf.	0.25 TFLOPS*	7.8 TFLOPS	9.7 TFLOPS

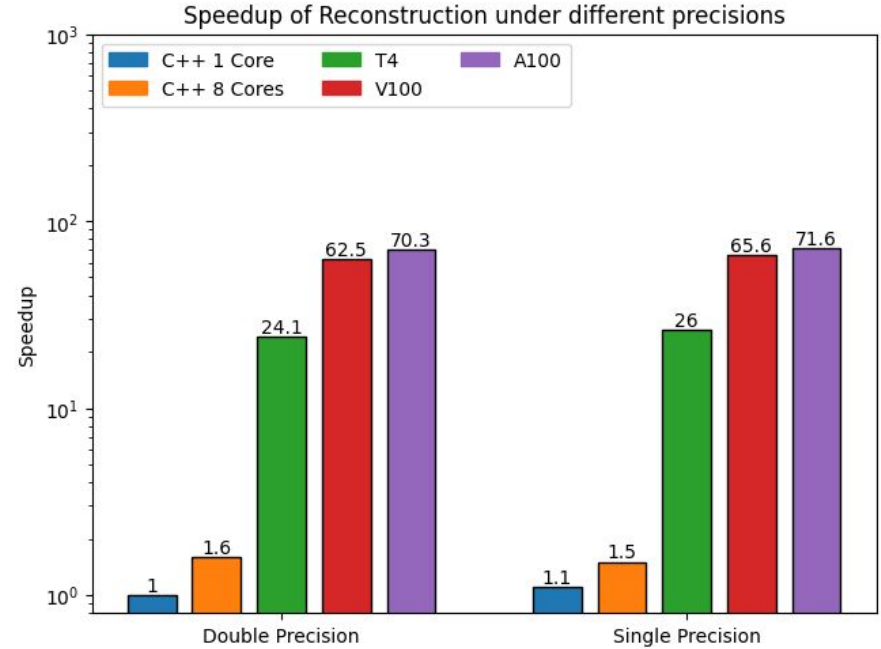
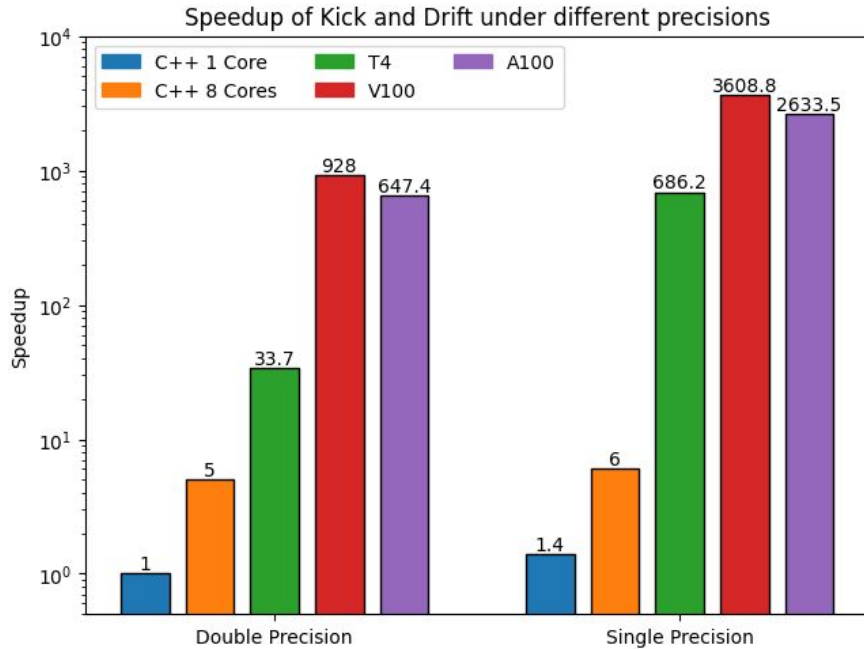
TFLOPS = 10^{12} floating-point operations per second

