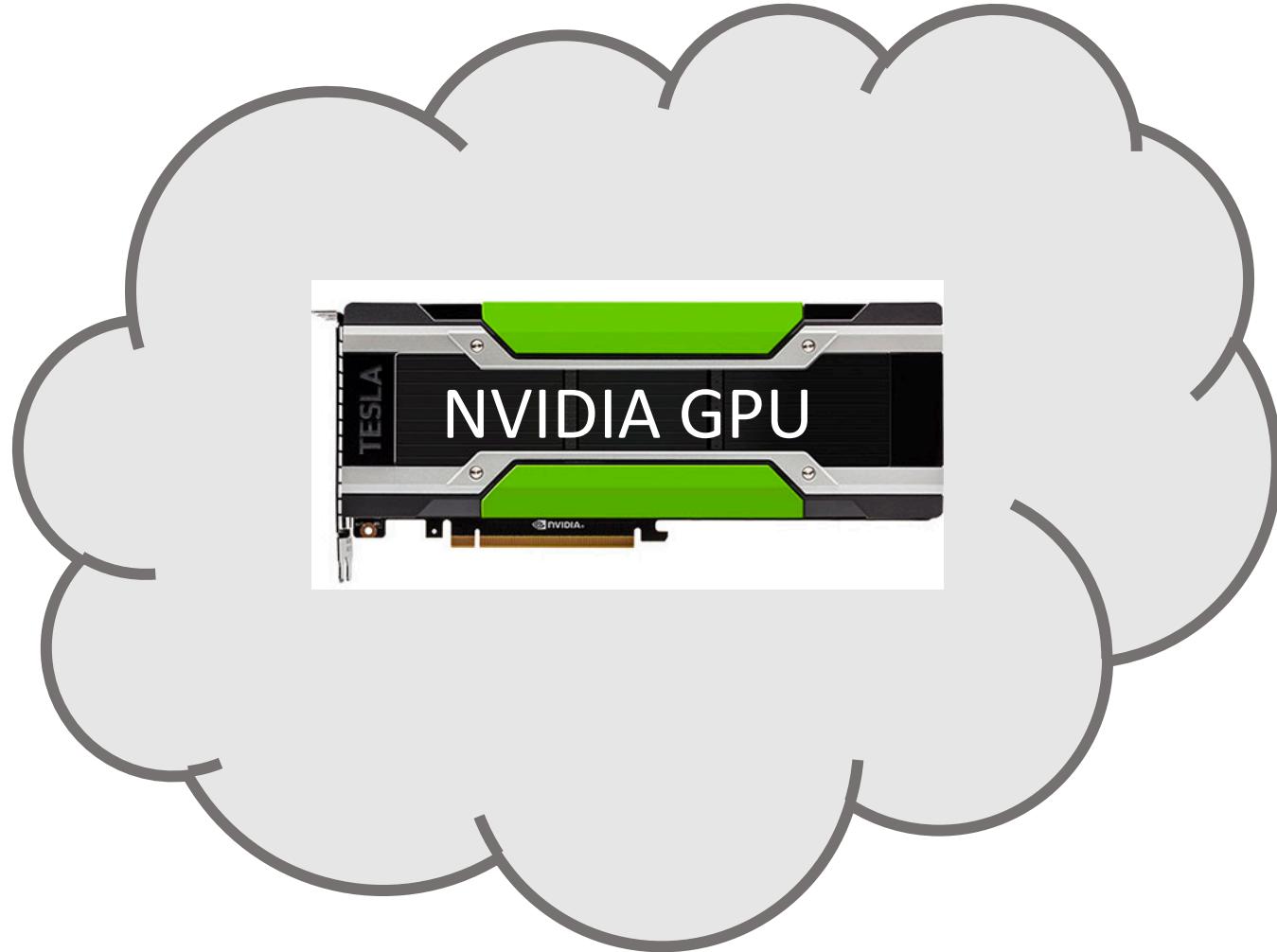


Offload Annotations: Bringing Heterogeneous Computing to Existing Libraries and Workloads

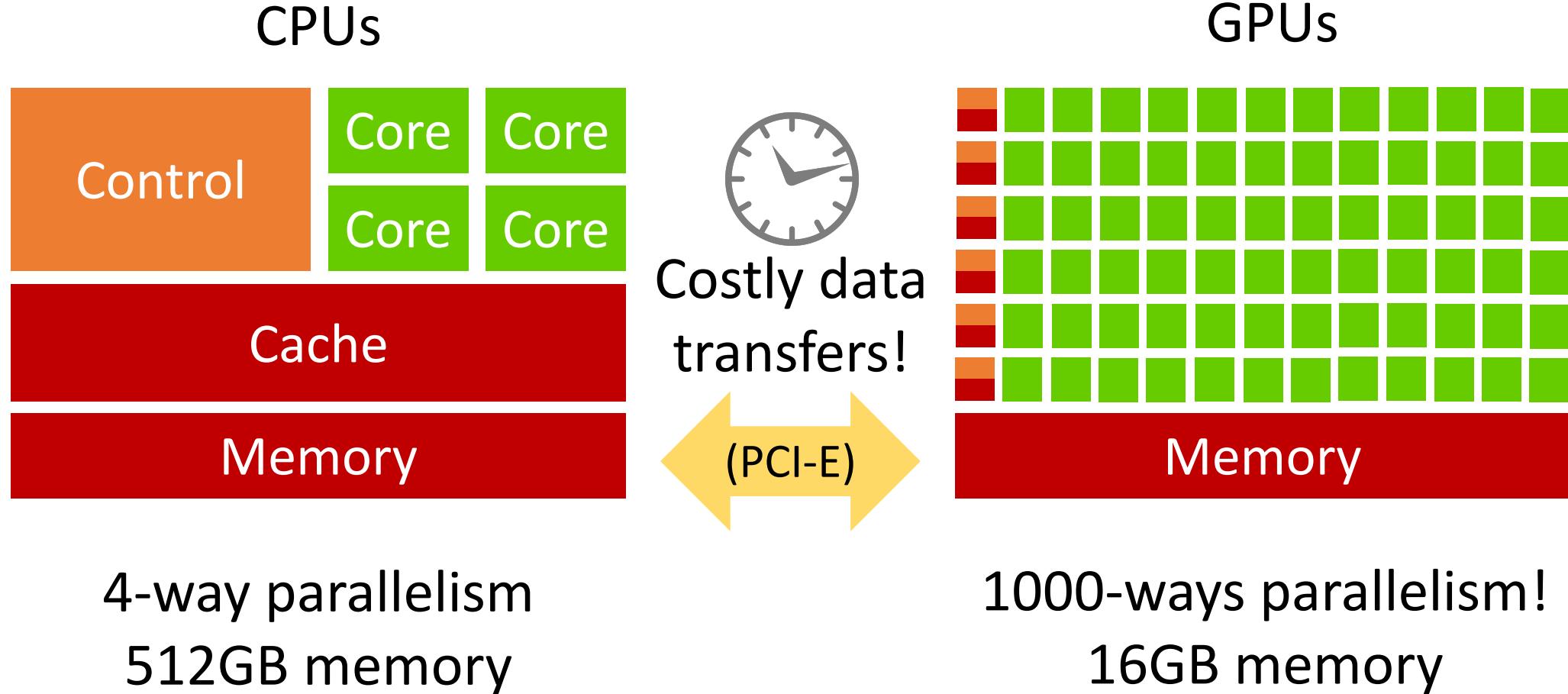
Gina Yuan, Shoumik Palkar, Deepak Narayanan, Matei Zaharia
Stanford University

USENIX ATC 2020 (July 15-17)

Background: Hardware Commoditization



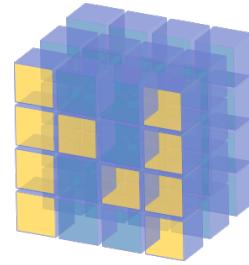
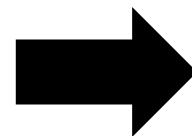
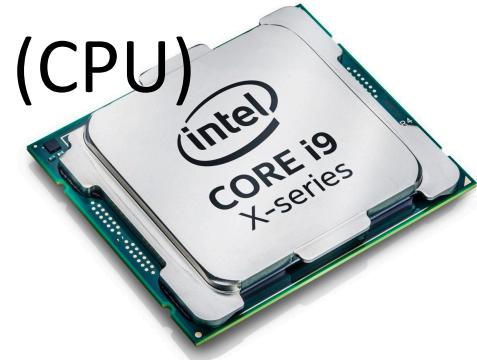
Background: CPUs vs. GPUs



Background: Data Science on the CPU



+



NumPy

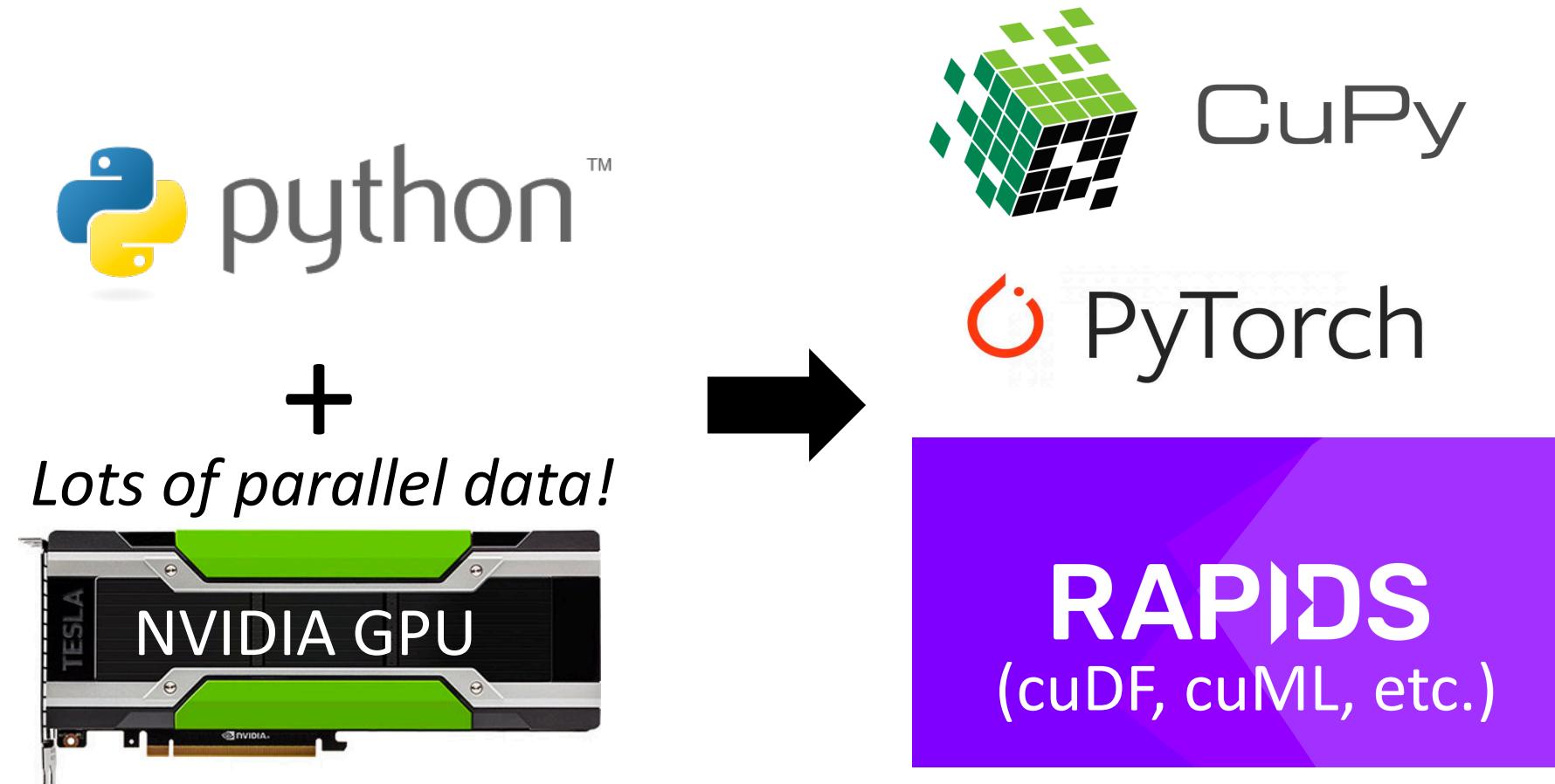


pandas



Popular Python data science libraries for the CPU.

