Offload Annotations: Bringing Heterogeneous Computing to Existing Libraries and Workloads

Gina Yuan, Shoumik Palkar, Deepak Narayanan, Matei Zaharia
Stanford University
USENIX ATC 2020 (July 15-17)
Background: Hardware Commoditization
Background: CPUs vs. GPUs

CPUs

- Control
- Core
- Core
- Core
- Cache
- Memory

4-way parallelism
512GB memory

GPUs

- (PCI-E)
- Memory

1000-ways parallelism!
16GB memory

Costly data transfers!
Popular Python data science libraries for the CPU.
Trend: Data Science on the GPU

Lots of parallel data!

NEW Python data science libraries for the GPU.
Trend: CPU Libraries vs. GPU Libraries

https://github.com/rapidsai/cudf

**cuDF provides a pandas-like API** that will be familiar to data scientists, enabling them to easily accelerate their workflows without going into the details of CUDA programming.

https://cupy.chainer.org/

**CuPy's interface is highly compatible with NumPy**; in most cases it can be used as a drop-in replacement. All you need to do is just replace `numpy` with `cupy` in your code. Basics of CuPy (Tutorial) is useful to learn more.

https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html

**WHAT IS PYTORCH?**

A replacement for NumPy to use the power of GPUs

https://github.com/rapidsai/cuml

**cuML enables data scientists, researchers, and software engineers to run traditional tabular ML tasks on GPUs without going into the details of CUDA programming.** In most cases, **cuML's Python API matches the API from scikit-learn.**
Trend: CPU Libraries vs. GPU Libraries

https://github.com/rapidsai/cudf

https://github.com/rapidsai/cuml

Are GPU libraries as straightforward to use as they seem?

https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html

cuDF provides a pandas-like interface that easily accelerate their workflows.

cuML enables data scientists, researchers, and software engineers to run traditional tabular ML tasks on GPUs without digging into the details of CUDA programming. In most cases, cuML's Python API matches the API from scikit-learn.

https://cupy.chainer.org/
Motivating Example

```
# Fit.
m1 = sklearn.StandardScaler()
m2 = sklearn.PCA()
m3 = sklearn.KNeighborsClassifier()
X_train = m1.fit_transform(X_train)
X_train = m2.fit_transform(X_train)
m3.fit(X_train, Y_train)

# Predict.
X_test = m1.transform(X_test)
X_test = m2.transform(X_test)
result = m3.predict(X_test)
plottinglib.plot(result)
```
Motivating Example

```python
# Fit.
m1 = sklearn.StandardScaler()
m2 = cuml.PCA()
m3 = cuml.KNeighborsClassifier()
X_train = m1.fit_transform(X_train)
X_train = m2.fit_transform(X_train)
m3.fit(X_train, Y_train)

# Predict.
X_test = m1.transform(X_test)
X_test = m2.transform(X_test)
result = m3.predict(X_test)
plottinglib.plot(result)
```
Motivating Example

- Missing Functions
- Manual Data Transfers

```python
# Fit.
m1 = sklearn.StandardScaler()
m2 = cuml .PCA()
m3 = cuml .KNeighborsClassifier()
X_train = m1.fit_transform(X_train)
X_train = transfer(X_train, GPU)
X_train = m2.fit_transform(X_train)
Y_train = transfer(Y_train, GPU)
Y_train = m3.fit(X_train, Y_train)

# Predict.
X_test = m1.transform(X_test)
X_test = transfer(X_test, GPU)
X_test = m2.transform(X_test)
result = m3.predict(X_test)
result = transfer(result, CPU)
plottinglib.plot(result)
```
Motivating Example

## Missing Functions

```python
X_train = m1.fit_transform(X_train)
X_train = transfer(X_train, GPU)
```

## Manual Data Transfers

```python
X_test[i,j] = transfer(X_test[i,j], GPU)
```

## Small GPU Memory

```python
result[i,j] = transfer(result[i,j], CPU)
```

### # Fit.

```python
m1 = sklearn.StandardScaler()
m2 = cuml.PCA()
m3 = cuml.KNeighborsClassifier()
X_train = m1.fit_transform(X_train)
X_train = transfer(X_train, GPU)
X_train = m2.fit_transform(X_train)
Y_train = transfer(Y_train, GPU)
m3.fit(X_train, Y_train)
for (i,j) in split(X_test):
    # Predict.
    X_test[i,j] = m1.transform(X_test[i,j])
    X_test[i,j] = transfer(X_test[i,j], GPU)
    X_test[i,j] = m2.transform(X_test[i,j])
    result[i,j] = m3.predict(X_test[i,j])
    result[i,j] = transfer(result[i,j], CPU)
plottinglib.plot(result)
```
Motivating Example

