DelayRepay: Delayed Execution for Kernel Fusion in Python

John Magnus Morton
University of Edinburgh
Edinburgh, UK
Magnus.Morton@ed.ac.uk

Kuba Kaszyk
University of Edinburgh
Edinburgh, UK
Kuba.Kaszyk@ed.ac.uk

Lu Li
University of Edinburgh
Edinburgh, UK
Lu.Li@ed.ac.uk

Jiawen Sun
University of Edinburgh
Edinburgh, UK
jsun2@exseed.ed.ac.uk

Christophe Dubach
McGill University
Montreal, Canada
Christophe.Dubach@mcgill.ca

Michel Steuwer
University of Edinburgh
Edinburgh, UK
Michel.Steuwer@ed.ac.uk

Murray Cole
University of Edinburgh
Edinburgh, UK
mic@inf.ed.ac.uk

Michael F. P. O'Boyle
University of Edinburgh
Edinburgh, UK
mob@inf.ed.ac.uk

Abstract

Python is a popular, dynamic language for data science and scientific computing. To ensure efficiency, significant numerical libraries are implemented in static native languages. However, performance suffers when switching between native and non-native code, especially if data has to be converted between native arrays and Python data structures. As GPU accelerators are increasingly used, this problem becomes particularly acute. Data and control has to be repeatedly transferred between the accelerator and the host.

In this paper, we present DelayRepay, a delayed execution framework for numeric Python programs. It avoids excessive switching and data transfer by using lazy evaluation and kernel fusion. Using DelayRepay, operations on NumPy arrays are executed lazily, allowing multiple calls to accelerator kernels to be fused together dynamically. DelayRepay is available as a drop-in replacement for existing Python libraries. This approach enables significant performance improvement over the state-of-the-art and is invisible to the application programmer. We show that our approach provides a maximum 377× speedup over NumPy - a 409% increase over the state of the art.

ACM ISBN 978-1-4503-8175-8/20/11…$15.00
https://doi.org/10.1145/3426422.3426980

CCS Concepts: **• Software and its engineering** → **Dynamic compilers**; **• Computer systems organization** → *Heterogeneous (hybrid) systems*; **• Computing methodologies** → *Parallel programming languages.*

Keywords: delayed evaluation, code fusion, dynamic compilation, GPU

ACM Reference Format:
John Magnus Morton, Kuba Kaszyk, Lu Li, Jiawen Sun, Christophe Dubach, Michel Steuwer, Murray Cole, and Michael F. P. O'Boyle. 2020. DelayRepay: Delayed Execution for Kernel Fusion in Python. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages (DLS '20), November 17, 2020, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3426422.3426980

1 Introduction

Python is a popular byte-code interpreted, general-purpose programming language. It is now commonly used for both numerical computing and machine learning, in spite of the poor baseline performance of CPython, the mainstream implementation. To improve on this, practitioners use libraries such as NumPy — for linear algebra and general numerical computing — and PyTorch — for deep learning. These libraries avoid the performance problems of the Python interpreter by offloading the computation to native code. NumPy, for example uses a mixture of C and FORTRAN. Unfortunately, data still has to be transferred to and from the Python interpreter, and there is an overhead when dispatching python function calls to native ones.

This problem is magnified when trying to use modern accelerator hardware. Graphics Processing Units (GPUs) are widely used for accelerating numerical computation, and are increasingly popular with the rise of deep neural networks. There is, therefore, a significant interest in accessing the computational power of GPUs while maintaining the programmability of Python. Unfortunately, GPUs are more challenging to program than CPUs, requiring the use of a specialized kernel language and host management APIs such as OpenCL or CUDA. In addition, the use of accelerators exacerbates the problems of data transfer between the Python interpreter and the kernel performing the computation, Figure 1 shows a 3x speedup can be achieved for a typical workload if such data transfers are optimized. As a result, several Python GPU libraries have been developed to make GPUs more accessible.

The simplest approaches, such as PyOpenCL and PyCUDA [14], only add thin wrappers to the existing OpenCL and CUDA APIs. Whilst this enables the writing of host management code directly in Python, programmers still have to write the low-level kernel code that will run on the GPU. Numba [17] is a more advanced approach which features a JIT compiler for accelerating Python functions. However, programmers still have to be strongly aware of the CUDA programming model, with the Python code resembling low-level CUDA code with a Python syntax.