Trend: Data Science on the GPU



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 and easily accelerate their workflows without going into the details of CUDA programming.

<https://cupy.chainer.org/>

HIGHLY COMPATIBLE WITH NUMPY

CuPy's interface is highly compatible with NumPy; in most cases it can be used as a drop-in replacement. All you have to do is just replace numpy with cupy in your code.

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

WHAT IS PYTORCH? 🔗

A replacement for NumPy to use the power of GPUs

PyTorch is just replace numpy with cupy in your code.

For example, [Basics of CuPy \(Tutorial\)](#) is useful to learn how to use CuPy.

<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>

cuDF provides a pandas-like API that allows data scientists and analysts to easily accelerate their work.

<https://pytorch.org/>

WHAT IS
A replacement for NumPy

<https://github.com/rapidsai/cuml>

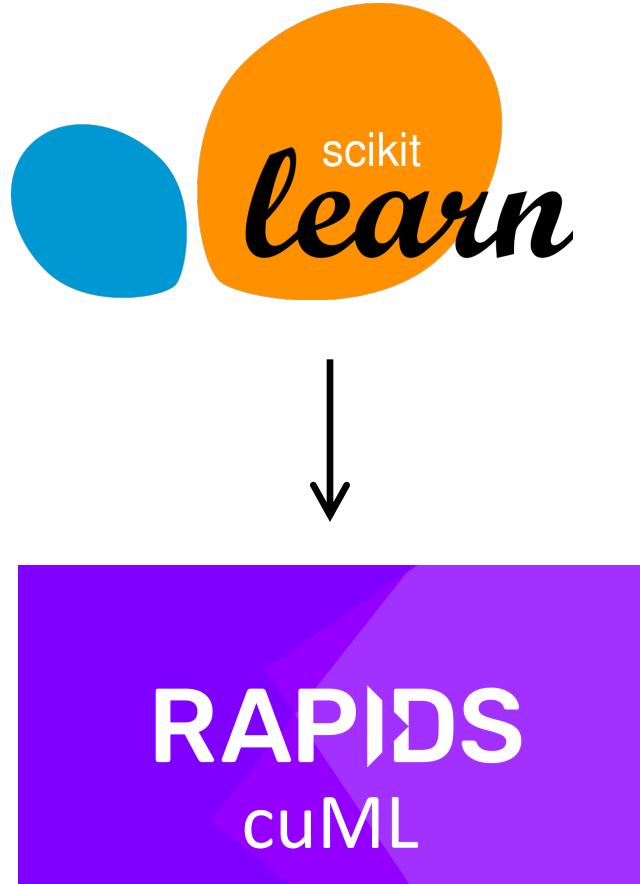
cuML enables data scientists, researchers, and software engineers to run traditional tabular ML tasks on GPUs without diving into the details of CUDA programming. In most cases, cuML's Python API matches the API from [scikit-learn](#).

<https://github.com/rapidsai/cugraph>

WITH NUMPY

cuGraph is compatible with NumPy; it provides a NumPy API replacement. All you need to do is replace `numpy` with `cupy` in your code. It's useful to learn

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

Missing Functions

```
# 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

```
# 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)
```

```
# 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

Manual Data Transfers

Small GPU Memory

```
# 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)
```

Motivating Example

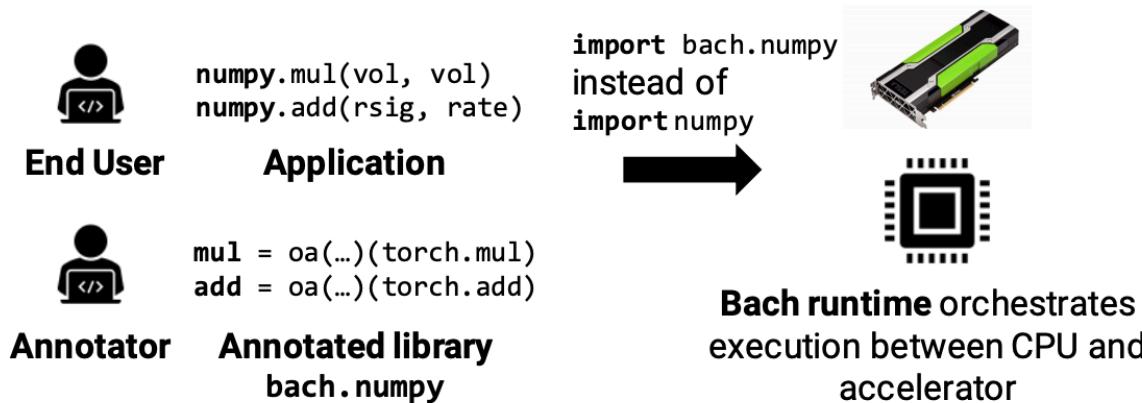
Missing Functions

Manual Data Transfers

Small GPU Memory

Scheduling

```
# 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:

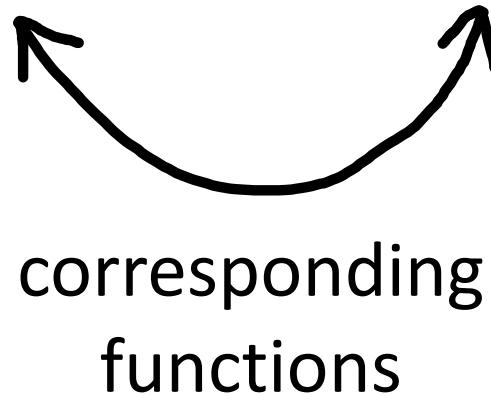
1. Match handwritten GPU performance
2. Scale to data sizes larger than GPU memory
3. Beat CPU performance

Step 1: Annotator – Function Annotations

GPU library CPU library
↓ ↓
`multiply = @oa(func=torch.mul)(np.multiply)`
`sqrt = @oa(func=torch.sqrt)(np.sqrt)`

Step 1: Annotator – Function Annotations

```
GPU library  CPU library  
↓           ↓  
multiply = @oa(func=torch.mul)(np.multiply)  
sqrt      = @oa(func=torch.sqrt)(np.sqrt)
```



corresponding
functions

Step 1: Annotator – Function 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)

          ↑   ↑
        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.

Step 1: Annotator – Allocation Annotations

```
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)
```

What's in an offload split type?

Step 1: Annotator – Offload Split Type

API	Description	
<code>device(value)</code>	Which device the value is on.	
<code>to(value, device)</code>	Transfers [value] to [device].	<i>offloading API</i>

Step 1: Annotator – Offload Split Type

API	Description	
<code>device(value)</code>	Which device the value is on.	<i>offloading API</i>
<code>to(value, device)</code>	Transfers [value] to [device].	

API	Implementation for NdArrayType()
<code>device(value)</code>	<code>...if isinstance(value, torch.Tensor): ...</code>
<code>to(value, device)</code>	<code>...value.to(torch.device('cpu')).numpy()</code>

Step 1: Annotator – Offload Split Type

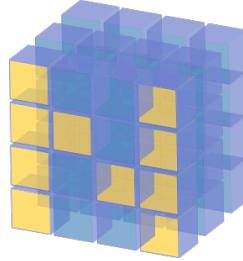
API	Description	
<code>size(value)</code>	Number of elements in the value.	<i>splitting API</i>
<code>split(start, end, value)</code>	Splits a value to enable paging.	<i>[Mozart SOSP '19]</i>
<code>merge(values)</code>	Merges split values.	(optional)

Step 1: Annotator – Offload Split Type

API	Description	
<code>size(value)</code>	Number of elements in the value.	
<code>split(start, end, value)</code>	Splits a value to enable paging.	
<code>merge(values)</code>	Merges split values.	<i>splitting API [Mozart SOSP '19] (optional)</i>

API	Implementation for NdArrayType()
<code>size(value)</code>	<code>return value.shape[-1]</code>
<code>split(start, end, value)</code>	<code>return value[start, end]</code>
<code>merge(values)</code>	<code>return np.concatenate(values)</code>

Step 1: Annotator – Offload Split Type



NumPy

NdArrayType()



pandas

DataFrameType()



ModelType()

Step 2: End User

```
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)
```



End User ≠
Annotator

(Simple, somewhat dumb, Python program.)

Step 2: End User

```
import batch.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.