- Missing Functions
- Manual Data Transfers
- Small GPU Memory
- Scheduling

---

```python
# Fit.
m1 = sklearn.StandardScaler()
m2 = cuml.PCA()
m3 = cuml.KNeighborsClassifier()
X_train = m1.fit_transform(X_train)
X_train = transfer(X_train, GPU)
X_train = m2.fit_transform(X_train)
Y_train = transfer(Y_train, GPU)
m3.fit(X_train, Y_train)
for (i,j) in split(X_test):
    # Predict
    X_test[i,j] = m1.transform(X_test[i,j])
    X_test[i,j] = transfer(X_test[i,j], GPU)
    X_test[i,j] = m2.transform(X_test[i,j])
    result[i,j] = m3.predict(X_test[i,j])
    result[i,j] = transfer(result[i,j], CPU)
plottinglib.plot(result)
```
Solution: Offload Annotations

The **annotator** writes offload annotations (OAs) for CPU libraries. An **end user** imports the annotated library instead of the CPU library. Our **runtime**, Bach, automatically schedules data transfers and pages computation.
Goals

With less developer effort:

1. Match handwritten GPU performance
Goals

With less developer effort:

1. Match handwritten GPU performance
2. Scale to data sizes larger than GPU memory
Goals

With less developer effort:

1. Match handwritten GPU performance
2. Scale to data sizes larger than GPU memory
3. Beat CPU performance
Step 1: Annotator – Function Annotations

GPU library  CPU library

\[
\text{multiply} = @oa (func=torch.mul)(\text{np.multiply}) \\
\text{sqrt} = @oa (func=torch.sqrt)(\text{np.sqrt})
\]
Step 1: Annotator – Function Annotations

GPU library   CPU library

\[\text{multiply} = @\text{oa}(\text{func}=\text{torch.mul})(\text{np.multiply})\]
\[\text{sqrt} = @\text{oa}(\text{func}=\text{torch.sqrt})(\text{np.sqrt})\]

[Diagram showing the correspondence between GPU and CPU libraries]

Corresponding functions
Step 1: Annotator – Function Annotations

```python
arg = (NdArrayType(),)
args = (NdArrayType(), NdArrayType())
ret = NdArrayType()

multiply = @oa(args, ret, func=torch.mul)(np.multiply)
sqrt   = @oa(arg, ret, func=torch.sqrt)(np.sqrt)
```

↑ inputs  outputs
Step 1: Annotator – Allocation Annotations

arg = (NdArrayType(),)
args = (NdArrayType(), NdArrayType())
ret = NdArrayType()

multiply = @oa(args, ret, func=torch.mul)(np.multiply)
sqrt = @oa(arg, ret, func=torch.sqrt)(np.sqrt)
ones = @oa_alloc(ret, func=torch.ones)(np.ones)

Allocations only have a return type.
arg = (NdArrayType(),)
args = (NdArrayType(), NdArrayType())
ret = NdArrayType()

"offload split type"
multiply = @oa(args, ret, func=torch.mul)(np.multiply)
sqrt = @oa(arg, ret, func=torch.sqrt)(np.sqrt)
ones = @oa_alloc(ret, func=torch.ones)(np.ones)
<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>device(value)</code></td>
<td>Which device the value is on.</td>
</tr>
<tr>
<td><code>to(value, device)</code></td>
<td>Transfers [value] to [device].</td>
</tr>
</tbody>
</table>
## Step 1: Annotator – Offload Split Type

### API

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>device(value)</td>
<td>Which device the value is on.</td>
</tr>
<tr>
<td>to(value, device)</td>
<td>Transfers [value] to [device].</td>
</tr>
</tbody>
</table>