CuPy [20] is a drop-in replacement for NumPy, a library that underpins other popular scientific Python libraries such as SciPy [27] and Pandas [19]. CuPy is built on CUDA and uses the CUBLAS library and custom GPU kernels to implement the various NumPy array operations. Promisingly, CuPy allows programmers to use NumPy arrays and operations on a GPU with minimal effort. However, it still suffers from the overhead of switching between Python and native code on each NumPy invocation.

Similarly to CuPy, Bohrium [15] provides a drop-in replacement for CuPy but focusing on performance portability across multiple parallel platforms. Similarly to DelayRepay, Bohrium supports kernel fusion. However, Bohrium has a radically different compilation model, using a bytecode virtual machine as an intermediate representation. This heavyweight runtime imposes severe performance penalties when compared to CuPy and DelayRepay, and is difficult to debug: CuPy and DelayRepay can output complete CUDA kernels as source that can be reused in other applications, while Bohrium produces compiled code and many JIT-compiled object files to support its runtime system. The ideal scenario would be to have the best of both worlds: easy access to GPU performance - but without the overheads.

This paper presents DelayRepay, a drop-in replacement for the NumPy library that allows seamless GPU acceleration of existing NumPy code. DelayRepay uses a lazy array operation mechanism and a just-in-time (JIT) compiler to generate and execute automatically fused CUDA kernels. The runtime system builds an execution graph of NumPy function calls which is then transformed into a graph of CUDA kernel fragments. Instead of converting and executing each kernel fragment independently, DelayRepay automatically fuses fragments. The compiled kernels are executed, with data transfer and device memory allocation handled automatically. This approach improves performance while still completely abstracting the GPU programming model from the programmer.

```
np.sin(x) ** 2 + np.cos(x) ** 2
```

Listing 1. Computation of the Pythagorean identity with NumPy operators `sin, cos, +` and `**` (power). `np` is an alias for the NumPy module import.

This paper makes the following contributions:

1. It presents a new delayed execution framework for NumPy;
2. It demonstrates a novel JIT compiler for compiling chains of NumPy operations to CUDA code;
3. It shows how this approach is used to automatically fuse array operations for improved performance.
4. It provides a thorough evaluation of this approach compared to the current state-of-art method.

The rest of this paper is structured as follows. Section 2 outlines our motivation. Sections 3 and 4 describe the approach and implementation of our NumPy delayed execution framework and JIT compiler. We present and discuss our experimental results in section 5. Finally, Sections 6 and 7 present related and future work.

2 Motivation

Python has become a popular language for intensive numerical code, despite its humble beginning as a scripting and application language. This success has been made possible thanks to the availability of a wide range of high performance libraries, such as NumPy. These libraries are typically implemented in native code under the hood, since pure Python code is often significantly slower than native code.

2.1 GPU Execution with CuPy

CuPy strives to deliver high-performance on GPUs by supporting most NumPy operations. It achieves this by having all major operators (*e.g.,* vector addition) implemented as CUDA kernels. CuPy keeps the data on the GPU between calls to different operators. This strategy minimizes data transfer which is one of the main source of overheads. Nonetheless, when executing a sequence of GPU operations, control will go back and forth between Python and the GPU. This results in multiple GPU kernel calls, which induce non-negligible overheads even when data remains on the GPU.

Consider the Python code in listing 1 which makes use of NumPy operations. This snippet computes the Pythagorean identity element wise over a NumPy input array `x`. Running this expression in NumPy invokes five separate computations implemented in the C backend of NumPy.

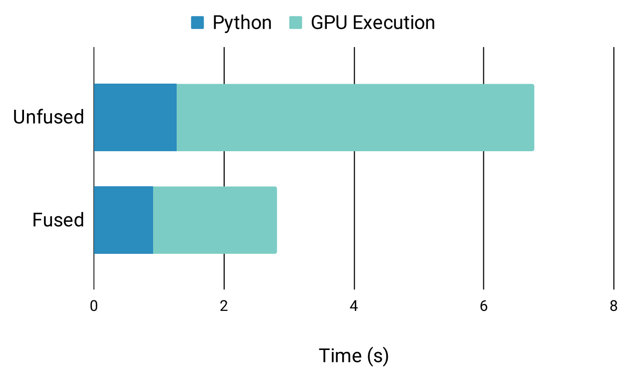

Figure 1. Execution time of separate kernels vs. fused kernels for the pythagorean_id benchmark. Time spent as GPU execution includes data transfer and kernel execution. Input array size is 808 MB.

