Offload Annotations: Bringing Heterogeneous Computing to Existing Libraries and Workloads

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

Abstract

As specialized hardware accelerators such as GPUs become increasingly popular, developers are looking for ways to target these platforms with high-level APIs. One promising approach is *kernel libraries* such as PyTorch or cuML, which provide interfaces that mirror CPU-only counterparts such as NumPy or Scikit-Learn. Unfortunately, these libraries are hard to develop and to adopt incrementally: they only support a subset of their CPU equivalents, only work with datasets that fit in device memory, and require developers to reason about data placement and transfers manually. To address these shortcomings, we present a new approach called *offload annotations (OAs)* that enables heterogeneous GPU computing in existing workloads with few or no code modifications. An annotator annotates the types and functions in a CPU library with equivalent kernel library functions and provides an *offloading API* to specify how the inputs and outputs of the function can be partitioned into chunks that fit in device memory and transferred between devices. A runtime then maps existing CPU functions to equivalent GPU kernels and schedules execution, data transfers and paging. In data science workloads using CPU libraries such as NumPy and Pandas, OAs enable speedups of up to 1200× and a median speedup of 6.3× by transparently offloading functions to a GPU using existing kernel libraries. In many cases, OAs match the performance of handwritten heterogeneous implementations. Finally, OAs can automatically page data in these workloads to scale to datasets larger than GPU memory, which would need to be done manually with most current GPU libraries.

1 Introduction

The public cloud has commoditized specialized hardware such as GPUs, giving developers a new way to speed up their applications using these new accelerators. One increasingly popular way of doing this is to use accelerator-compatible *kernel libraries* with APIs that mirror those of popular CPU libraries. For example, in just the last two years, the Python ecosystem has seen a rapid explosion of popular GPU libraries such as PyTorch [24], cuML [8] and cuDF [7], and the RAPIDS [9] suite for data science; these libraries mirror the APIs of popular packages such as NumPy, Scikit-Learn, and Pandas. In keeping a familiar interface, accelerator libraries promise a seamless path to supporting new hardware for both new and existing applications.

Unfortunately, in reality, accelerator libraries for GPUs are not so simple to adopt into new or existing code. First, many of these libraries only implement a subset of functionality present in their CPU counterparts, e.g., because some functions are inefficient on parallel architectures such as GPUs [14, 15, 28]. This means that most applications must use *both* CPU and accelerator libraries, thus violating the promise of seamless integration with new hardware platforms. In addition, small API differences mean that even supported functions often warrant application changes. Finally, since most accelerator libraries operate over data that fits entirely in device memory, workloads in domains such as data science cannot seamlessly reap the benefits of new hardware because their working sets far outsize device memory. Developers must manually page and transfer data between the accelerator and CPU, or forego accelerators altogether.

In this paper, we propose *offload annotations (OAs)*, a new approach for incrementally integrating existing CPU libraries with emerging accelerator libraries. With this approach, an *annotator* (e.g., an application or library developer) adds annotations to CPU functions that specify a corresponding accelerator function from an accelerator library. An underlying runtime uses the annotations to automatically schedule functions either onto the CPU or the accelerator. In our system, we show that annotations enable end users to use both established CPU libraries and emerging GPU libraries without having to change their code or learn a new API. We also show that our runtime can allow end users to invoke GPU functions transparently even when data does not fit in accelerator memory. However, designing and leveraging annotations to offload computation to accelerators poses a unique set of challenges.

First, applications that mix CPU and accelerator code must be cognizant of *data placement*. For example, accelerator libraries such as PyTorch [24] can only process data resident in device memory, and GPU-resident data has an entirely different format than CPU-resident data. Our annotations explicitly keep track of the device on which a particular data value resides, and include a new API to let annotators specify how to transfer data between devices. In addition, some library functions that *allocate* new data, such as numpy.eye() (which creates an identity matrix), have equivalent functions in an accelerated library, so OAs let users explicitly identify such functions. The runtime uses this additional information specified in OAs to ensure that data is in the correct format on each device, helping to optimize the computation.

A second challenge in offloading functions to accelera-

tors is *memory management*. Accelerators generally have far smaller attached memories than CPUs: this means that large CPU-resident datasets cannot naively be copied to the accelerator in entirety. To address this challenge, OAs leverage the splitting mechanism of *split annotations* [23], originally used for cross function cache pipelining on the CPU, in the new use case of *paging* memory. Our runtime partitions inputs into chunks that fit in device memory, and automatically schedules data transfer and function invocation on partitioned values. Values are partitioned and merged using a user-defined *splitting API*. By identifying splittable functions using OAs, developers can run existing CPU workloads across different platforms transparently, even if the working set does not fit in device memory. Some accelerated functions available in libraries *cannot* be split into smaller computations, however, but OAs can still offload the computation up to a certain size.

Finally, a third challenge unique to offloading functions is determining where to execute annotated functions. Since functions that execute on an accelerator must first transfer their inputs to device memory, they have an additional associated data transfer cost. This can negatively impact performance in cases where the function itself executes quickly. To address this challenge, our runtime includes a new scheduler that uses estimates of transfer cost and compute cost to determine when functions should be executed on the GPU vs. the CPU. We show that simple linear cost model estimators can yield $26\times$ performance improvements compared to greedily executing all supported functions on a GPU.

The adoption of annotation-based approaches has already seen success in the past with systems such as TypeScript [12, 25]. In the TypeScript community, annotators crowdsource and share annotations for existing libraries. Unlike approaches that require building a complete end-to-end compiler, such as Weld [22] or Delite [31], the annotation-based approach also allows annotators to *incrementally* add support for individual accelerated functions. We hope to see a similar community develop around OAs to bridge existing CPU libraries with their accelerator library equivalents.

We implemented OAs by extending the Python runtime for split annotations, Mozart [23]. Our extended runtime, *Bach*, considers the challenges unique to offloading computation to capture device placement information and schedule computation in a heterogeneous setting. We evaluate OAs by integrating several CPU-only data processing libraries with their GPU library equivalents: PyTorch and cuPy for NumPy, cuDF for Pandas, and cuML for Scikit-learn. Our integration experience demonstrates the generality of OAs for popular data science libraries, and the minimal developer effort involved that requires little to no code modifications. On data science workloads ranging from options pricing to principal components analysis, OAs are able to achieve up to $1200\times$ improvement with a median $6.3\times$ over CPU-only code, with few or no application changes. OAs also enable many workloads to use GPUs when the dataset size exceeds the GPU

```
# Fit.
m1 = sklearn.StandardScaler()
m2 = sklearn.PCA() # or cuml.
m3 = sklearn.KNeighborsClassifier() # or cuml.
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 chunk in f:
  # 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)
```

Listing 1: Example data science workload using `sklearn`. Lines preceeded with a **+** show modifications required for using a GPU with the `cuML` accelerator library.

memory, which would require manually paging code with the existing GPU computation libraries.

In summary, we make the following contributions:

- We introduce offload annotations (OAs), an interface for heterogeneous computing with no library modifications that allows third-party annotators to incrementally add accelerated versions of library functions, and manages the offload and execution of these functions on devices. The OA interface extends split annotations with support for representing data in different formats on different devices and deciding when to offload a computation based on its estimated transfer size and computation cost.