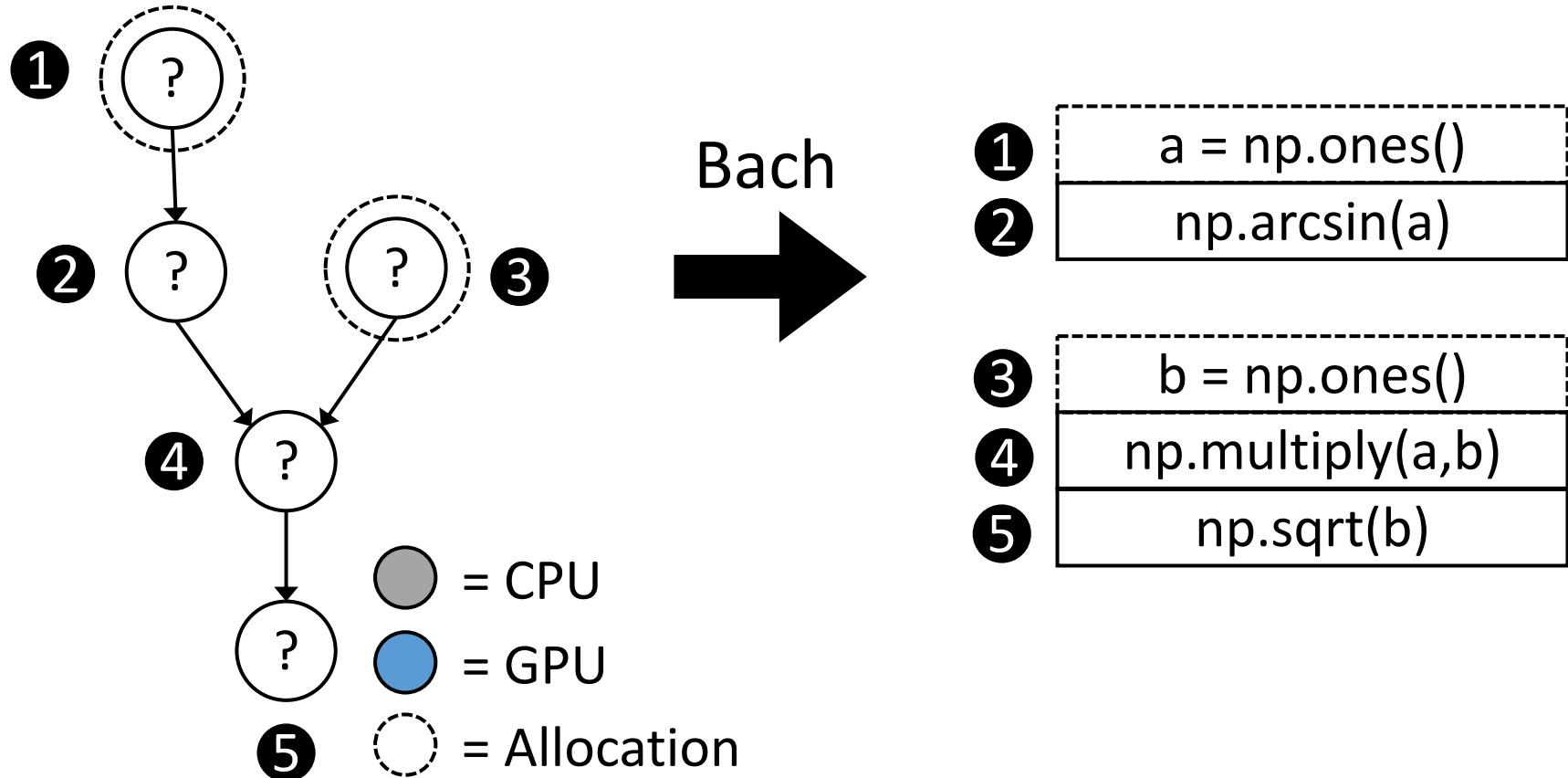
Step 2: End User

```
import batch.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)
```

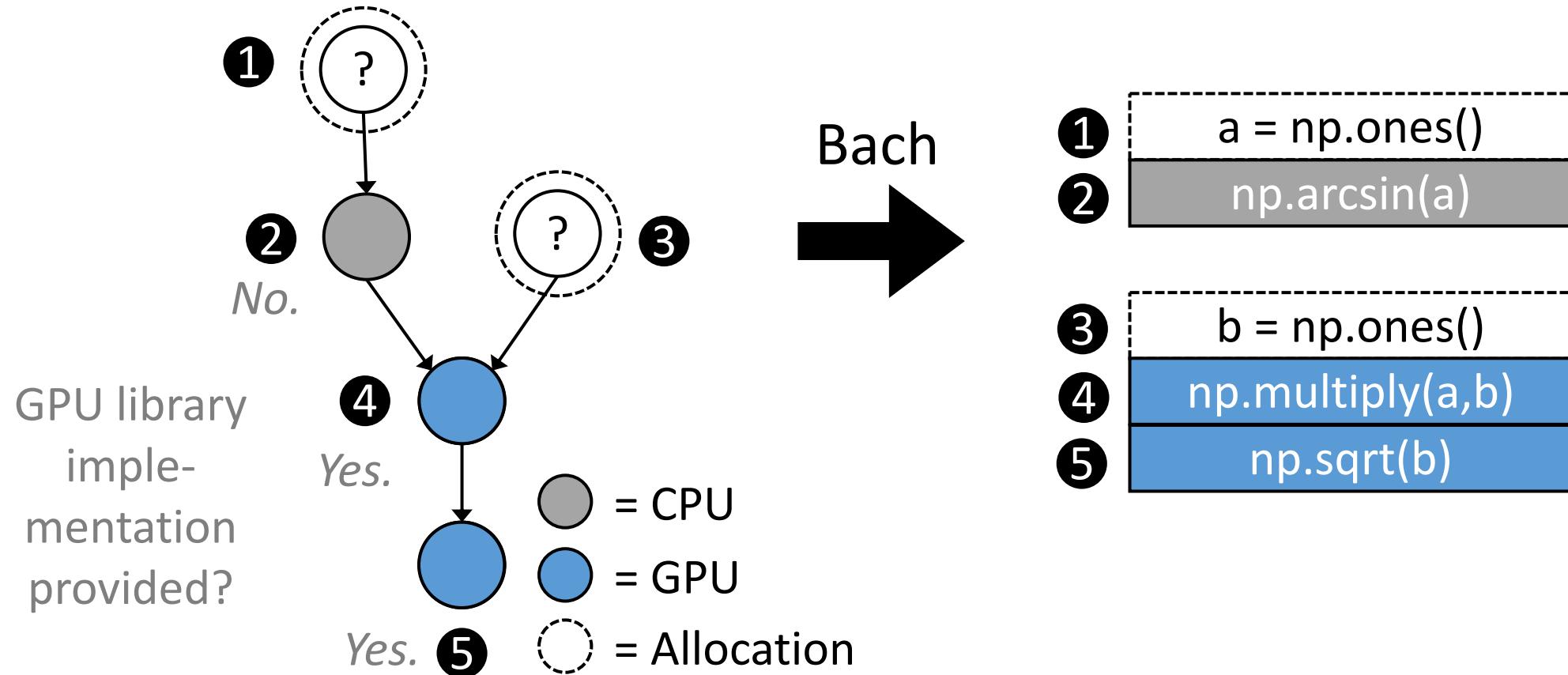
`np.evaluate()`

Explicitly materialize lazy values with included evaluate() function.