This code can be executed with CuPy unchanged by modifying some import statements. Running it with CuPy results in compiling and executing five CUDA kernels, and the allocation of four on-device buffers (three for intermediate results and one for the final result).

2.2 Lazy Fusion with Bohrium

Similarly to CuPy, Bohrium [15] provides a drop-in replacement for NumPy. Unlike CuPy, Bohrium aims for performance portability, targeting multiple backends including CUDA, OpenCL and OpenMP. Bohrium achieves this using its *vector bytecode* virtual machine, a heavyweight compilation and runtime system. Similarly to DelayRepay, Bohrium has support for array operator fusion via lazy evaluation [16]. However, Bohrium's *vector bytecode* imposes a severe performance penalty (Section 5.4.2).

In contrast, DelayRepay's novel lightweight lazy fusion at the NumPy AST level allows Python developers to benefit from kernel fusion with a significantly reduced overhead. We have designed DelayRepay explicitly for extensibility, allowing the possibility of programmable heuristics being added at the Python level. DelayRepay also provides benefits in debugging, with the generated kernels available to view if required. These kernels are portable, and can be reused in other CUDA programs.

2.3 The Need for Fusion of Operations

Figure 1 shows the run time breakdown for executing the five separate GPU kernels corresponding to the code in Listing 1. It also shows the time it takes, if these five kernels are fused into a single GPU kernel. The fused versions is 2× faster than the un-fused version. Since CuPy keeps the data on the GPU between kernel invocations, the difference in execution time is mainly due to:

- the overheads of launching five separate kernels instead of a single one from Python;
- the additional memory reads and writes required to pass data between the separate kernels.

This example illustrates the core problem that this paper addresses. Python GPU libraries, such as CuPy, implement a wide range of small primitives, each with their corresponding GPU kernels.

GPU kernel fusion [28, 29] is an optimization technique that is especially well-fitted when crossing a language border is costly. Fusing kernels also has the potential for increasing the compute intensity and increasing data locality between operations. Fusing also decreases memory footprint by removing the need for intermediate results.

The goal of this paper is to develop a runtime mechanism to support the automatic fusion of GPU kernels, without having to make modifications to the existing Python code. We wish to achieve this in a way which allows implicit and automatic fusion of GPU kernels, to reduce the overheads associated with several separate GPU kernel calls.

2.4 Overview

This paper introduces a *delayed execution* mechanism for the popular NumPy library coupled with a *compiler-based fusion* mechanism. Figure 2 presents the overview of our approach. DelayRepay takes NumPy code as input with minimal modifications required. It only requires to replace the import statement of NumPy to our library.

The delayed execution mechanism returns an AST node when calling a NumPy operation, *delaying* its execution. When the result of such an operation serves as an input to another NumPy operation, the AST is simply extended, implementing a lazy evaluation strategy. When a non-NumPy operation is called on the resulting AST node, this triggers the compilation to a CUDA kernel and finally, execution on the GPU. This lazy evaluation strategy is the first major difference to existing solutions. CuPy for instance, eagerly executes each NumPy operation as an individual kernel on the GPU.

The delayed execution mechanism enables the ability to fuse operations before execution on the GPU. This is the second big difference to CuPy. Our compiler-based fusion mechanism fuses multiple NumPy operations together. This reduces the overheads of individual kernel launches. Once the operators are fused, DelayRepay generates and executes CUDA kernels on the GPU.

As we will see in the following sections, our approach is built on top of the existing CuPy library and requires only minor modifications to the original library code.

3 Delayed Execution

This section describes the design and implementation of DelayRepay, our delayed execution framework for NumPy.