- We describe Bach, a Python runtime that uses annotations to capture a lazy task graph of program operations and schedules execution and data transfer, including allocations, in order to improve application performance while staying within the accelerator's memory limits.

- We integrate OAs with four kernel libraries for GPU computation, and show that they can accelerate applications by a median of $6.3\times$ over the CPU-only version of the library. We also compare the performance of OAs to hand-written heterogeneous code that manually combines these libraries with CPU libraries.

2 Motivating Example

To motivate the OA approach, consider the following simple scenario: A data scientist has a machine learning workload originally written for the CPU using `sklearn`. She reads the

data from a file on disk, preprocesses the data, trains the model, and then tests it on a real dataset (Listing 1).

As her company sends her more and more data, the data science pipeline takes an order of magnitude longer to run. She learns about accelerators and accelerator libraries such as cuDF and cuML built to speed up data science workloads using GPUs. Since the pipeline was already running on cloud instances, the data scientist decides to move her code to another instance with accelerators attached.

The online documentation for these kernel libraries promises a seamless integration experience, offering almost exactly the same interface as their CPU library counterparts, but she soon discovers it is not as easy as it seems. Some of the functions have different names, requiring her to scour the documentation for functions with the corresponding functionality. Some functions do not have corresponding implementations at all, and can run only with the CPU library.

Next, the data scientist analyzes the program and manually inserts data transfer statements between the GPU and the host CPU so that data resides on the same device as the corresponding functions. Since the inputs to the program are usually read from another step in the pipeline, she assumes the inputs initially reside on the CPU. Since she initializes some of the intermediate objects herself, she allocates those directly on the GPU. Finally, since her plotting library requires the data to be on the CPU, she transfers the results back to the CPU at the end of the program.

With most CPU library functions replaced with the corresponding GPU library functions, she is ready to run the program, but it crashes due to insufficient GPU memory. Although her dataset was small enough to fit in CPU memory, the amount of memory attached to the GPU is significantly smaller. She writes additional code to page the data transfers to the GPU in the prediction phase, since the `predict` computation can be done independently on different data batches. The training function in the accelerator library *cannot* be run in independent batches, but fortunately, her training data is smaller than the data she predicts on, so that computation can run as one GPU function call.

Finally, after all the developer effort required to learn and integrate the GPU libraries, the code runs to completion, and the data scientist receives the same results on her dataset as when the pipeline ran only on the CPU. The performance of her new program has also improved. Nonetheless, the data scientist's code is now complex (Listing 1), with many new function calls and new control logic just to manage the GPU computation. Furthermore, all this new logic may need to change in the future as her workload changes or her dataset changes in size.

3 Design Overview

With *offload annotations* (or *OAs*, Figure 1), we reduce the developer effort for porting an existing workload to the GPU to just importing the annotated CPU library in place of the
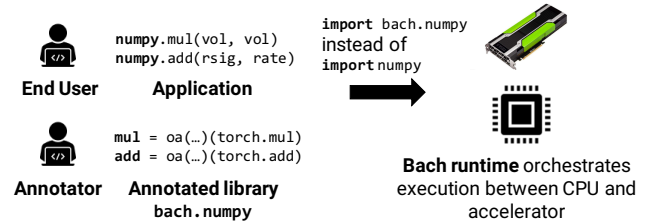

Figure 1: Overview of writing and using OAs. An *annotator* writes annotations to bridge a CPU library with an accelerator library. An *end user* imports the new annotated library to automatically use new accelerators in her existing code.

original CPU library. The annotator could be the kernel library developer, the end user writing applications, or any other third-party developer (similar to the open source contributors that provide type definition files for libraries in TypeScript [12]). Our system, Bach, uses the information in OAs to automatically offload functions to a GPU, page large datasets, transfer data across devices automatically, and manage allocations to minimize data transfer for better performance.

Adding OAs to CPU libraries. First, an annotator identifies a corresponding accelerator library for her CPU library (e.g., torch for numpy). She then identifies a corresponding accelerator function for each CPU function she wishes to annotate (e.g,. torch.multiply for numpy.mul). The annotator then writes an OA for her CPU function: the OA specifies the corresponding GPU function that should be called in place of the CPU function, and split types for each function input and output (adapted from the split types used in split annotations [23]). Split types are an interface implemented by an annotator that provide information about a function input or output *at runtime* (e.g., the size of an array). The annotator can also write special OAs for allocation functions (e.g., torch.zeros for numpy.zeros), which enable data to be allocated directly on the device where they will first be used.

OAs extend the original split type interface to provide additional information about *data placement*. In particular, the annotator must implement a new *offloading API* for each split type. Most libraries only require implementing the offloading API once per *data type* in the library (e.g., for ndarray in NumPy). The offloading API specifies (1) where inputs to a function reside before execution (e.g., in GPU memory or CPU memory), and (2) how to *transfer* values from from one device to another. Since the offloading API extends the split annotation splitting API, it can also *optionally* describe how to split and merge data for data-parallel workloads. Splitting in OAs is used for paging data into an accelerator with limited memory (unlike in split annotations, where splitting enables cross-function cache pipelining on a CPU).

Once an annotator adds OAs and implements the offloading API for the data types in the CPU library, she can share the annotation file to allow other end users to reap their benefits.

Using annotated libraries. An *end user* integrates the annotated library into her code by changing the line that imports the CPU library to import the annotated library instead (the annotation file is just a Python module).

Generally, little to no code modification is required to use the annotated library in place of the original CPU library. The main difference is that Bach, our runtime, uses *lazy evaluation* to optimize data transfer across many functions. OAs can automatically evaluate lazy values in many cases (e.g., when calling `str()` in Python), but an end user may have to add `evaluate()`—a function automatically provided in annotated libraries—into her code to explicitly materialize lazy values.

Bach runtime scheduler. Bach builds on split annotations' Mozart runtime to automatically build and maintain a lazy task graph. Internally, when a lazy value is materialized, Bach uses OAs to automatically schedule data transfers and computation, deciding the device on which to run each operation (§5). Note that scheduling is completely packaged in our runtime, and does not require any additional annotator or end user code.

Although the Bach runtime defaults to greedily scheduling operations on the GPU, the annotator may still provide custom cost model estimators to the function annotations to assist the runtime with making scheduling decisions. However, these cost models are optional and are not usually required to benefit from OAs, as shown in our evaluation.

4 Offload Annotation Interface

The offload annotation (OA) interface provides a corresponding accelerator function for each function in a CPU library. The interface also provides a mechanism to discover runtime information about function arguments and outputs: namely, how to split inputs into chunks that will fit in device memory, the device on which an input is allocated before executing annotated functions, and how inputs can be transferred between devices. This information is relayed via *split types*, an abstraction from split annotations [23]. The OA interface also includes new *alloc annotations*, which specify functions that allocate new data (e.g., `numpy.zeros` to allocate an array of zeros). These annotations allow further optimizations when scheduling data transfer and are described further below.