Step 3: Runtime - Scheduling

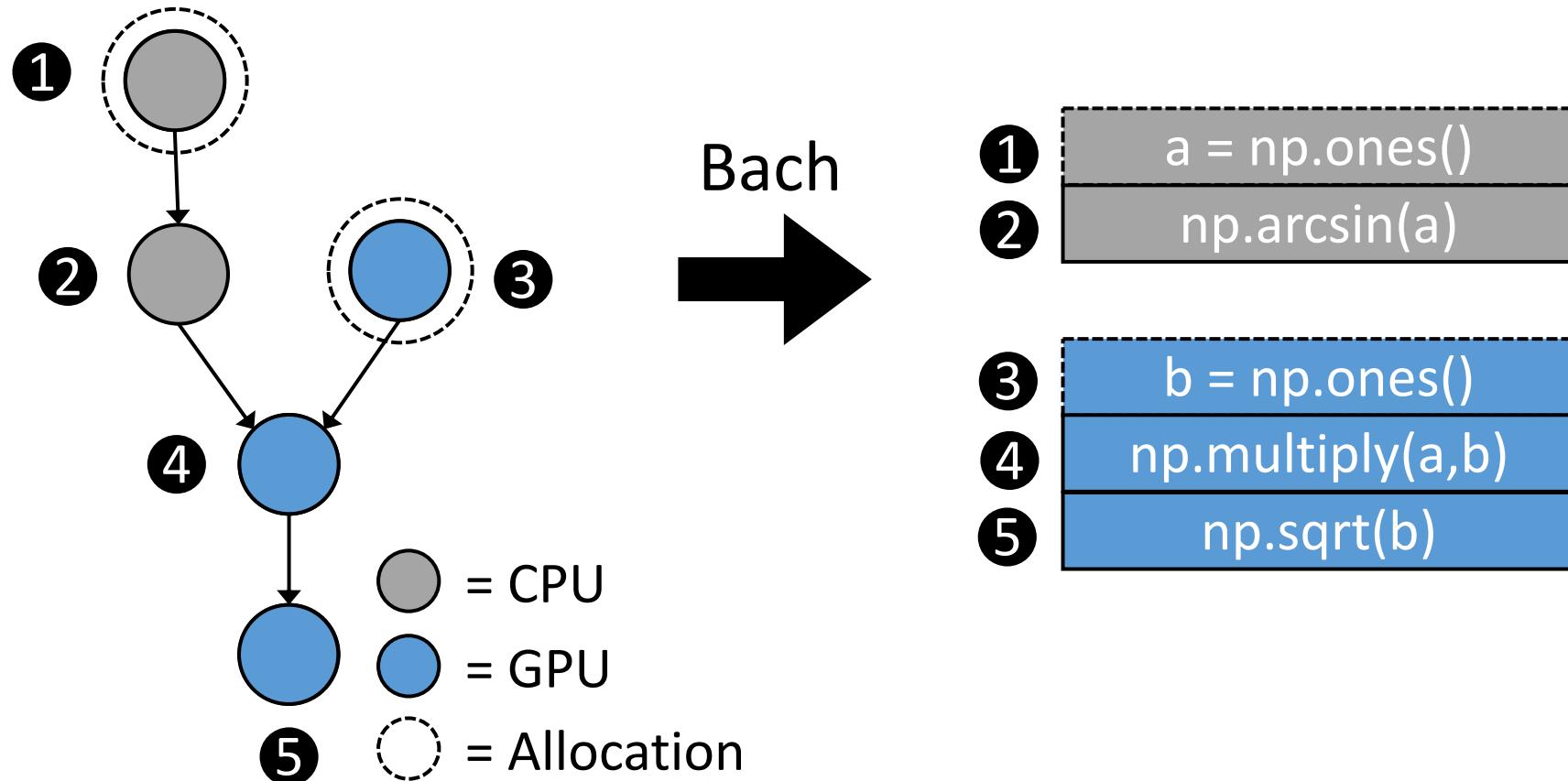


Step 3: Runtime - Scheduling



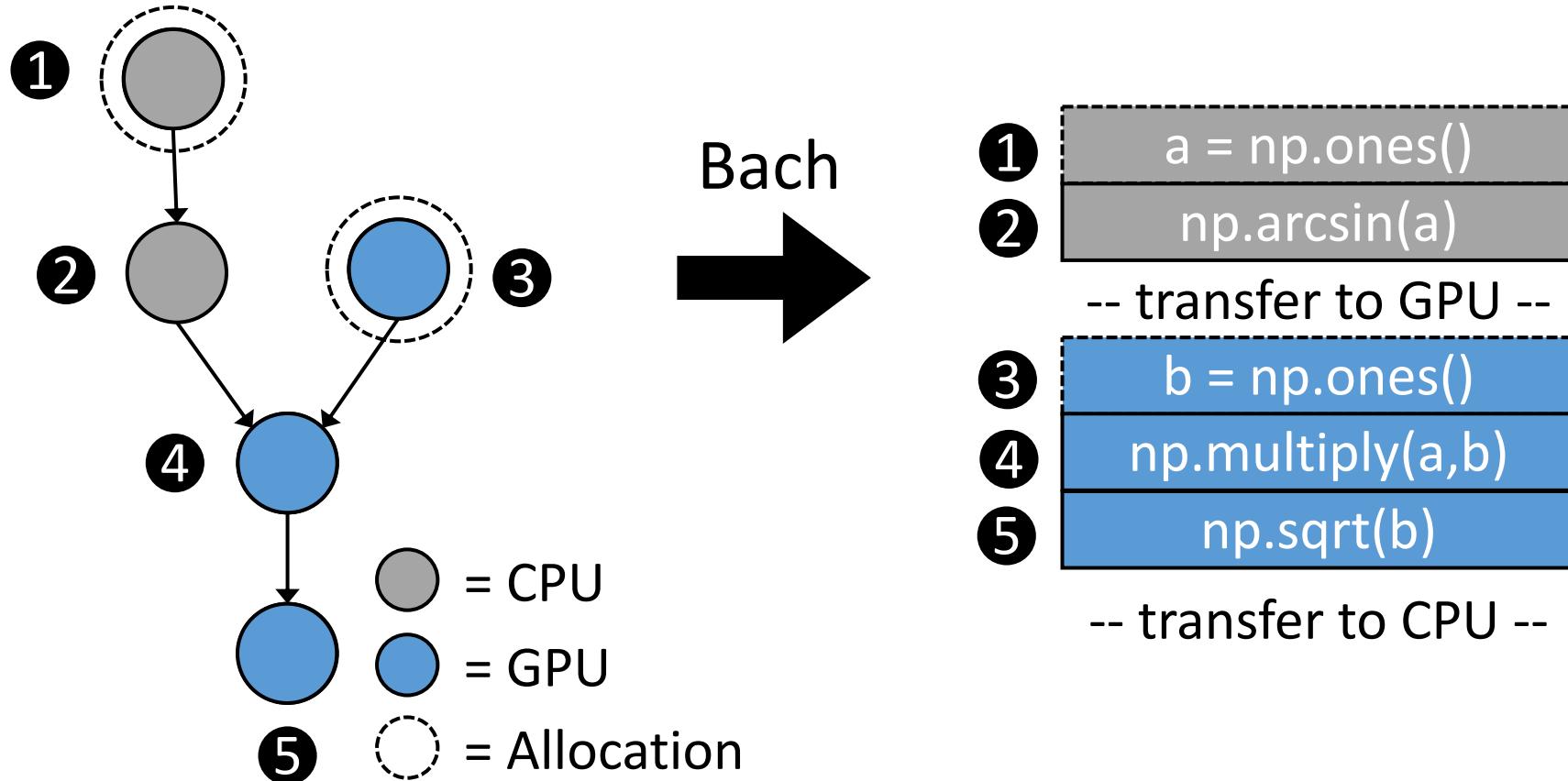
Assign functions to the CPU/GPU based on whether a GPU library implementation is provided in the annotation.

Step 3: Runtime - Scheduling



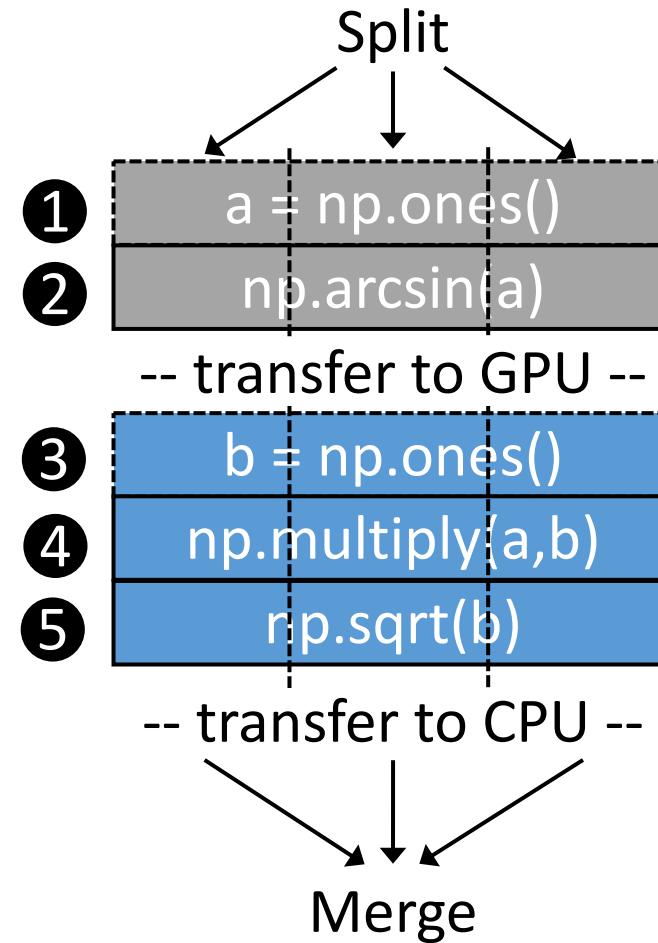
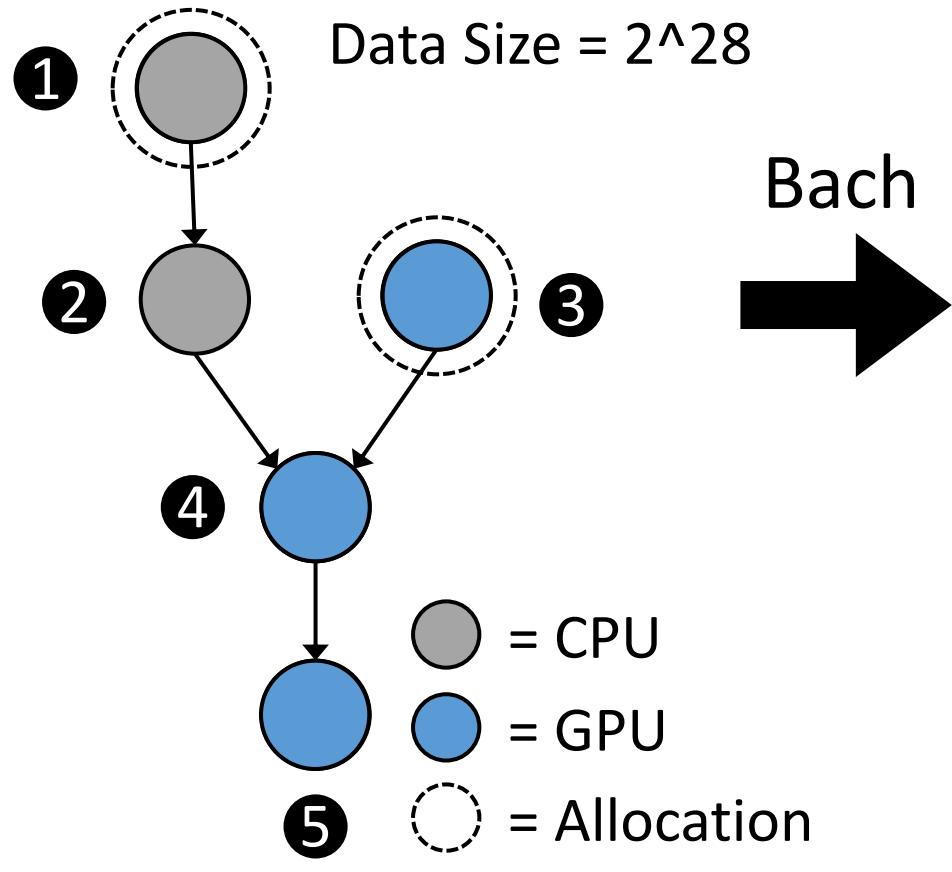
Assign allocations to the CPU/GPU so they are on the same device as the first function that uses the data.

Step 3: Runtime – Offloading API

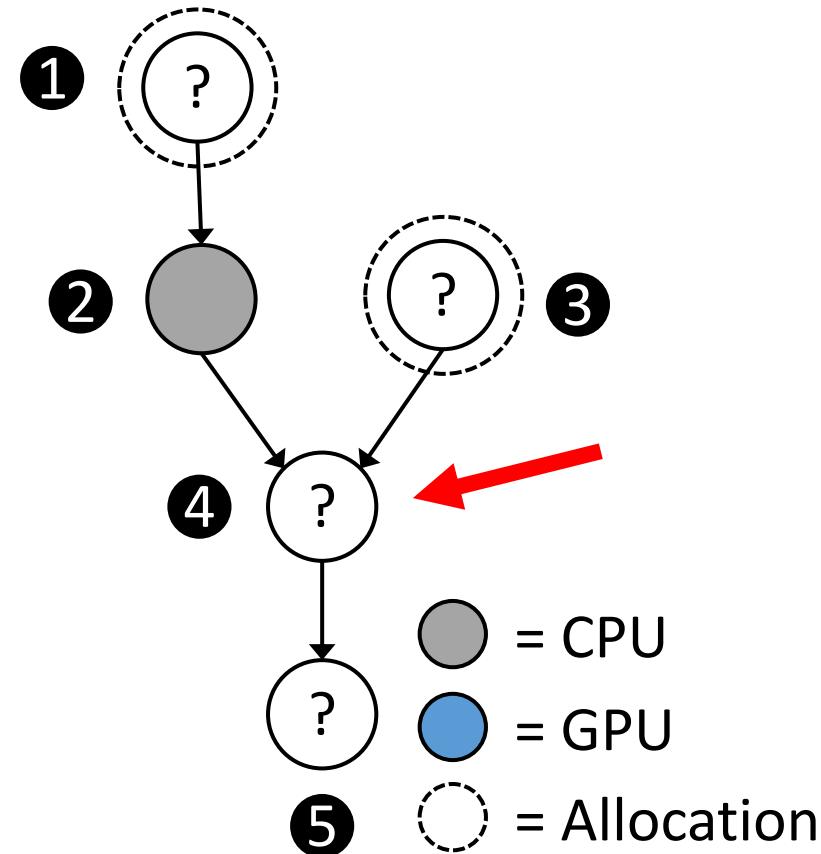
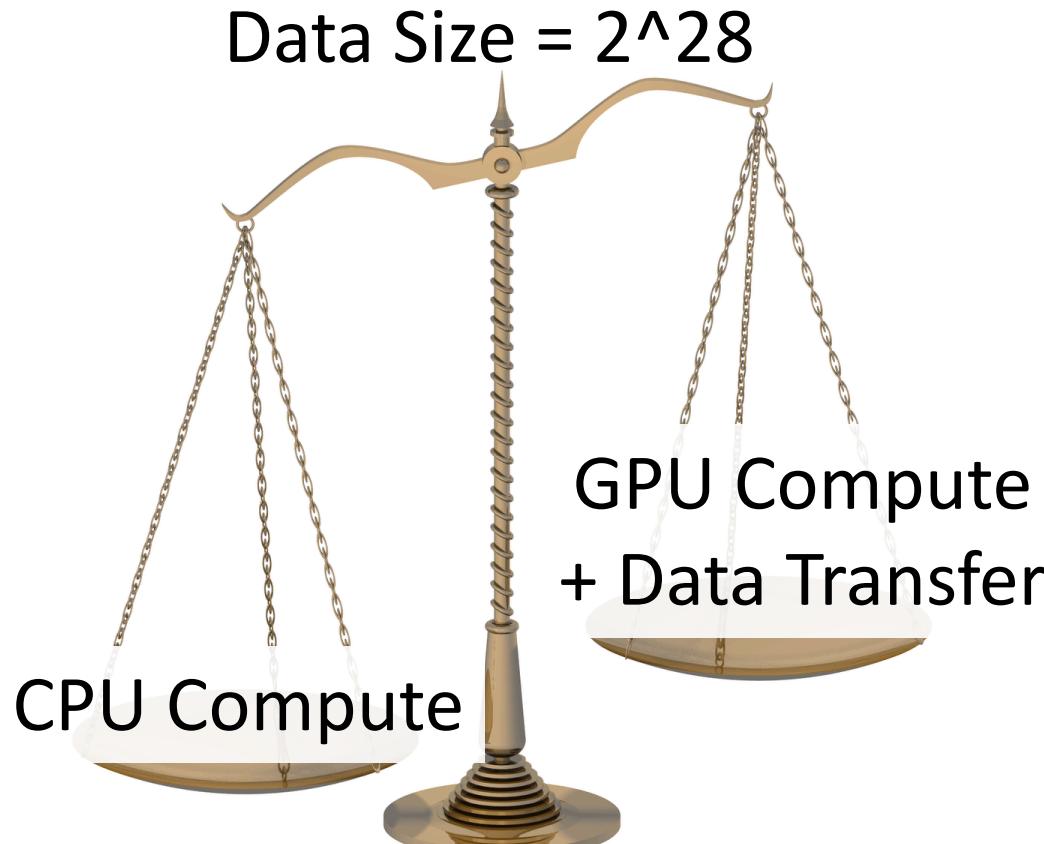