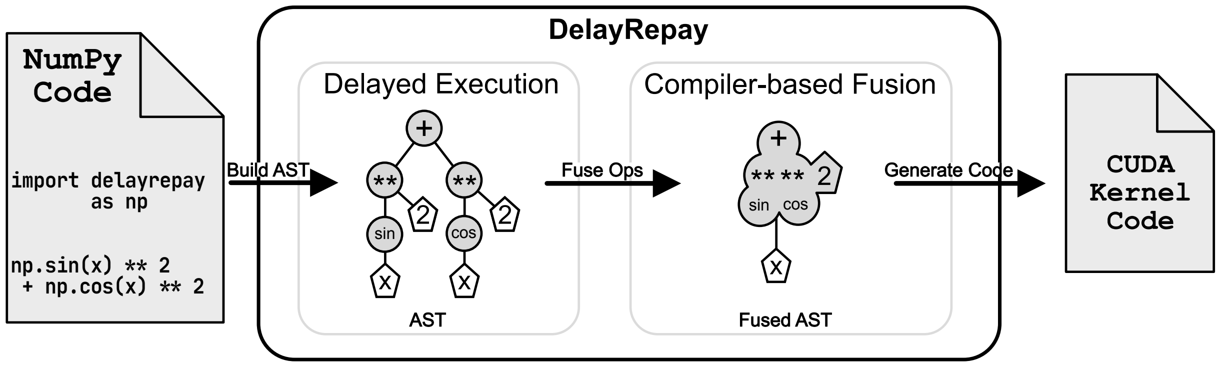

Figure 2. DelayRepay starts with standard NumPy code with a modified import. During runtime, NumPy operations are delayed by building, step-by-step, an AST. Then, chains of operations are fused. Finally, the compiler generates a single CUDA kernel from the fused operations and the kernel is executed.

Driven by the issues discussed in Section 2, our design goals with DelayRepay are to develop a drop-in replacement for NumPy that improves performance with automatic kernel fusion. This is achieved without any changes to existing programs, other than replacing a single import statement.

3.1 Numpy Basics

The NumPy library is built around its native n-dimensional array implementation the `ndarray` type. *Universal Functions* (known as ufuncs) are elementwise functions that operate on `ndarray` instances. These ufuncs are *broadcast* over the `ndarray`, i.e., these functions are vectorised, automatically. Examples of ufuncs include elementwise add and trigonometric functions. NumPy has other built in functions that operate on `ndarray` such as matrix multiplication or dot product. The semantics of some functions, such as `dot`, change depending on the shapes of its inputs. The `dot` function can be a dot product, matrix multiplication or matrix-vector multiplication, and this is only determined at runtime. Unlike ufuncs, these functions are not broadcast at the NumPy level.

3.2 Delayed Mechanism with NumPy

The NumPy original `ndarray` is implemented as a class in Python. To achieve our goal of delaying execution of any `ndarray` operations, we simply extend this class with our own `DelayArray` class. Using the existing Python inheritance mechanism, our class simply overrides all the original operators from the `ndarray` class.

Our implementation of the operators always returns a `DelayArray` object rather than the result of executing the operator. Our `DelayArray` object captures the call graph of the various operations performed as an AST, as seen in fig. 2. No NumPy calls are evaluated during this process, and other Python interpretation carries on as normal. The AST is not compiled or executed until it absolutely needs to be.

Listing 2 shows a simplified version of our implementation. As can be seen, our `DelayArray` class overrides the main entry points of the original NumPy `ndarray`. The difference is that instead of directly executing the various operators, we construct AST nodes which are all sub-classes of the `NumpyEx` class, representing NumPy expression. This allows us to chain multiple such operations and trigger execution at a later stage. We have a `BinaryFuncExpression` class representing binary ufuncs (*e.g.,* add, multiply) and specific nodes for other array functions (*e.g.,* `ReduceEx`). The leaf nodes in the tree can be either `Scalar`, representing a numeric constant, or `NPArray`, representing a reference to an original NumPy array.

The special `__repr__` function returns a String representation of the array. The special `__array__` function returns an actual original NumPy array. These functions will be discussed in more detail below.

3.3 Implicit Execution

Tensor-based machine learning frameworks such as PyTorch and Tensorflow also build expression graphs, but evaluation is forced explicitly and eagerly. In DelayRepay, the expression graph is compiled and evaluated only when computation cannot feasibly continue on the GPU. This happens for instance when the result needs to be sent to I/O (*e.g.,* printing result) or if the resulting array is needed in non-NumPy or unsupported computation.