Listing 2 shows an example of the extended offload split type API, and Listing 3 show examples of OAs, with `numpy` as the CPU library and `torch` as the accelerator library.

4.1 Primer: Split Types

Split types provide a mechanism that allow a runtime to discover runtime properties about a value (e.g., size of an array or dimensions of a matrix). In split annotations, split types are used to ensure that data is *split* in a consistent way across functions to enable safe pipelining of split values. For example, a split type will ensure that split arrays passed into a function together still have the same lengths at runtime.

```python
class DataFrameSplit(OffloadSpliType):
  def split(start, end, value):
    # Splits a value to enable paging.
    return value[start:end]
  def merge(values):
    # Merges split values
    return pandas.concat(values)
  def size(value):
    # Returns number of elements in value
    return len(value)
  def device(value):
    # Specifies where this value is allocated.
    # Used by scheduler to decide where to run
    # functions.
    if isinstance(value, pandas.DataFrame):
      return Device.CPU
    else: # if a cuDF DataFrame.
      return Device.GPU
  def to(value, device):
    # Transfers [value] to specified [device].
    if device == Device.GPU:
      return cudf.from_pandas(value)
    else:
      return value.to_pandas()
```

Listing 2: Example implementation of the offload split type API for Pandas and cuDF DataFrames. The API adds two new functions—`device` and `to`—to the original split type API.

To use split types, an annotator implements an API that the runtime calls to interact with runtime values. In split annotations, split types provide a `split` function to partition values into chunks, and a `merge` value to merge split values together. The API also contains a `size` function for discovering the size of inputs (e.g., to determine split sizes). Listing 2 shows an example of these functions for DataFrames. We extend the split type API to allow our runtime to offload values onto other devices.

When splitting and merging data in a non-trivial way, end users must ensure the correctness of the application. Many data science applications operating on large collections of data, as in our workloads, are trivially parallelizable. Even when splitting and merging is impossible due to algorithmic correctness constraints or unavailable kernel library implementations, applications can still benefit from automatic offloading at smaller data sizes.

4.2 Offload Split Types

In addition to specifying how data should be split and merged, our extended *offload split types* additionally specify *(1)* the device on which a value is allocated, and *(2)* how to transfer a value between devices.

Device API. The `device` method specifies the device its input resides on. The method might check the instance type

```
# Offload split types for binary function inputs.
args = (NdArraySplit(), NdArraySplit())
# Offload split type of return value.
ret = NdArraySplit()

# OAs to provide offload split types for each
# argument and return value, as well as a corresonding
# accelerator function.
np.add = @oa(args, ret, func=torch.add)
np.subtract = @oa(args, ret, func=torch.sub)

# Allocation function.
np.empty = @oa_alloc(NdArraySplit(), func=torch.empty)
```

Listing 3: Offload annotations using `numpy` and `torch`.

of the input, or properties of the input. For example, NumPy arrays are on the CPU while CuPy arrays are on the GPU. Torch tensors can reside on either device, and have a property to describe where a particular value resides.

To API. The `to` method transfers the provided value to the target device. This usually involves converting a CPU library type (e.g., `numpy` array) to an accelerator library type (e.g., `torch` tensor) using a accelerator library function (e.g., `torch.to()`). The ability to transfer values is necessary to ensure that values reside on the device where the operation will run.

4.3 Using Offload Split Types in Annotations

OAs assign each input and output an offload split type. Additionally, the OA provides the name of the corresponding accelerator library function. If the accelerator function has a different function signature, an annotator can wrap the accelerator function in a lambda with an interface consistent with the CPU function.

Example. Listing 3 shows several examples where NumPy functions are annotated using PyTorch. The OAs assign the two arguments of `np.add` and `np.subtract` the offload split type **NdArraySplit**. This split type will define how to split, merge, and transfer `ndarray` and `torch.tensor` values. It will also tell Bach whether a particular value passed to these functions is already on the accelerator or CPU using the `device` API. The outputs of these functions also have the offload split type **NdArraySplit**.

4.4 Allocation Function Annotations

One unique challenge the runtime faces is avoiding unnecessary data transfers, which can lead to performance degradation. Consider when data is allocated on one device, and then immediately passed to a function that can be offloaded. In this case, it is more efficient to allocate the data directly on the device of the following function.

To support this, OAs provide a special kind of annotation, called an `alloc` annotation, which specify that the annotated function performs allocation. Like other annotated functions, annotators provide CPU allocation functions with an equivalent accelerator library functions (e.g., `torch.zeros` for `numpy.zeros`). Allocation functions differ from regular functions in one key aspect: their inputs do not need to be annotated with offload split types. Outputs are still annotated with a offload split type, similar to a regular function. CPU-only split annotations did not require allocation annotations since they did not need to avoid expensive data transfers.

Example. In Listing 3, the `np.empty` function allocates data: its OA specifies a corresponding PyTorch function, but does not provide offload split types for inputs. Its output has the same **NdArraySplit** offload split type.

5 Bach Runtime

The Bach runtime uses the information in the OAs to schedule and execute functions across the CPU and accelerator. Figure 2 provides an overview of the Bach runtime.

Step 1: Construct Dataflow Graph. Bach's first goal is to extract a dataflow graph from the user program. To do this, Bach uses the same approach as Mozart: when annotators apply an annotation to a function, Bach wraps it to return a lazily evaluated placeholder object, using Python's metaprogramming facilities [23]. When placeholders objects are passed to other annotated functions, Bach stitches these functions into a task graph. This task graph captures dependencies between annotated functions: an edge between two operations exists if the output of one operation is used as an input into another.

Two scenarios trigger evaluation of the task graph. First, Bach detects accesses to the lazy placeholder objects by intercepting certain Python methods (e.g., `str` to convert an object into a string, or `__getitem__` to index into a collection). Internally, the placeholder object will trigger evaluation of the full task graph required to build it, and then forward these method invocations to the evaluated object. For example, this means that placeholder objects will be evaluated if the user passes them to `print()`. Alternatively, the user can also explicitly call an `evaluate()` function to trigger execution.

Step 2: Estimate Data Size and Allocate. In order to correctly partition the data for splitting, Bach must estimate the data size by using the `size()` API on the program inputs. Unlike in Mozart, Bach can lazily allocate values to optimally place data on the same device as the first function that uses that value. However, if all the program inputs are lazy allocations, there are no available values with which to estimate data size. Thus Bach must allocate lazy values before starting execution to estimate the data size.