Automatically transfer data using the *offloading API*.

Step 3: Runtime – Splitting API

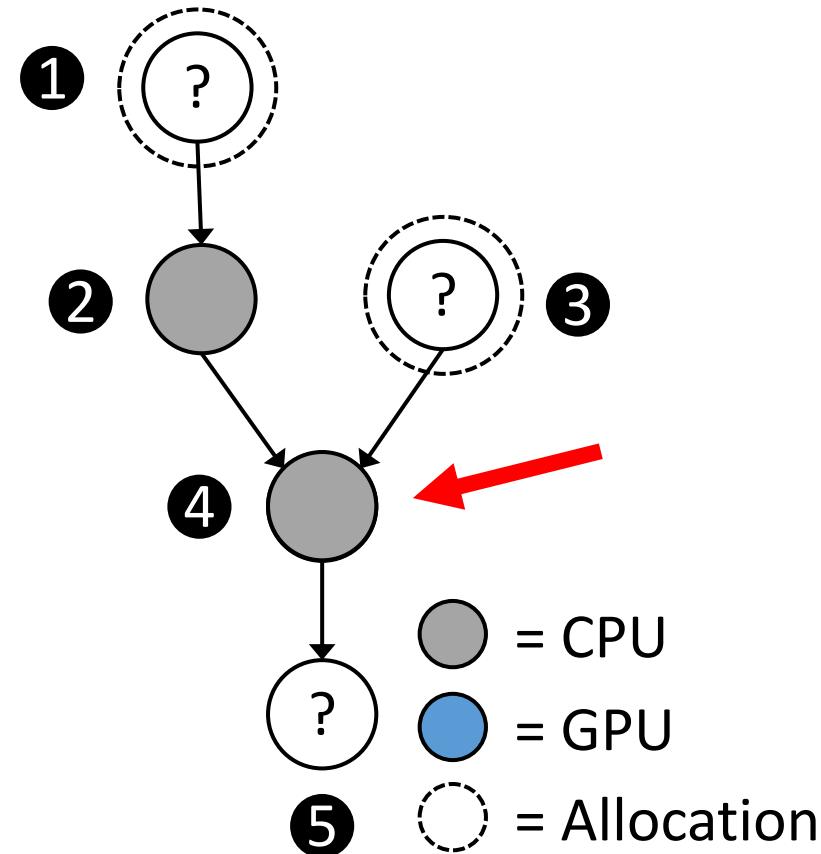
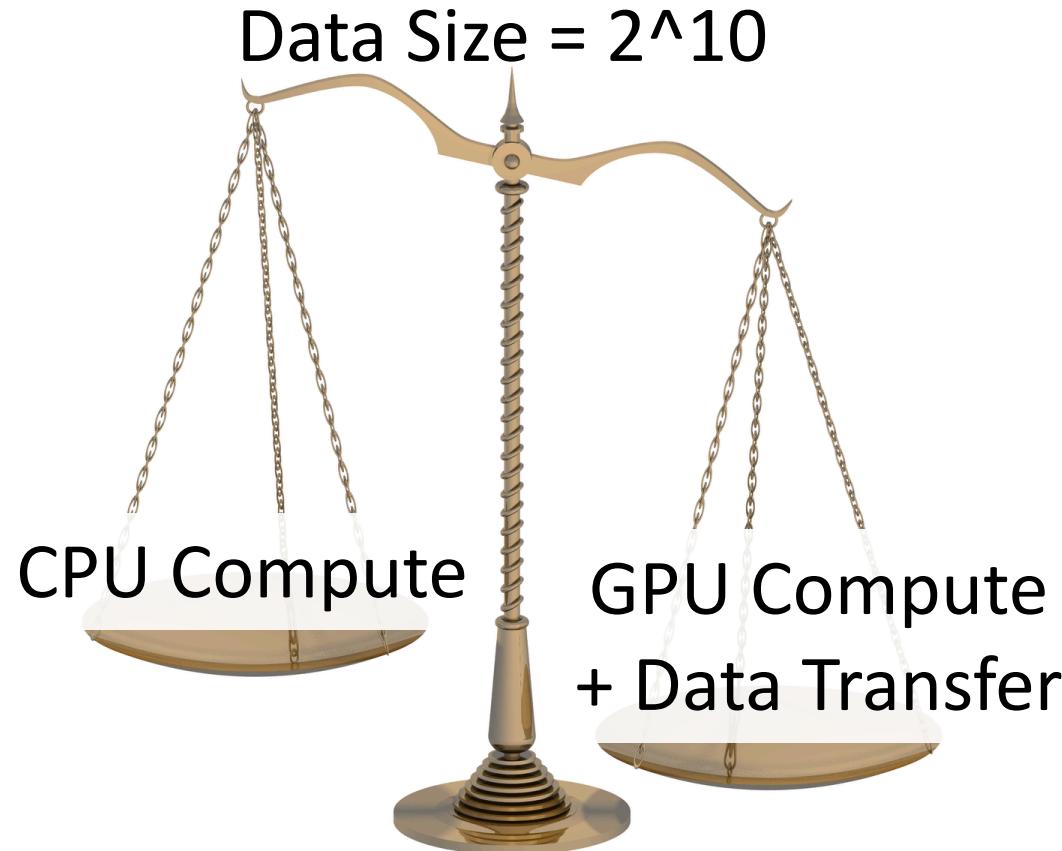


Step 3: Runtime – Scheduling Heuristics (optional)



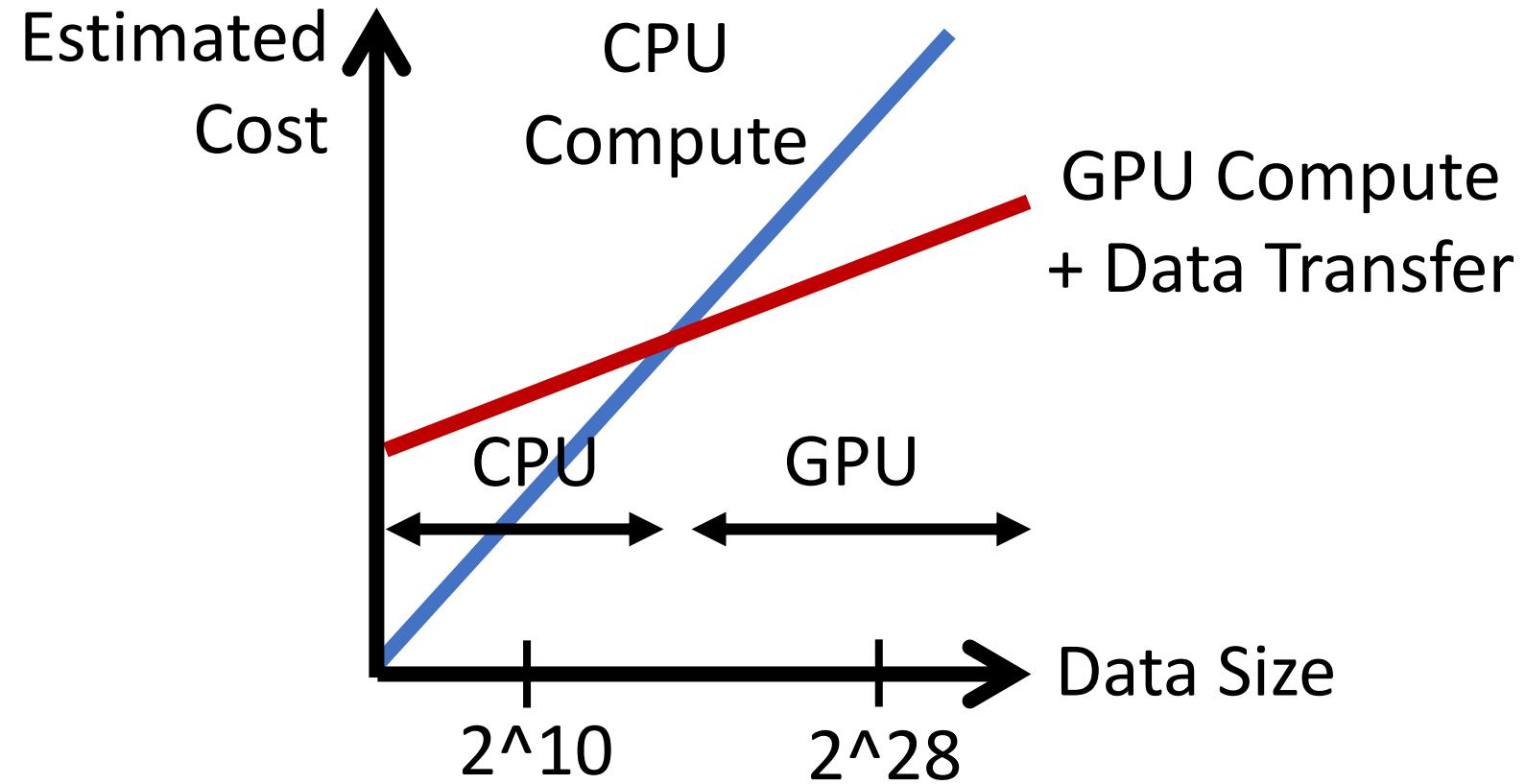
Naive cost-benefit analysis between data transfer and computation cost.

Step 3: Runtime – Scheduling Heuristics (optional)



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

Step 3: Runtime – Scheduling Heuristics (optional)



Naive implementations of cost estimators.

Evaluation

4 library integrations and 8 data science and ML workloads.

