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

[1] https://www.karlrupp.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
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)
  - 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
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

<table>
<thead>
<tr>
<th>Particle</th>
<th>Phase</th>
<th>Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0.2</td>
<td>0.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Particle</th>
<th>Phase</th>
<th>Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1</td>
<td>0.05</td>
</tr>
<tr>
<td>2</td>
<td>0.2</td>
<td>0.8-0.02</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Particle</th>
<th>Phase</th>
<th>Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1+0.04</td>
<td>0+0.05</td>
</tr>
<tr>
<td>2</td>
<td>0.2-0.06</td>
<td>0.8-0.02</td>
</tr>
</tbody>
</table>
Use Case: Beam Longitudinal Dynamics (BLonD) Code

- **Particle tracking simulator**, specializes on the longitudinal plane ($\delta \tau$, $\delta E$)
- **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]

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)

1. LHC controlled emittance blow-up by injection of RF phase noise

2. PS-to-SPS transfer with RF manipulations: bunch splittings, bunch shortening and rotation.

3. LHC Injection errors correction with beam feedbacks: phase-loop (top) and synchro-loop (bottom)

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

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
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

- [link](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


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

```python
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

```python
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

```python
# 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

```python
# 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

```python
# 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)
```
1. Supported Numpy functions:

```python
# 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)
```

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

```python
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)
```

3. With “raw” CUDA code

```python
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,))
```

Automatic number of threads definition

Bonus: Fuse operations in a single kernel

```python
@cp.fuse(kernel_names='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)
```

Automatic number of threads definition

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

Flexibility for expressing and launching GPU kernels

Ease of use  Expressivity  Performance

Ease of use  Expressivity  Performance

Ease of use  Expressivity  Performance
CuPy Features: Access CUDA API

Exploring the available device and its attributes

```python
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

```python
# 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:  
Memcpy (H2D)  Kernel Exec Memcpy (D2H)

Concurrent:  
H2D-1  K1  D2H-1  
H2D-2  K2  D2H-2  
H2D-3  K3  D2H-3

H2D: Host to Device, D2H: Device to host
CuPy Advanced Features: Streams

```python
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

- 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

Introduction to Efficient Computing, A. Nowak, tCSC-22
## GPU Models used at CERN

<table>
<thead>
<tr>
<th>Model</th>
<th>Tesla T4</th>
<th>V100</th>
<th>A100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transistors</td>
<td>13.6 billion</td>
<td>21.1 billion</td>
<td>54.2 billion</td>
</tr>
<tr>
<td>RAM</td>
<td>16 GB</td>
<td>32 GB</td>
<td>40 GB</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>320 GB/s</td>
<td>900 GB/s</td>
<td>1555 GB/s</td>
</tr>
<tr>
<td>Cores</td>
<td>40</td>
<td>80</td>
<td>108</td>
</tr>
<tr>
<td>Peak FP32 Perf.</td>
<td>8.1 TFLOPS</td>
<td>15.7 TFLOPS</td>
<td>19.5 TFLOPS</td>
</tr>
<tr>
<td>Peak FP64 Perf.</td>
<td>0.25 TFLOPS*</td>
<td>7.8 TFLOPS</td>
<td>9.7 TFLOPS</td>
</tr>
</tbody>
</table>

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

Speedup of Kick and Drift under different precisions

<table>
<thead>
<tr>
<th></th>
<th>C++ 1 Core</th>
<th>T4</th>
<th>A100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Single</td>
<td>1.4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Precision</td>
<td>1.4</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Speedup of Reconstruction under different precisions

<table>
<thead>
<tr>
<th></th>
<th>C++ 1 Core</th>
<th>T4</th>
<th>A100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Single</td>
<td>1.4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Precision</td>
<td>1.4</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
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`

Accessing Resources Interactively

- **Notebooks (GUI):**
  - Swan (Need to request access)
    - [https://swan-k8s.cern.ch](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:
  
  ```bash
  condor_submit -i condor.sub
  ```

- Available GPU models:
  - A100
  - V100
  - T4

- Better for longer runs

- More information at:
  [https://batchdocs.web.cern.ch/](https://batchdocs.web.cern.ch/)

---

### Example Condor Submit File `condor.sub`

```bash
# File: condor.sub
# HTCondor submit file

# Define executable script
exec = 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
```

---

### Example Condor Executable Script `condor.sh`

```bash
#!/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