Bach decides where to allocate each lazy value based on the device of the first function to use the value. OAs differ from split annotations in this regard because they need to decide which device to run each function on. A function can
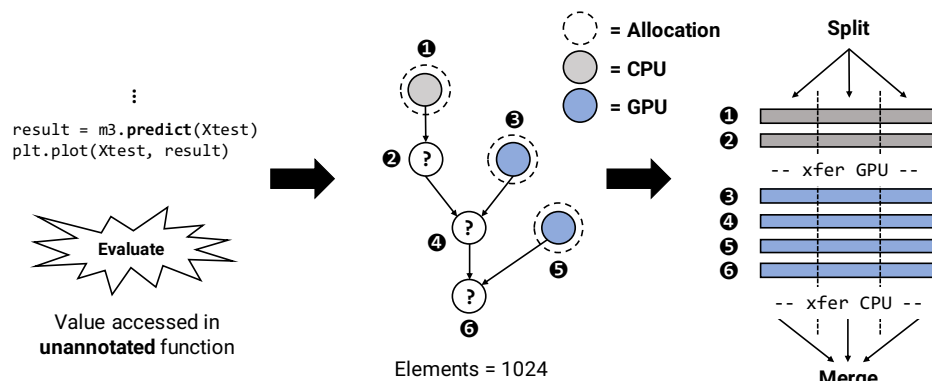
Figure 2: Overview of the steps involved in Bach's runtime. Step 1 triggers evaluation of the dataflow graph with an access to a lazy value. Step 2 allocates the lazily allocated nodes and estimates the program data size. Step 3 dynamically schedules the instructions across devices, inserting data transfers and paging the inputs.

```python
# Heuristic for estimating data transfer cost.
def transfer_estimator(ty, values, device):
    x = ty.size(values)
    return a * x + b

# Heuristic for estimating compute cost.
def compute_estimator(ty, values, device):
    x = ty.size(values[0])
    if device == CPU:
        return a_cpu * x + b_cpu
    else:
        return a_gpu * x + b_gpu
```

Listing 4: Linear estimators for estimating data transfer and compute cost.

run on the accelerator if it has an `oa` or `oa_gpu` annotation, and if its inputs either can be transferred to the device or already reside on the accelerator. All functions must be able to run on the CPU, which is the default device.

To decide where to run this first function, Bach estimates the data transfer costs and compute costs involved with running the function on either device and suggests the device with the lower cost. These cost estimates are calculated using heuristic functions optionally provided by the user. The heuristic functions are functions of the input values, their offload split types, and the target device, and we provide a simple linear cost model estimator (Listing 4). If cost models are not provided or all other inputs are also lazy allocations, Bach naively suggests the function run on the accelerator if possible. Otherwise Bach defaults to the CPU.

Step 3: Schedule and Execute. Once the data size is estimated, the operations in the task graph are converted into a list of instructions that can be executed serially for each split piece. To do this, the Bach runtime performs a topological

sort of the task graph to obtain a list of instructions that satisfy data dependencies.

The device an instruction runs on is decided dynamically right before executing the instruction. The instruction first determines if it is eligible to run on the accelerator based on the requirements described in the previous section. If it is not eligible, it defaults to running on the CPU. Otherwise, the runtime performs the same cost model analysis as when deciding where to lazily allocate a value to determine which device to run the instruction on. Unlike before, all the inputs will be fully evaluated. After deciding which device to use, the instruction transfers inputs that are on a different device using the `to()` API in the offload split type for the input. Bach discovers where inputs are prior to executing functions by using the `device()` API. As before, if cost models are not provided, Bach defaults to using the accelerator if possible.

When executing functions, Bach uses the ability to partition data to enable *paging*: this allows for large, CPU-memory-resident datasets to be streamed through device memory, even when device memory is far smaller than the CPU memory. To achieve this, Bach splits the inputs into chunks based on the estimated data size and a default piece size. Inputs are split using the `split()` API provided in the input's offload split type. For each chunk, the program executes the generated list of instructions. The chunk is transferred to the device of the input argument if its current device does not match the device of the instruction. Each chunk is moved out of the device after the functions finish executing, to free space for the next chunk.

The final outputs are moved to the CPU by default after execution, but the end user can elect to keep the output on the accelerator by explicitly calling `evaluate()`. If paging is used to stream data through a device, the output is always allocated on the CPU (since it may not fit entirely in device memory).

6 Design Discussion

As described, OAs and the Bach runtime are designed specifically for offloading computation to a single GPU using Python kernel libraries. In this section, we discuss the possibilities of extending OAs for use with multiple GPUs or with other programming languages and accelerators.

Multiple GPUs. Similar to how split annotations split data-parallel workloads across CPU cores, we can extend OAs to automatically split computation across multiple GPUs. The implementation would need to modify the scheduler to recognize multiple GPU targets, and factor data transfer and concurrency into scheduling decisions. We do not expect these modifications to impact end user experience.

Non-Python programming languages. We chose to implement Bach in Python since Python is one of the most popular languages for data analysis, with a vast ecosystem of Python GPU libraries. We believe our implementation uses principles common to many programming languages and do not rely on any language-specific hacks. Specifically, "annotations" in Python are simply functions that wrap other functions, and the runtime logic is language-agnostic. Mozart [23] demonstrates that the annotation framework is viable in C++, so we imagine we could implement a similar runtime for C++ for GPU libraries like Thrust.

Non-GPU accelerators. Any accelerator with a kernel library that closely mirrors a CPU-only library and an API to offload data to that accelerator could potentially be suitable for OAs. We may also be able to adapt split data in streaming accelerators to overlap data transfer with computation like in CUDA streams. Data transfer costs are an issue common to many accelerators, and we could apply ideas about memory management and data placement from OAs even if the system cannot be used directly.

7 Library Integrations

We annotated three different CPU-only Python libraries for data science and machine learning: NumPy, Pandas, and Scikit-learn. The annotations used four different GPU kernel libraries: CuPy, PyTorch, cuDF, and cuML. The latter two kernel libraries are part of the RAPIDS [9] suite.

NumPy. NumPy is a library for high-level math operations on multi-dimensional arrays and matrices on the CPU. NumPy was the most popular library in terms of number of accelerator library equivalents. In their online documentation, these accelerator libraries directly claim to provide a NumPy-like API. CuPy is described as a NumPy-like API accelerated with CUDA, while PyTorch is described as a replacement for NumPy that leverages the power of GPUs. We integrate NumPy with CuPy and PyTorch in two separate OA-libraries, replacing NumPy `ndarrays` with CuPy `ndarrays` and PyTorch `tensors`.

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

Table 1: Integration effort for annotating various libraries. Lines of code include annotations, split type transfer functions, and splitting functions.

Pandas. Pandas is a data analytics library for operating on structured table-like or time series data. The cuDF accelerator library provides a Pandas-like API. We replace Pandas `DataFrame` and `Series` data types with the corresponding cuDF types.

Scikit-learn. Scikit-learn is a popular machine learning library. Machine learning is a natural fit for the GPU with its dense linear algebra operators. The cuML library's Python API attempts to closely match the Scikit-learn API.

7.1 Integration Experience

From our experience, library integrations required around 130 lines of code per library (Figure 1). The most lines of code come from implementing the offload split type API for transferring, splitting, and combining types. However, the split and combine API is optional if a user does not need to page large datasets. In the simplest and most common case, an OA requires only a single line of code per function to specify the offload split types of inputs and outputs, and the name of the kernel library function. These annotations resemble boilerplate code when libraries repeat a common pattern, like binary operations with a single output in NumPy. In more complex function annotations, the benefit of OAs is that it only needs to be done once in the annotated library as opposed to every instance in every workload.