### Implementation for NdArrayType()

<table>
<thead>
<tr>
<th>API</th>
<th>Implementation for NdArrayType()</th>
</tr>
</thead>
<tbody>
<tr>
<td>device(value)</td>
<td>...if isinstance(value, torch.Tensor): ...</td>
</tr>
<tr>
<td>to(value, device)</td>
<td>...value.to(torch.device('cpu')).numpy()</td>
</tr>
</tbody>
</table>
Step 1: Annotator – Offload Split Type

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size(value)</td>
<td>Number of elements in the value.</td>
</tr>
<tr>
<td>split(start, end, value)</td>
<td>Splits a value to enable paging.</td>
</tr>
<tr>
<td>merge(values)</td>
<td>Merges split values.</td>
</tr>
</tbody>
</table>

splitting API
[Mozart SOSP '19] (optional)
### Step 1: Annotator – Offload Split Type

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size(value)</td>
<td>Number of elements in the value.</td>
</tr>
<tr>
<td>split(start, end, value)</td>
<td>Splits a value to enable paging.</td>
</tr>
<tr>
<td>merge(values)</td>
<td>Merges split values.</td>
</tr>
</tbody>
</table>

**Implementation for NdArrayType()**

<table>
<thead>
<tr>
<th>API</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>size(value)</td>
<td>return value.shape[-1]</td>
</tr>
<tr>
<td>split(start, end, value)</td>
<td>return value[start, end]</td>
</tr>
<tr>
<td>merge(values)</td>
<td>return np.concatenate(values)</td>
</tr>
</tbody>
</table>

splitting API

[Mozart SOSP ‘19](https://www.mozartresearch.com) (optional)
Step 1: Annotator – Offload Split Type

- NumPy NdArrayType()
- pandas DataFrameType()
- scikit-learn ModelType()
import numpy as np

# Allocate
a = np.ones(size, dtype='float64')
b = np.ones(size, dtype='float64')

# Compute
np.arcsin(a, out=a)
np.multiply(a, b, out=b)
np.sqrt(b, out=b)

(Simple, somewhat dumb, Python program.)
import bach.numpy as np

# Allocate
a = np.ones(size, dtype='float64')
b = np.ones(size, dtype='float64')

# Compute
np.arcsin(a, out=a)
np.multiply(a, b, out=b)
np.sqrt(b, out=b)

Import the annotated library instead of the CPU library.
import bach.numpy as np

# Allocate
a = np.ones(size, dtype='float64')
b = np.ones(size, dtype='float64')

# Compute
np.arcsin(a, out=a)
np.multiply(a, b, out=b)
np.sqrt(b, out=b)

np.evaluate()  # Explicitly materialize lazy values with included evaluate() function.
Step 3: Runtime - Scheduling

Generate a lazy computation graph and do a topological sort.
Assign functions to the CPU/GPU based on whether a GPU library implementation is provided in the annotation.
Assign allocations to the CPU/GPU so they are on the same device as the first function that uses the data.

\[ a = \text{np.ones()} \]
\[ \text{np.arcsin}(a) \]
\[ b = \text{np.ones()} \]
\[ \text{np.multiply}(a,b) \]
\[ \text{np.sqrt}(b) \]
Automatically transfer data using the offloading API.

Step 3: Runtime – Offloading API

Bach

---

1. `a = np.ones()`
2. `np.arcsin(a)`
3. `b = np.ones()`
4. `np.multiply(a, b)`
5. `np.sqrt(b)`

---

1. -- transfer to GPU --
2. -- transfer to CPU --
Automatically page large datasets using the *splitting API*. 

**Step 3: Runtime – Splitting API**

Data Size = $2^{28}$

1. $a = \text{np.ones()}$
2. $\text{np.arcsin}(a)$
3. $b = \text{np.ones()}$
4. $\text{np.multiply}(a, b)$
5. $\text{np.sqrt}(b)$

---

**Split**

1. $a = \text{np.ones()}$
2. $\text{np.arcsin}(a)$

---

3. $b = \text{np.ones()}$
4. $\text{np.multiply}(a, b)$
5. $\text{np.sqrt}(b)$

---

**Merge**

---

**Allocation**

---

**CPU**

---

**GPU**
Step 3: Runtime – Scheduling Heuristics

Naive cost-benefit analysis between data transfer and computation cost.