Integration Experience

CPU-only library	GPU kernel library	LOC	# Split Types	# Funcs
NumPy	CuPy	103	1	20
NumPy	PyTorch	90	1	10
Pandas	cuDF	241	7	27
Scikit-learn	cuML	81	2	12

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

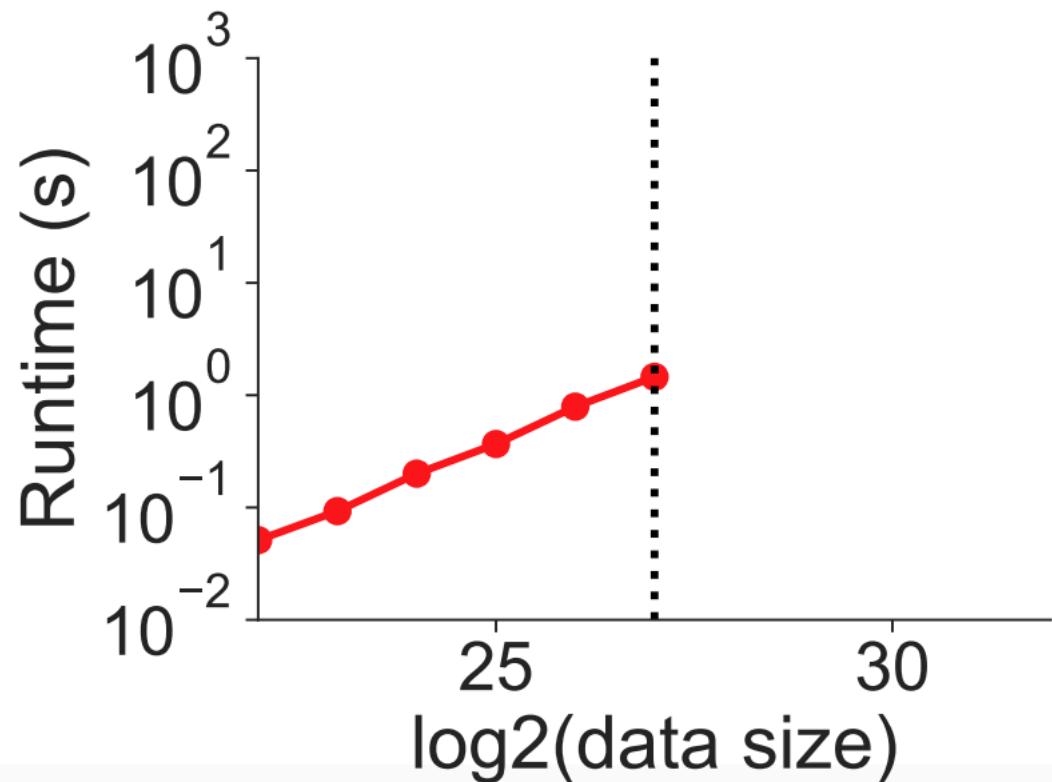
Evaluation: Summary

Workload	Ops	CPU Library	Max Speedup
Black-Scholes	39	NumPy ¹	5.7×
Black-Scholes	39	NumPy ²	6.9×
Haversine	19	NumPy ¹	0.81×
Haversine	19	NumPy ²	1.7×
Crime Index	15	Pandas	4.6×
DBSCAN	7	NumPy ¹ /Sklearn	1200×
PCA	8	Sklearn	6.8×
TSVD	2	Sklearn	11×

Speedup: max **1200x**, median **6.3x**.

Evaluation: Summary

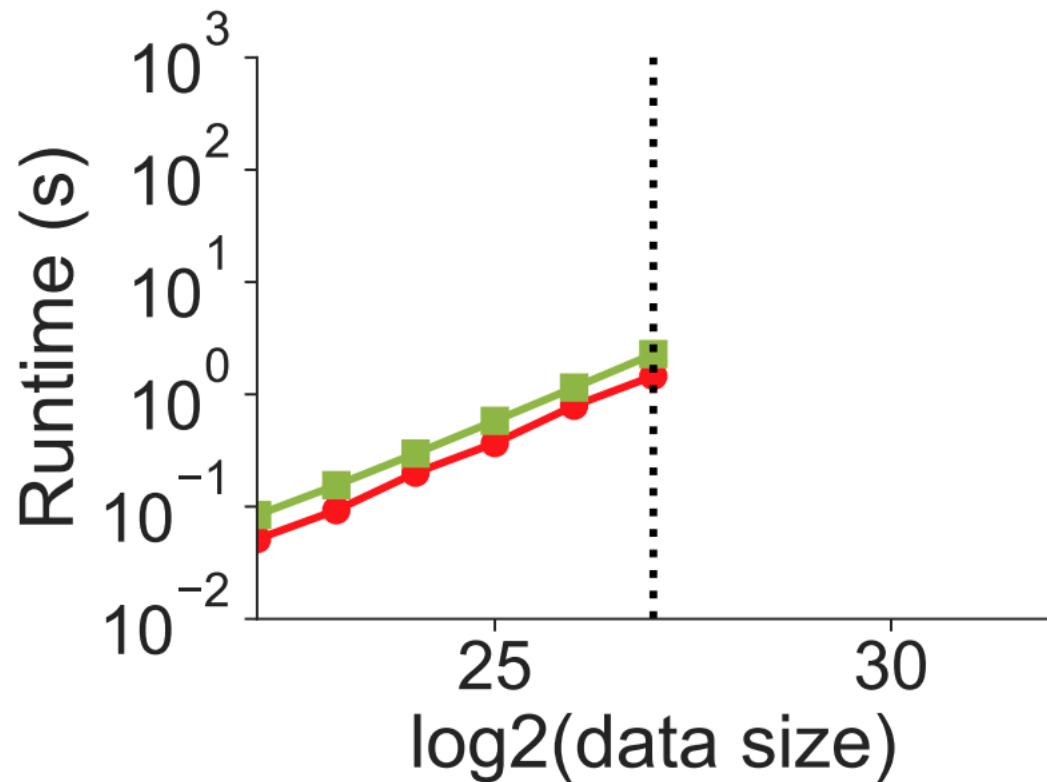
—▲— CPU Library —●— GPU Library —■— Bach



(b) Black-Scholes (Torch).

Evaluation: Summary

—▲— CPU Library —●— GPU Library —■— Bach



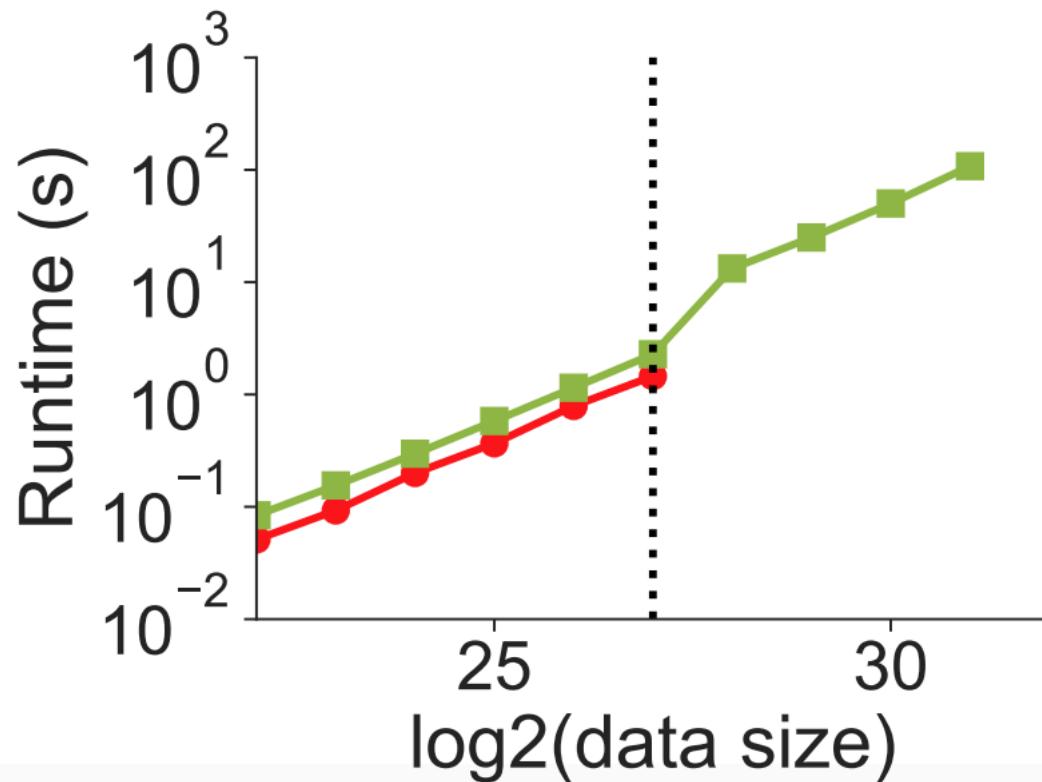
With less developer effort, Bach can:

1. Match handwritten GPU performance

(b) Black-Scholes (Torch).

Evaluation: Summary

—▲— CPU Library —●— GPU Library —■— Bach



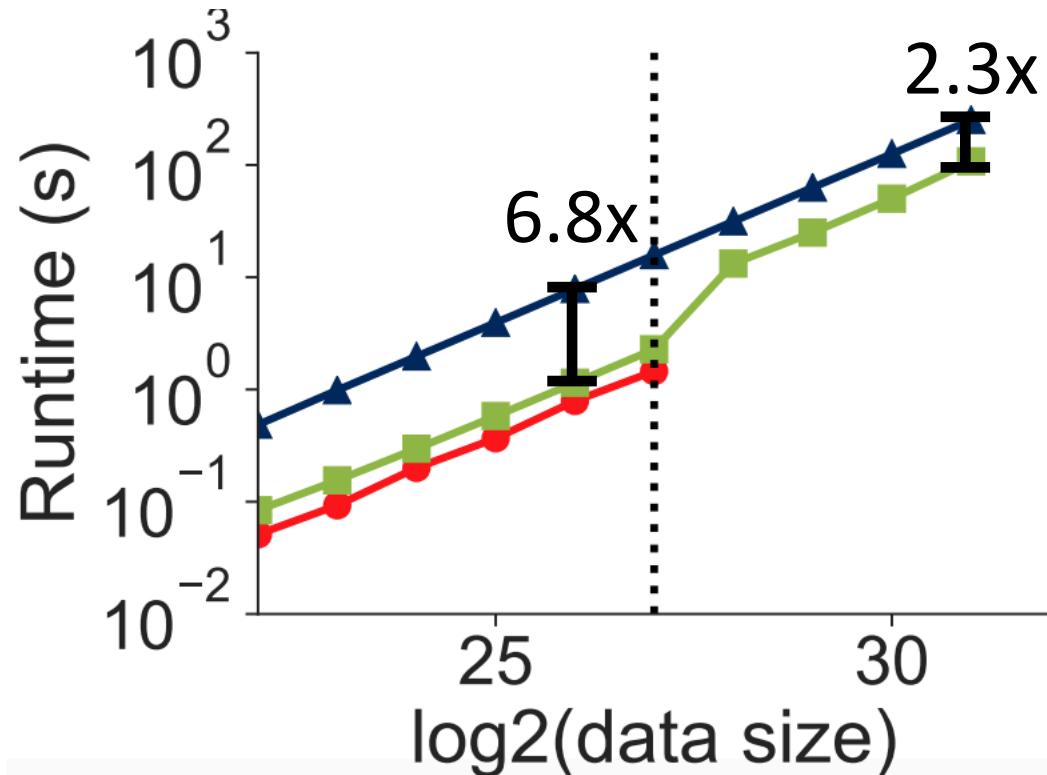
(b) Black-Scholes (Torch).