7.1.1 Straightforward Drop-In Replacements

Every library has a straightforward way to transfer data to the appropriate device. Most accelerator library types automatically reside on the GPU, so using CuPy's `ndarray` or cuDF's `DataFrame` automatically transfers the data to the GPU. Scikit-learn can use either CuPy's `ndarray` or cuDF's `DataFrame` as the underlying representation. PyTorch tensors can reside on both CPUs and GPUs, so the transfer implementation from Numpy `ndarray` first converts the `ndarray` to a `torch.Tensor` (a zero-copy cast) and then calls a method on the tensor to transfer it to the GPU.

Just as many accelerator libraries claim to closely resemble the CPU libraries they attempt to replace, many accelerator library functions are indeed a drop-in replacement for the corresponding CPU library function. For ex-

ample, `numpy.add()`, `cupy.add()`, and `torch.add()` are the same across all three libraries. Sometimes, the method names are trivially different but represent the same functionality, like `numpy.arcsin()` and `torch.asin()`. Scikit-learn's API utilizes a complex module structure that does not exist in cuML (e.g., `sklearn.decomposition.PCA` vs `cuml.PCA`), so annotators sometimes must mock module structure to make integration as seamless as possible.

7.1.2 Different Function Specifications

Even if two CPU and kernel library functions appear to be equivalent based on name, the annotator must be careful to ensure the function parameters and specifications are identical. For example, the array allocation functions `numpy.ones()` and `torch.ones()` both take a parameter `dtype` to specify the data type of the array. However, NumPy can accept strings like 'int8' as a parameter, while PyTorch only accepts library-defined types like `torch.int8`. In this case, the annotator must write a custom wrapper that converts function parameters.

We experienced another challenge involving different function specifications when integrating Pandas with cuDF. Both `pandas.read_csv()` and `cudf.read_csv()` read a CSV into a DataFrame object. In Pandas, the `squeeze` parameter causes the function to return a Series if the parsed data only contained one column. To achieve the same functionality in cuDF, which does not have this parameter, we wrote a custom function that converted the returned DataFrame into a Series if the squeeze parameter was included. Of the 48 parameters in v0.25.2 of `pandas.read_csv()`'s documentation, others are also bound to not exactly match the specifications in cuDF and require special implementation.

7.1.3 Missing Functions

When a function is missing from an accelerator library, any library annotator can annotate the library with a custom function. We implemented a custom GPU version of the Pandas `mask()` function, used for replacing values in a DataFrame based on a conditional DataFrame, by using the `series.loc[cond] = val` notation from the cuDF library instead. In our cuML annotations, we mimicked Scikit-learn's `StandardScaler()` model using CuPy operations for removing the mean and scaling to unit variance on the GPU. In general, cuML's preprocessing library is far behind Scikit-learn's, potentially because Scikit-learn's functionality is easy enough to achieve with other accelerator library functions.

In other cases, functions do not exist because the algorithm required to provide the functionality is either impossible or simply unsuitable for the GPU. In particular, many Pandas functions do not work in cuDF when string operations are involved, requiring the program to execute on the CPU. Though libraries like `nvStrings` and `cuStrings` are working to close the gap in text processing on the GPU, the state of strings on the GPU today still requires the developer to have a deeper understanding of its literal representation on the GPU.

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×

Table 2: The evaluated workloads, the number of annotated function operators, and the CPU Python libraries used by each workload. The median speedup across workloads is 6.3× with a maximum speedup of 1200× on DBSCAN. Annotated with CuPy[1]. Annotated with PyTorch[2].

7.1.4 Multi-Library Integration

Just as libraries in the Python data science ecosystem use each other in their implementations, annotated libraries must also be able to import and operate on other annotated libraries. In the CPU ecosystem, Scikit-learn's functions use the NumPy `ndarray`, while in the GPU ecosystem, cuML's functions use the CuPy `ndarray`. In our annotated Scikit-learn library, we analogously import the `NdArraySplit` type from our annotated NumPy library to define the argument and return types in the OAs. As OAs grow in popularity, we imagine an ecosystem of increasingly-interconnected annotated libraries that allow seamless execution across multiple devices in existing workloads.

8 Evaluation

We ran experiments on a 56-CPU server (2× Intel E5-2690 v4) with 512GB of memory, running Linux 4.4.0. The machine has a single NVIDIA Tesla P100 GPU with 16GB of memory and CUDA 10.2 installed. Each result is the median of five runs, with one warm-up run omitted to initialize the CUDA driver. Workload runtimes are measured end-to-end, including allocation and synchronization operations at the end. Our source code is available at https://github.com/stanford-futuredata/offload-annotations.

8.1 Workloads

We evaluated offload annotations on a variety of workloads adapted from common mathematical formulas, data science library tutorials, and popular online blog posts (Table 2).

Black-Scholes [1]. Determines the theoretical value for a large array of call or put options.

Haversine [3]. Determines the great-circle distance between points on a sphere.

Crime index [4]. Reads population and robbery data from a file on disk and calculates a numerical crime index score.

DBSCAN [5]. Standardizes a dataset with 256 features around 32 centers, and clusters the data with the DBSCAN algorithm, analyzing the predicted labels on the CPU.

PCA [6]. Standardizes training data and reduces the dimensionality with PCA, then classifies the points with K-Neighbors. Predicts the results, followed by plotting.

TSVD [10]. Applies the linear dimensionality reduction algorithm to a synthetic dataset with 512 features.

8.2 Results

Figure 3 showcases the performance of the data science and machine learning workloads on inputs of various sizes. We evaluate the workloads on a CPU-only implementation, a handwritten accelerator kernel implementation, and an annotated implementation with Bach.

We verified the numerical results of the workloads for correctness against each implementation. In the machine learning workloads, we selected parameters that produced reasonable results given the inputs (e.g., DBSCAN predicted 32 clusters and PCA classified points with greater than 90% accuracy). We maintained the same parameters for each implementation to ensure the parameters did not affect the runtime.

The results show that with less developer effort, Bach is able to match the performance of handwritten GPU implementations, scale to larger input data sizes that normally cause the GPU to run out of memory, and outperform CPU library implementations. We now discuss these results in more detail.

8.2.1 Results Summary

We make three main observations.

First, Bach matches the performance of handwritten GPU implementations in all workloads. Bach selects which CPU library functions to offload and how to transfer data, with minimal developer effort.

Second, Bach scales to larger input data sizes that normally cause the handwritten GPU implementations to run out of memory. In most workloads, Bach's performance continues to scale at the same rate as before the GPU implementation runs out of memory, giving the appearance of running the same program on a GPU with infinite memory. DBSCAN is the only workload that cannot be split since the clustering model must be fit on all data at once. However, this is not a significant limitation since DBSCAN also has the largest runtime, with Bach taking 7.5 hours to run at the largest piece size before running out of memory, and the CPU implementation taking considerably more for the same input size.

Third, Bach outperforms CPU implementations for almost all workloads. One exception is Haversine Torch, which has worse performance than the CPU implementation when paging large datasets due to the overhead from the additional data transfers. Another exception is PCA at data sizes below 2^{12}. We attribute this to overheads associated with initialization and launching GPU kernels with significant launch overhead.