Data Size = $2^{28}$

CPU Compute

GPU Compute + Data Transfer
Data Size = $2^{10}$

Naive cost-benefit analysis between data transfer and computation cost.
Naive implementations of cost estimators.
Evaluation

4 library integrations and 8 data science and ML workloads.
Integration Experience

<table>
<thead>
<tr>
<th>CPU-only library</th>
<th>GPU kernel library</th>
<th>LOC</th>
<th># Split Types</th>
<th># Funcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumPy</td>
<td>CuPy</td>
<td>103</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>NumPy</td>
<td>PyTorch</td>
<td>90</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Pandas</td>
<td>cuDF</td>
<td>241</td>
<td>7</td>
<td>27</td>
</tr>
<tr>
<td>Scikit-learn</td>
<td>cuML</td>
<td>81</td>
<td>2</td>
<td>12</td>
</tr>
</tbody>
</table>

~130 LOC per library including offloading / splitting APIs and function annotations.
Evaluation: Summary

<table>
<thead>
<tr>
<th>Workload</th>
<th>Ops</th>
<th>CPU Library</th>
<th>Max Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black-Scholes</td>
<td>39</td>
<td>NumPy$^1$</td>
<td>5.7×</td>
</tr>
<tr>
<td>Black-Scholes</td>
<td>39</td>
<td>NumPy$^2$</td>
<td>6.9×</td>
</tr>
<tr>
<td>Haversine</td>
<td>19</td>
<td>NumPy$^1$</td>
<td>0.81×</td>
</tr>
<tr>
<td>Haversine</td>
<td>19</td>
<td>NumPy$^2$</td>
<td>1.7×</td>
</tr>
<tr>
<td>Crime Index</td>
<td>15</td>
<td>Pandas</td>
<td>4.6×</td>
</tr>
<tr>
<td>DBSCAN</td>
<td>7</td>
<td>NumPy$^1$/Sklearn</td>
<td>1200×</td>
</tr>
<tr>
<td>PCA</td>
<td>8</td>
<td>Sklearn</td>
<td>6.8×</td>
</tr>
<tr>
<td>TSVD</td>
<td>2</td>
<td>Sklearn</td>
<td>11×</td>
</tr>
</tbody>
</table>

Speedup: max 1200x, median 6.3x.
Evaluation: Summary

(b) Black-Scholes (Torch).
Evaluation: Summary

With less developer effort, Bach can:
1. Match handwritten GPU performance

(b) Black-Scholes (Torch).
Evaluation: Summary

With less developer effort, Bach can:

1. Match handwritten GPU performance
2. Scale to data sizes larger than GPU memory
Evaluation: Summary

With less developer effort, Bach can:
1. Match handwritten GPU performance
2. Scale to data sizes larger than GPU memory
3. Beat CPU performance

(b) Black-Scholes (Torch)
Crime Index saves time by eliminating the initial data transfer, while the allocation still fits in GPU memory.
At smaller data sizes, TSVD schedules all computation on the CPU.
In-Depth Evaluation: Splitting/Paging Datasets

[Motivating Example]
The "fit" phase dominates the runtime until the "predict" phase can split/page data into the GPU.
Evaluation: Summary

Max Speedup

(a) Black-Scholes (CuPy).
(b) Black-Scholes (Torch).
(c) Crime Index.
(d) Haversine (CuPy).
(e) Haversine (Torch).
(f) DBSCAN.
(g) PCA.
(h) TSVD.
Evaluation: Summary

Max Speedup

(a) Black-Scholes (CuPy).
(b) Black-Scholes (Torch).
(c) Crime Index.
(d) Haversine (CuPy).

(e) Haversine (Torch).
(f) DBSCAN.
(g) PCA.
(h) TSVD.
Conclusion

OAs enable heterogeneous GPU computing in existing libraries and workloads with little to no code modifications.

With less developer effort, Bach + OAs can:

- Match handwritten GPU performance
- Scale to data sizes larger than GPU memory
- Beat CPU performance

[github.com/stanford-futuredata/offload-annotations](https://github.com/stanford-futuredata/offload-annotations)

[gyuan@cs.stanford.edu](mailto:gyuan@cs.stanford.edu)