With less developer effort, Bach can:

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

Evaluation: Summary

—▲— CPU Library —●— GPU Library —■— Bach

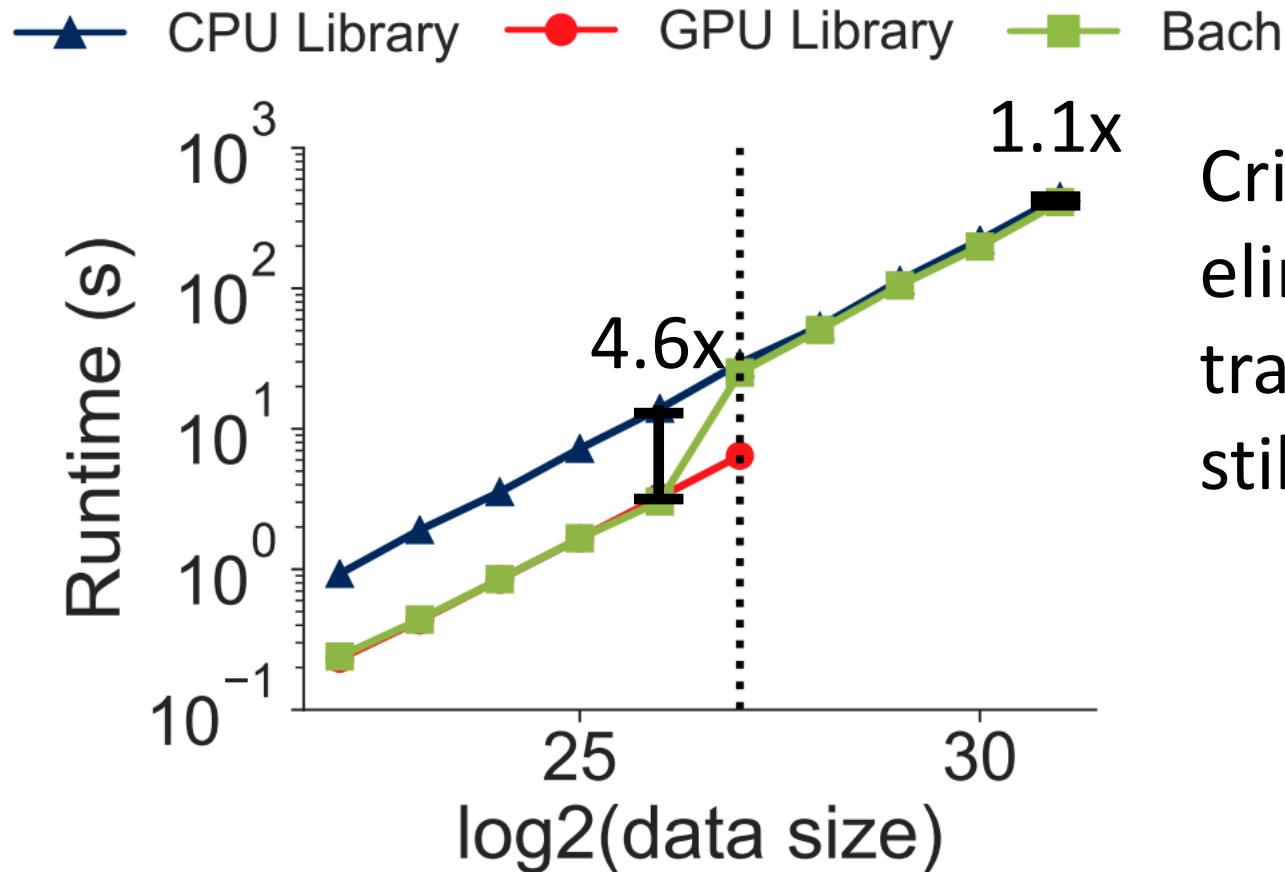


(b) Black-Scholes (Torch).

With less developer effort, Bach can:

1. Match handwritten GPU performance
2. Scale to data sizes larger than GPU memory
3. Beat CPU performance

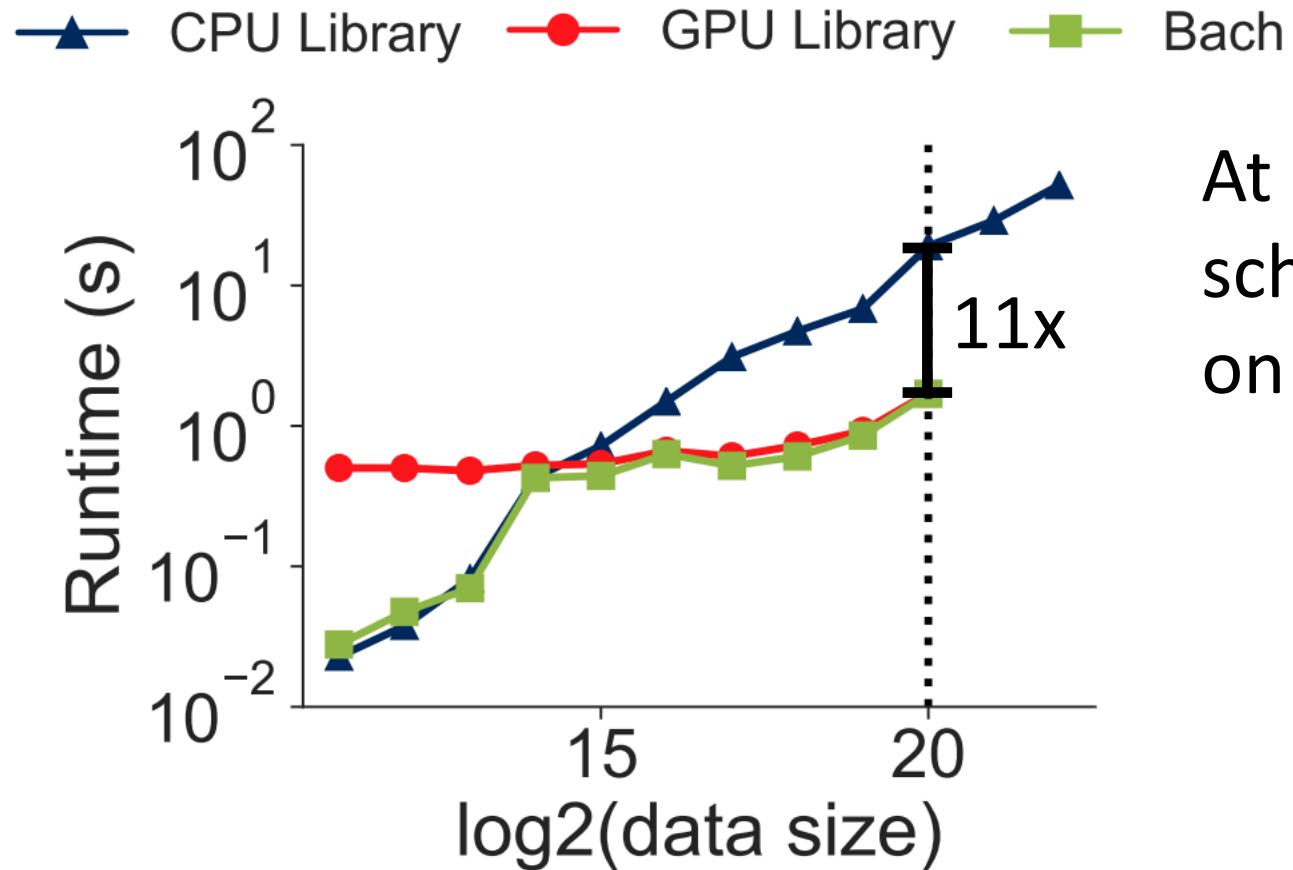
In-Depth Evaluation: Allocations



Crime Index saves time by eliminating the initial data transfer, while the allocation still fits in GPU memory.

(c) Crime Index.

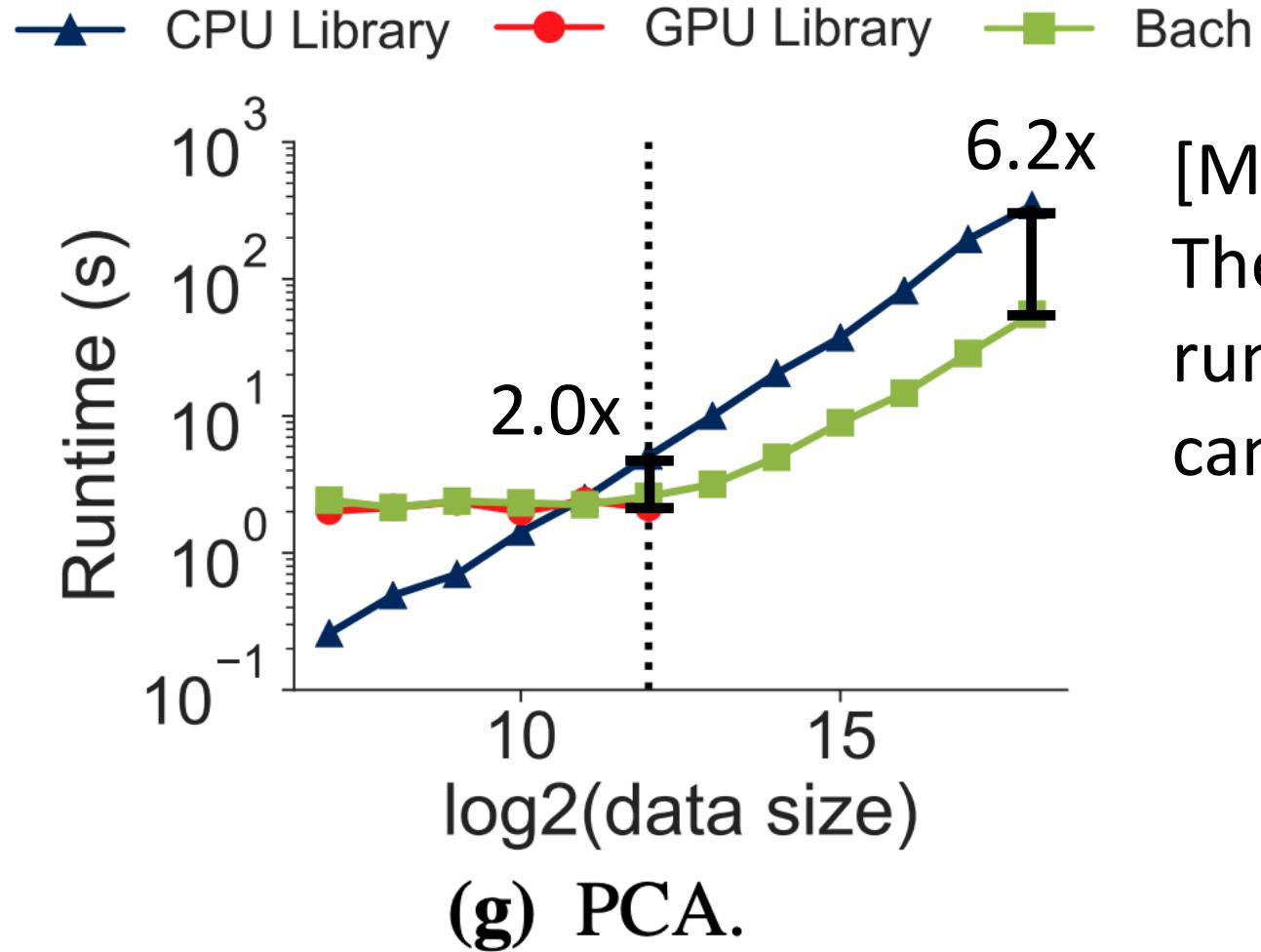
In-Depth Evaluation: Heuristics



At smaller data sizes, TSVD schedules all computation on the CPU.