CPU implementations outperform Bach for other workloads as well, but only for small input sizes when runtimes are in milliseconds or less.

8.2.2 Runtime Distributions

Workloads that benefit from GPU acceleration are suitable for acceleration using OAs. At a high level, the decision to use the GPU is a tradeoff between expensive data transfers and faster compute. The impact of OAs, and more broadly the impact of the GPU, depends on the distribution of runtime between data transfer, computation, and memory allocation in each workload (Fig. 4).

Allocation. Crime Index spends 93% of its total runtime of 29.74s on allocating data. In particular, the workload reads 1GB of data into memory before performing a series of fast numerical operations and calculating a single number, the crime index, as an output. Managing memory allocations are particularly important for expensive I/O operations like reading files from disk. Workloads with these kinds of operations particularly benefit from lazy allocation.

Though allocation can be fast on either device, the primary performance benefit of lazy allocation is eliminating the additional data transfer required to move the input to the appropriate device. At large data sizes that do not fit in GPU memory, the initial memory must instead be allocated on the CPU where it does fit and paged into GPU memory using data transfers. As shown by the dotted lines in Fig. 3, beyond a specific data size, Crime Index, Haversine Torch, and Black-Scholes Torch all have additional overhead when paging large data sets due to the extra initial data transfer.

When the Crime Index (Fig. 3c) dataset fits in GPU memory and the workload is able to use lazy allocation, an additional million rows of data increases the runtime by only 50 ms, compared to 190 ms when the dataset does not fit in GPU memory. The dataset size exceeds GPU memory beyond $2^{27} \approx 100$ million rows. We calculated these overheads as an average of the scaled Bach implementation runtimes for log data sizes 21-26 and 27-31, respectively. We do not consider data sizes below 2^{21} to discount the small absolute scheduler overheads that are independent of data size.

Data Transfers. Haversine CuPy and Black-Scholes CuPy spend 52% and 60% of their total runtimes of 2.85s and 2.60s on data transfers (Fig. 4). Both workloads apply a sequence of fast numerical operations on large arrays initialized on the CPU, which must be transferred to the GPU. The workloads output more large arrays which must be transferred back. However, Black-Scholes still beats the CPU library implementation with Bach at all sizes, meaning larger absolute performance gains at larger input sizes.

As the less computationally-intensive workload, Haversine is more significantly affected by additional data transfers when paging large datasets, no longer beating the CPU implementation at large dataset sizes. It should be noted that in
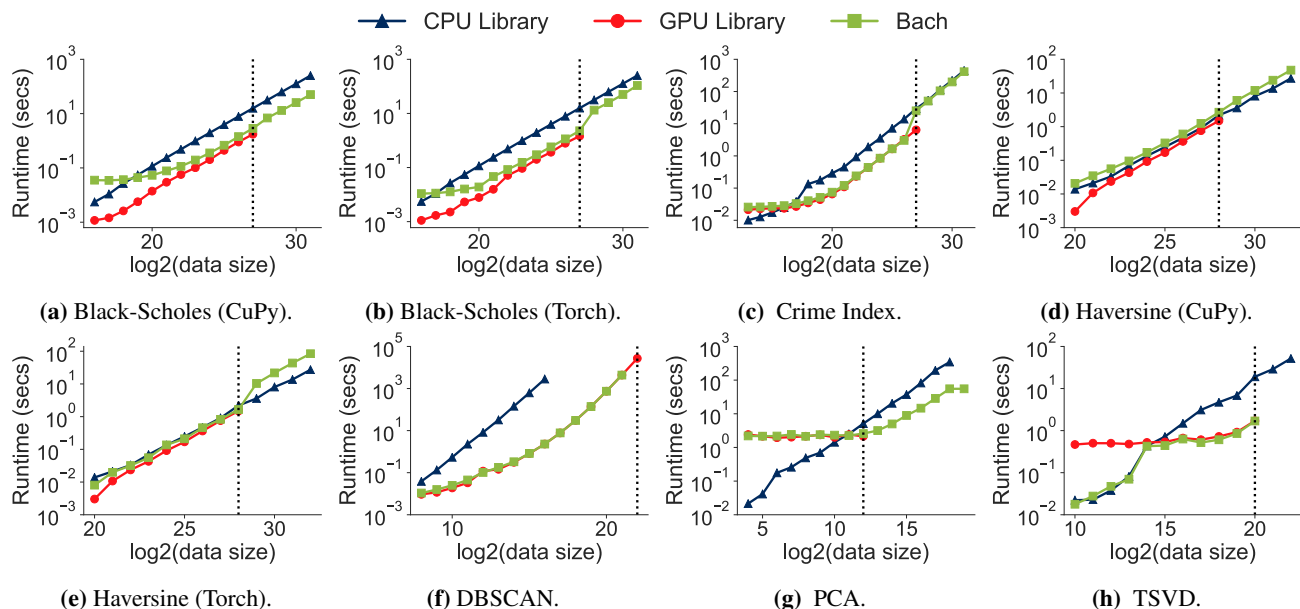
Figure 3: Runtime vs. input data size for the workloads in Table 2. Bach is able to match the performance of the corresponding GPU library for most applications. The framework in parentheses refers to the GPU library used by Bach in the workload. The dotted line represents the input data size at which the GPU runs into an out-of-memory exception; Bach is able to run workloads past this size by paging chunks in and out of GPU memory. (Log2 piece size = 27, 27, 21, 21, 26, 22, 12, 20 from (a-h))
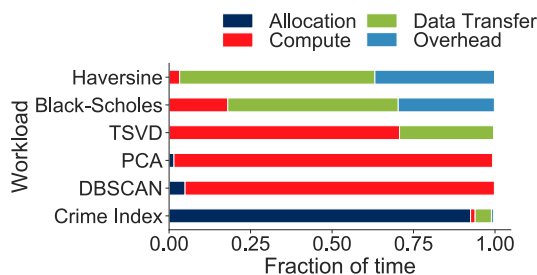

Figure 4: Proportion of total runtime split between memory allocation, data transfer, compute time, and runtime overhead in various workloads using Bach. Haversine and Black-Scholes mostly consist of data transfers; PCA, DBSCAN, and TSVD are compute-heavy; Crime Index spends the most time on allocation. (Log2 data size = 27, 28, 27, 19, 12, from top to bottom.)

absolute terms, the total time spent on data transfers for these workloads is relatively small. However, it is still important to optimize wherever possible, since the overheads can be exacerbated at scale or the applications using these pipelines might require real-time results.

Computation. DBSCAN and PCA, on the other end of the spectrum, are computationally-intensive workloads that are

highly optimized for the GPU. In DBSCAN, data transfer took less than 1% of the total runtime of 130.81s, compared to the 94% spent on compute (Fig. 4). PCA spent 95% of the total runtime of 1.02s on compute. Machine learning models, with their parallelizable and computationally-intensive numerical operations, are particularly suited for the GPU.