* Estimated value, not given in documentation

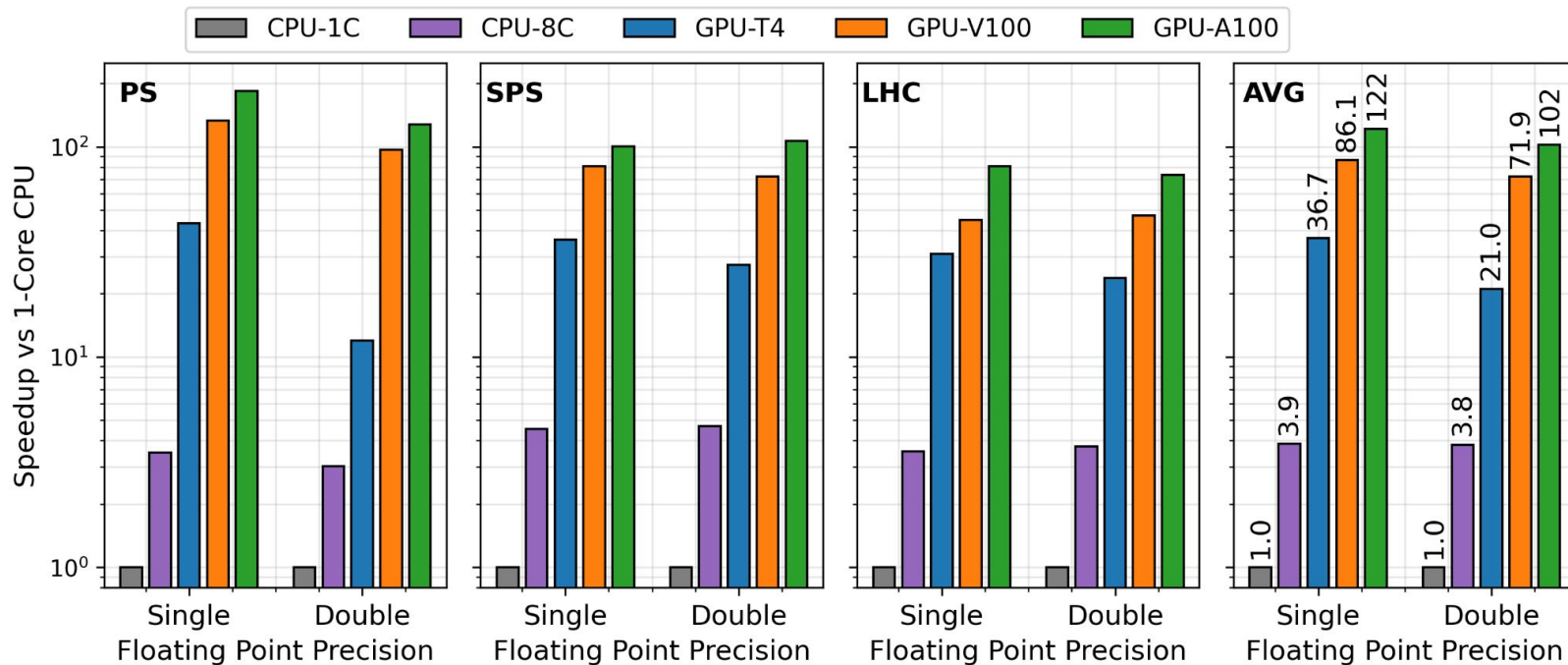
Results of using GPU for Longitudinal Tomography

- Initially: Python and C++/OpenMP
- Now: Python, choice between C++/OpenMP and CuPy with raw CUDA kernels
- Tracking and reconstruction more than 25x faster on GPU
- Impact stronger with 32-bit floats (single precision)
- Side effects: Better performance outside of CUDA kernels (using CuPy functions instead of NumPy functions)

GPU Benchmarks for Longitudinal Tomography



CuPy BLonD Speedup



Over two-orders of magnitude speedup in three real-world test-cases. Baseline: Intel Xeon Silver 4216

Getting Started with CuPy

- Requirements:
 - NVIDIA CUDA GPU
 - CUDA Toolkit v11.2 or higher
 - Python 3.9 or higher
- Easy installation with pip or conda:
 - `conda install -c conda-forge cupy`
 - `pip install cupy`
- More information: <https://docs.cupy.dev/en/stable/install.html>

Accessing Resources Interactively

- Notebooks (GUI):
 - Swan (Need to request access)
 - <https://swan-k8s.cern.ch>
 - Equipped with T4 GPUs
- Scripts & command line interface
 - LXPLUS Service
 - Equipped with T4 GPUs
 - ssh address: [user]@lxplus-gpu.cern.ch

Accessing Resources in Batch mode

- Submit jobs:
`condor_submit -i`
`condor.sub`
- Available GPU models:
 - A100
 - V100
 - T4
- Better for longer runs
- More information at:
<https://batchdocs.web.cern.ch/>

```
#####  
# File: condor.sub  
# HTCondor submit file  
#####  
  
# Define executable script  
executable = condor.sh  
  
# Define output/ error files  
output = output.txt  
error = error.txt  
log = log.txt  
  
# Request 1 GPU  
request_gpus = 1  
  
# Optionally, specify GPU model name  
requirements = regexp("A100", TARGET.GPUs_DeviceName)  
  
+MaxRuntime = 3600  
  
queue
```

```
#!/bin/bash  
  
#####  
# File: condor.sh  
# Simple executable script  
#####  
  
source $USER/.bashrc  
  
python my_script.py
```


Key Takeaways

- GPUs offer massive computing capacity
 - Harvesting it can be tedious
- High-level libraries can simplify GPU development
- CuPy: A good first step to start with GPU programming
 - User-friendliness
 - Flexibility
 - Performance, low-level support
- Impressive real-world speedup
 - BLoND: 20-100x faster
 - Tomography: 6-20x faster
- Easy to get started
 - Plenty of resources at CERN
 - Interactive and batch access