(h) TSVD.

In-Depth Evaluation: Splitting/Paging Datasets

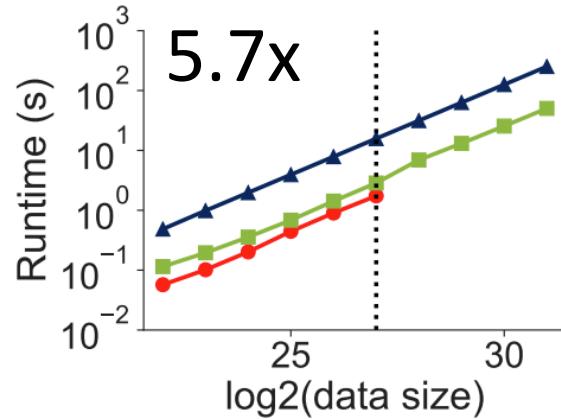


[Motivating Example]
The "fit" phase dominates the runtime until the "predict" phase can split/page data into the GPU.

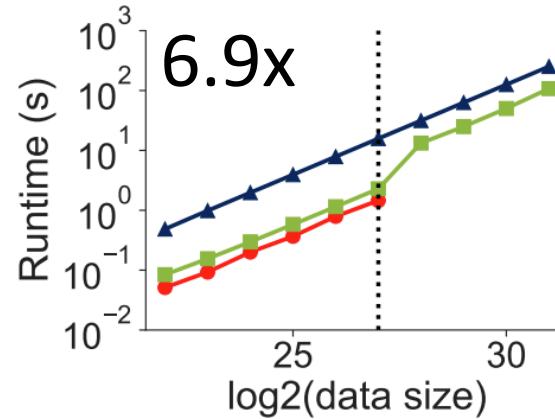
Evaluation: Summary

Max Speedup

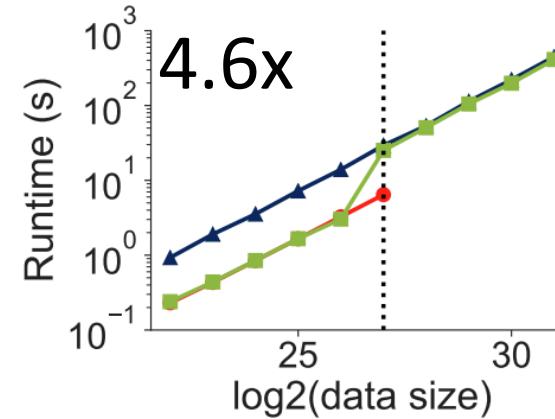
—▲— CPU Library —●— GPU Library —■— Bach



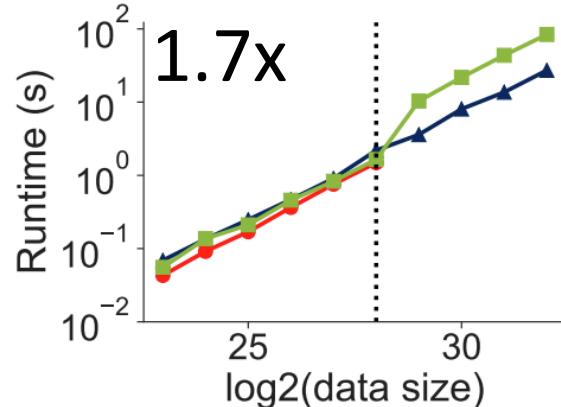
(a) Black-Scholes (CuPy).



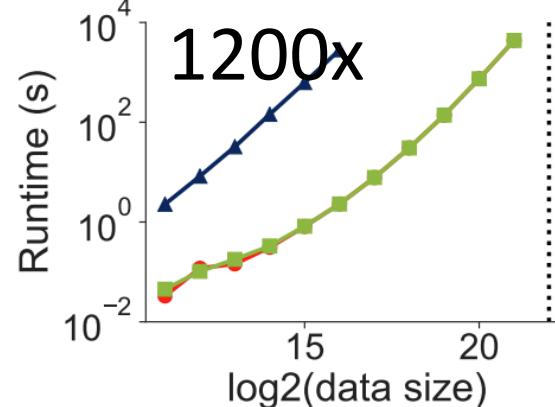
(b) Black-Scholes (Torch).



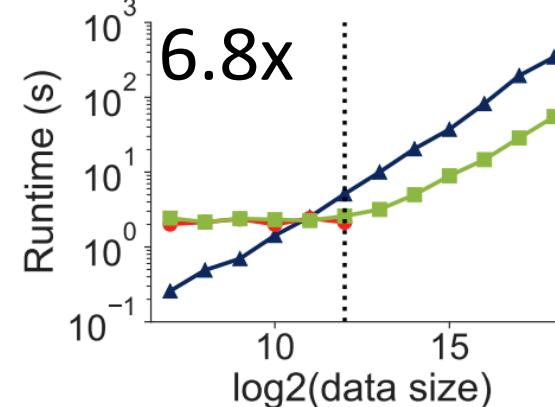
(c) Crime Index.



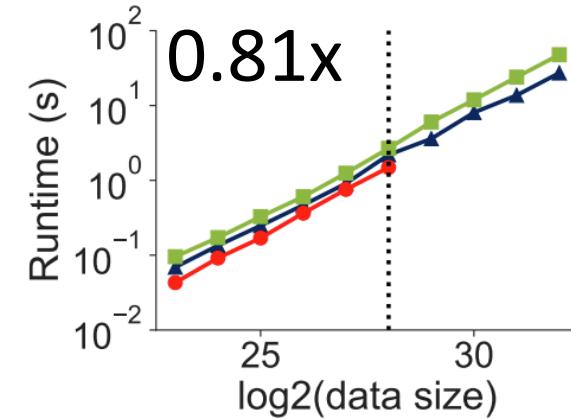
(e) Haversine (CuPy).



(f) DBSCAN.



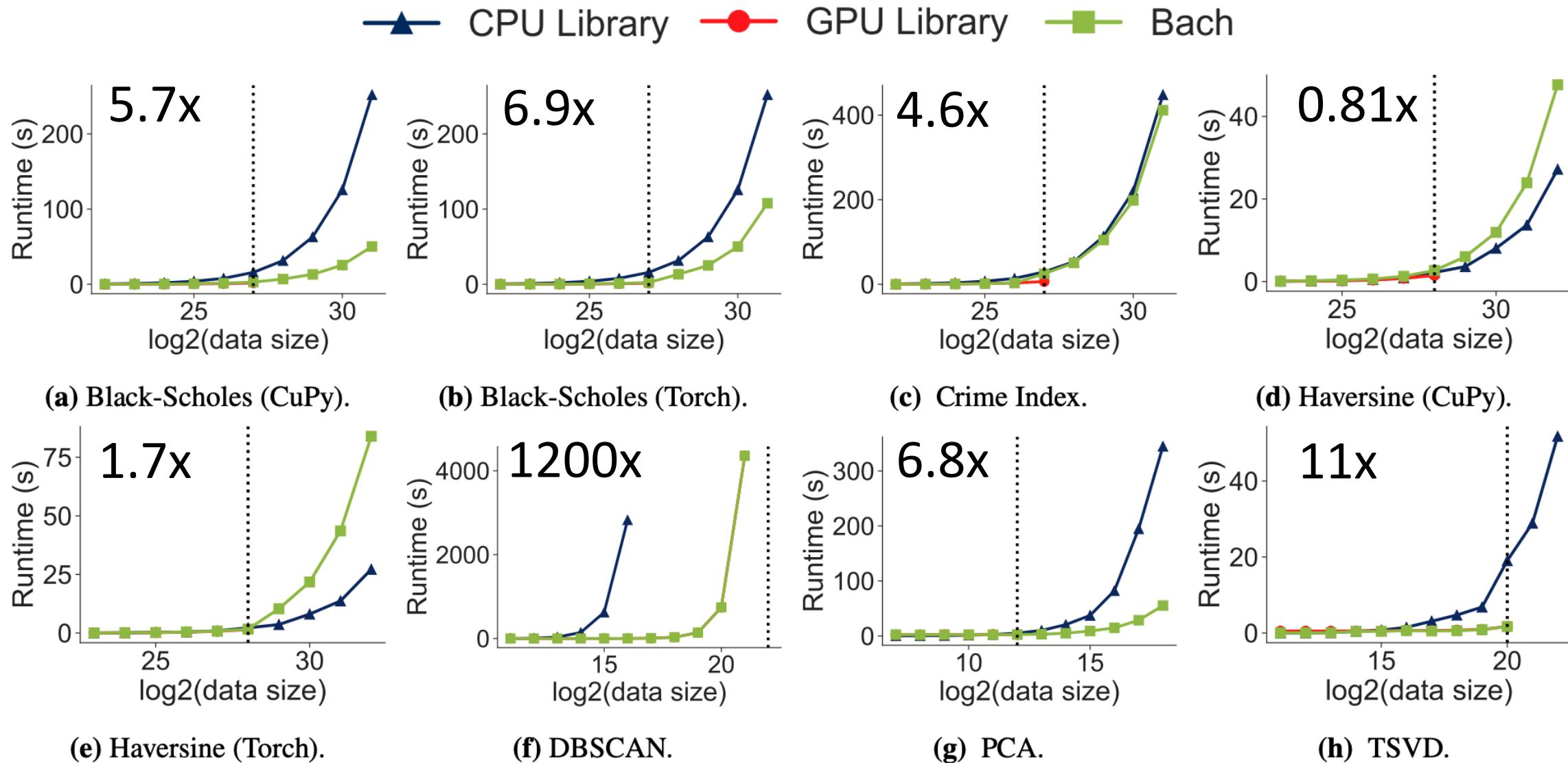
(g) PCA.



(h) TSVD.

Evaluation: Summary

Max Speedup



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



gyuan@cs.stanford.edu