8.2.3 Scheduling

The dynamic scheduling algorithm has minimal effect when all program inputs can be lazily allocated. In this case, all program inputs start out on the GPU if possible, even if the data size is small. Even though execution would have been been faster exclusively on the CPU, it is no longer worth transferring the inputs back to the CPU due to the overheads involved. This occurs in all workloads except the machine learning workloads: DBSCAN, PCA, and TSVD.

Among the machine learning workloads, we provided cost estimators to the TSVD workload to enable dynamic scheduling. DBSCAN already optimally executes the entire program on the GPU for all data sizes, and would not benefit from dynamic scheduling. We did not evaluate the dynamic scheduler on PCA, though it exhibits a similar runtime profile to TSVD.

In the TSVD workload, we used the linear estimators in Listing 4 as heuristics for transferring the input `ndarray` and computing the model's `fit()` function. In the transfer estimator, we use parameters `a = 1` and `b = 0` to indicate that data

transfer is proportional to the size of the data regardless of device. In the compute estimator, we selected parameters based on the equilibrium point between and the CPU and kernel library lines in Figure 3h. We use `a_cpu = 2` and `b_cpu = 0` to indicate that on the CPU, computation is highly correlated to input size. For the GPU, we use `a_gpu = 0` and `b_gpu=14` to indicate that the computation is extremely cheap but incurs kernel launch overheads.

The result of using these linear estimators is a threshold scheduling algorithm that causes the total runtime of TSVD to be the minimum of the CPU library implementation and GPU library implementation (Fig. 3h). The scheduler switches from greedy GPU scheduling to CPU-only scheduling below the equilibrium data size 2^{14}. Below this data size, the GPU implementation remains constant at around 500 ms, while the CPU and Bach implementations become as low as 20 ms for data size 2^{10}. Thus Bach improves the TSVD runtime by up to 480 ms with the dynamic scheduler estimators, as it otherwise would have defaulted to using the kernel library.

It should be noted that in these workloads, the dynamic scheduler only generates an advantage at smaller data sizes where the overhead of GPU initialization is more pronounced. CPU libraries also tend to benefit from cache performance cliffs, which is why the CPU runtimes are not perfectly linear at smaller data sizes. In these cases however, the absolute runtime improvements from using the dynamic scheduler are small. In general, workloads do not require the annotator to implement any cost model at all to benefit from OAs especially at larger data sizes, where the execution is more likely to perform better on the GPU with the greedy scheduler.

8.2.4 Discussion

Our experience integrating multiple kernel libraries with their CPU library equivalents gave us several interesting insights into the existing Python ecosystem for GPUs, including the lack of GPU kernels for several CPU functions and the differences in seemingly identical kernel library implementations.

Missing GPU implementations. Black-Scholes CuPy and PCA both contained functions without kernel library equivalents, the `numpy.erf()` error function and `sklearn.StandardScaler()`, respectively, despite these functions being trivial to run on the GPU. We addressed these performance issues in different ways.

In Black-Scholes, we wrote a custom GPU function to annotate `numpy.erf()` and eliminate additional data transfers that disrupted a numerical analysis pipeline that otherwise ran completely on a GPU. We copied the implementation from a file in CuPy's experimental folder for SciPy routines. This example demonstrates that regardless of whether the kernel library developer or third-party annotator contributes to an annotated library, annotations can make it easier to incrementally add support for GPUs into an existing workload.

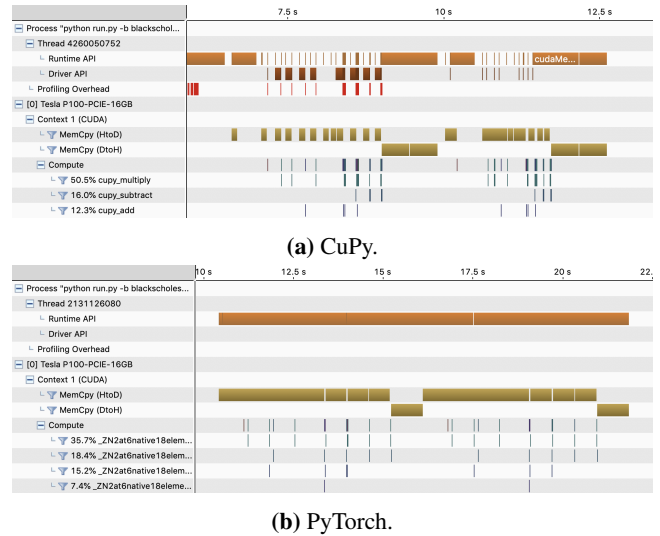Though we could have implemented a custom GPU function for `sklearn.StandardScaler()`, we did not do so because

(a) CuPy.

(b) PyTorch.

Figure 5: The NVIDIA Visual Profiler visualization of Black-Scholes annotated with CuPy and PyTorch, when paging large datasets into GPU memory. (Data size = 2^{28}; Piece size = 2^{27})

the compute time for this numerical preprocessing step was small in comparison to the prediction part of the workload, and the performance impact would not have been significant.

CuPy vs PyTorch. We were able to observe subtle differences between CuPy and PyTorch when integrating them with the NumPy data science library. When paging large datasets, annotated PyTorch incurred a higher overhead from the additional data transfers compared to annotated CuPy. Using the NVIDIA Visual Profiler, we observed that while PyTorch explicitly executed the memory transfers and kernels that we invoked, CuPy was significantly more complicated (Figure 5). In particular, CuPy made several calls to functions like `cudaHostAlloc()` and `cuModuleLoadData()`. These extra functions may have allowed CuPy to perform more efficient data transfers when paging large datasets, incurring less overhead.

9 Limitations

Without a more sophisticated scheduling algorithm, the end user may not get optimal performance using an OA-annotated library since all possible annotated functions will execute on the accelerator by default. Some functions, such as specific machine learning algorithms or analytics operations like joins, may be more efficient on the CPU in some cases. Although GPU scheduling is a complex problem, we found that greedy scheduling was effective in many applications. Though we cannot definitively state whether a holistic scheduler is better, we believe this problem is worth exploring and can utilize the information captured in the runtime's dataflow graph.

Another limitation is the need, in some cases, for the end user to provide cost models to their workload. It is difficult to know what constitutes a good cost model, much less an

optimal one. With OAs, the end user can at least more easily evaluate different models and their downstream scheduling decisions with little code modification. In this case, the end user can simply change a few parameters and re-run the application, as opposed to re-inserting data transfer statements into the application code for each schedule. However, optimal scheduling remains a complex problem.

OAs are also unable to apply some types of low-level optimizations that require changes to the accelerator library functions. For example, the interface for CUDA streams, a method for overlapping data transfer and compute, is unavailable in cuDF and has a library-specific interface in PyTorch. Because OAs rely on diverse, existing libraries to provide optimized kernel implementations, indirectly calling into lower-level CUDA libraries like cuBLAS, cuDNN, and Thrust, they cannot coordinate calls to interfaces such as CUDA streams across these libraries. Nonetheless, users combining these accelerator libraries by hand would face the same limitation.

10   Related Work

OAs build on split annotations [23], which provide per-function annotations over existing CPU-based libraries to enable cross-function data pipelining and improved cache utilization. OAs extend split annotations by considering several new problems unique to accelerators, including data transfer and allocation across devices, memory limits of accelerators, and the problem of scheduling computations and transfers across CPUs and accelerators. These problems were not considered in the design of split annotations, so they require both an extended annotation interface (§4) and a different runtime and scheduler (§5).

Existing accelerator libraries such as PyTorch [24], cuDF [7], RAPIDS [9], and cuML [8] provide interfaces for targeting GPUs that intend to mirror CPU library APIs. However, they usually cannot handle data that does not fit in the accelerator memory, only support a subset of their CPU counterparts, and invariably involve application rewrites. OAs are a system designed to bridge these shortcomings, by leveraging new accelerator implementations of library functions but using annotations to automatically page and schedule work across the accelerator and CPU in complex applications that call multiple library functions.

Another popular approach for targeting heterogeneous platforms is compilation, where a compiler generates code (e.g., CUDA) underneath an existing library interface. Several solutions exist for data analytics [20–22, 27, 31, 34] and machine learning [11, 13, 30]. As one example, Numba [2] compiles NumPy code into an intermediate representation (IR), and then to optimized CPU or GPU code. However, compilers trade off good performance for high complexity: they are difficult to implement and to integrate into existing libraries, and the generated code may not match the performance of heavily hand-optimized kernels such as linear algebra functions [23]. In contrast, annotation-based approaches such as split annota-tions achieved similar speedups to these compilers in many cases with substantially less developer effort (e.g., 10× less code than accelerating functions with Weld in the Mozart evaluation [23]) by leveraging individual, hand-written kernel functions and only optimizing the data movement across them. Like split annotations, OAs propose using expert-written kernels to offload computations rather than attempting to generating code that matches the performance of these kernels.

Several existing systems schedule tasks in a heterogeneous environment [16–19, 26, 32, 33]. Our system primarily aims to provide an interface to bridge existing CPU and GPU code. The algorithms presented in these systems are complementary, and can be used to schedule the task graphs produced by OAs.

Hardware-portable languages like OpenCL [29] provide a common interface to target both CPU and accelerators. However, they require end users to write custom code in these languages, whereas our goal is to leverage optimized CPU and accelerator kernels that have already been written by expert developers and automatically invoke these kernels in an application written using high-level APIs.

11   Conclusion

We have presented *offload annotations (OAs)*, a new approach for bridging existing CPU libraries with emerging GPU libraries with no library code changes. Annotators use OAs to specify an accelerator function for a corresponding CPU function, and to define how inputs to a function can be transferred between devices. Optionally, annotators can also annotate allocation functions and provide cost model estimators that assist the runtime in making scheduling decisions. Our runtime, Bach, uses the information encapsulated in OAs to automatically schedule functions in an end-user application across devices, manage data transfer, and page large datasets. We apply OAs to several existing to several existing CPU libraries and show that they can improve their by up to 1200× and a median of 6.3× by offloading work to a GPU, with little to no code changes. We also show that OAs enable workloads that could previously not fit in GPU memory to reap the benefits of hardware acceleration, without manual effort by the application developer.

12   Acknowledgments

References

[1] Black Scholes Formula. http://gosmej1977.blogspot.com/2013/02/black-and-scholes-formula.html, 2013.

[2] Numba. https://numba.pydata.org, 2018.

[3] A Beginner's Guide to Optimizing Pandas Code for Speed. goo.gl/dqwmrG, 2019.

[4] Computational Formulas. https://oag.ca.gov/sites/all/files/agweb/pdfs/cjsc/prof10/formulas.pdf, 2020.

[5] Demo of DBSCAN clustering algorithm. https://scikit-learn.org/stable/auto_examples/cluster/plot_dbscan.html#sphx-glr-auto-examples-cluster-plot-dbscan-py, 2020.

[6] Importance of Feature Scaling. https://scikit-learn.org/stable/auto_examples/preprocessing/plot_scaling_importance.html, 2020.

[7] NVIDIA cuDF. https://github.com/rapidsai/cudf, 2020.

[8] NVIDIA cuML. https://github.com/rapidsai/cuml, 2020.

[9] NVIDIA RAPIDS. https://developer.nvidia.com/rapids, 2020.

[10] Truncated Singular Value Decomposition (TSVD). https://github.com/rapidsai/notebooks/blob/branch-0.12/cuml/tsvd_demo.ipynb, 2020.

[11] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[12] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards Type Inference for JavaScript. In *European conference on Object-oriented programming*, pages 428–452. Springer, 2005.

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[14] Tianyi David Han and Tarek S Abdelrahman. Reducing Branch Divergence in GPU Programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 3. ACM, 2011.

[15] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive KD Tree GPU Ray Tracing. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, pages 167–174. ACM, 2007.

[16] Víctor J Jiménez, Lluís Vilanova, Isaac Gelado, Marisa Gil, Grigori Fursin, and Nacho Navarro. Predictive Runtime Code Scheduling for Heterogeneous Architectures. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 19–33. Springer, 2009.

[17] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Chunling Hu, Brian T Lewis, and Keshav Pingali. Adaptive Heterogeneous Scheduling for Integrated GPUs. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 151–162. IEEE, 2014.

[18] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *Proc. USENIX ATC*, pages 17–30, 2011.

[19] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*, pages 341–352. ACM, 2012.

[20] HyoukJoong Lee, Kevin Brown, Arvind Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Implementing Domain-Specific Languages for Heterogeneous Parallel Computing. *IEEE Micro*, 31(5):42–53, 2011.

[21] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, et al. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *Proceedings of the VLDB Endowment*, 11(9):1002–1015, 2018.

[22] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. Weld: A Common Runtime for High Performance Data Analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2017.

[23] Shoumik Palkar and Matei Zaharia. Optimizing Data-Intensive Computations in Existing Libraries with Split Annotations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 291–305, 2019.

[24] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[25] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. In *ACM SIGPLAN Notices*, volume 50, pages 167–180. ACM, 2015.

[26] Vignesh T Ravi, Michela Becchi, Wei Jiang, Gagan Agrawal, and Srimat Chakradhar. Scheduling Concurrent Applications on a Cluster of CPU-GPU Nodes. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 140–147. IEEE, 2012.

[27] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: A Compiler and Runtime for Heterogeneous Systems. pages 49–68. ACM, 2013.

[28] Mark Silberstein. GPUs: High-Performance Accelerators for Parallel Applications: The Multicore Transformation (Ubiquity Symposium). *Ubiquity*, 2014(August), August 2014.

[29] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering*, 12(3):66–73, 2010.

[30] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 609–616, 2011.

[31] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):134, 2014.

[32] Lei Wang, Yong-zhong Huang, Xin Chen, and Chun-yan Zhang. Task Scheduling of Parallel Processing in CPU-GPU Collaborative Environments. In *2008 International Conference on Computer Science and Information Technology*, pages 228–232. IEEE, 2008.

[33] Yuan Wen, Zheng Wang, and Michael FP O'boyle. Smart Multi-Task Scheduling for OpenCL Programs on CPU/GPU Heterogeneous Platforms. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2014.

[34] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, volume 8, pages 1–14, 2008.