We achieve this by defining the `__repr__` and `__array__` methods of our `DelayArray` class. The `__repr__` method is used to define the pretty-printed representation (and in our case, the string representation) of an object. The `__array__` method is invoked when NumPy converts an array-like object into an ordinary NumPy `ndarray`. When `__array__` is invoked on a `DelayArray` instance, evaluation is forced.

```
class DelayArray:
    ...
    def __array_ufunc__(self...):
        ...
        if isunaryfunc(func):
            return UnaryFuncEx(func,
               ↪ arg)
        ...
    def __array_function__(self...):
       ...
    def dot(self...):
       ...
    def __repr__(self:...):
       ...
    def __array__(self:...):
       ...
class NumpyEx(DelayArray):
    ...
class ReduceEx(NumpyEx):
    ...
class UnaryFuncEx(NumpyEx):
    def __init__(self, func,arg):
        # store function name
        self.func = func
        # store function argument
        self.children = [arg]
class BinaryFuncEx(NumpyEx):
    ...
class NPArray(NumpyEx):
    ...
class Scalar(NumpyEx):
    ...
class CuPyExp(NumpuEx):
    ...
...
```

Listing 2. Interface to our NumPy array replacement and delayed AST types

```
1  import delayrepay as np
2
3  data = np.random.random((1000,))
4  # data = NPArray
5
6  l = np.sin(data)
7  # l = UnaryFuncEx('sin', data)
8
9  l_sq = l ** 2
10 # l_sq = BinaryNumpyEx('**', l, 2)
11
12 r = np.cos(data)
13 # r = UnaryFuncEx('cos', data)
14
15 r_sq = r ** 2
16 # r_sq = BinaryNumpyEx('**', r, 2)
17
18 tot = l_sq + r_sq
19 # tot = BinaryNumpyEx('+', l_sq,
      ↪ r_sq)
20
21 print(tot)
22 # GPU evaluation forced here
```

Listing 3. Annotated code sample showing how intermediate wrapper types are returned during delayed evaluation

3.4 Example

Listing 3 shows the step-by-step execution of the Pythagorean identity Python benchmark. Line 3 starts by creating an array filled with random values. Then line 6 calls the `sin` function which is one of the operators that our `DelayArray` class supports. As a result of this call, an AST node representing the sin unary function is returned in lieu of the actual data, *delaying* the execution of this operation. On line 9, a similar process occurs where an AST node representing the power function, applied to its argument (one of them being an AST node itself) is returned. This process continues with the other function calls until a call the call to `print` on line 21. At this point, print will internally call our overriden `__repr__` function which will trigger a compilation and execution of the tree of operations.

3.5 Fall-Back Mechanism

Our delayed execution targeting the GPU is built on the GPU execution capabilities of CuPy. To maintain full backwards compatibility with NumPy we provide a fall-back mechanism. For any un-implemented NumPy functions, we implicitly fall-back to the implementation provided by CuPy targeting the GPU. For operations executed via the fall-back mechanism we wrap the resulting array using a `CuPyExp` class instance to allow composition with the rest of our system. When such an operation is encountered, the system will trigger an immediate evaluation since a `CuPyExp` is treated as an opaque operation that will never be fused with any of the preceding operations.

4 Compiler-Based Fusion

The delayed execution phase produces an AST containing nodes representing NumPy operations. In a second phase, when execution is implicitly triggered, DelayRepay fuses compatible operators together. A GPU kernel is generated for each fused operation and then executed. The final result of the computation is copied from the GPU back into a fresh NumPy array which is returned to the Python code.

4.1 Operator Fusion

Unlike traditional forms of GPU kernel fusion that work at the level of kernel source code, DelayRepay's fusion approach works on a high-level abstraction - our AST of NumPy expressions. This has several benefits:

- first, the implementation is fairly simple and not concerned with performing analysis on low-level code (e.g. investigating array indexing and pointer arithmetic) to determine which kernels can be fused;
- secondly, this results in a very quick fusion process;
- finally, our fusion implementation is not tied to one accelerator programming API and could easily be extended to others such as OpenCL or SYCL.

DelayRepay's fusion algorithm is simple: DelayRepay greedily fuses every lazy AST into one kernel each. This is safe since operations that would break fusion e.g. a reordering of an array, trigger eager evalution of their input, falling-back to the appropriate CuPy operation. NumPy's broadcasting rules also prevent the fusion of shape-mismatched operations.

There are cases where operations can not be fused straightforwardly or when fusion is not desirable. For example, when computing the sum of an array and using this result in a larger computation, generating a performant fused kernel is difficult. DelayRepay's fallback mechanism handles such cases and produces multiple CUDA kernels. Work is ongoing on a programmable heuristic system to control when fusion will happen in such cases.

4.2 CUDA Kernel Generation

When eager evalution is triggered, DelayRepay generates CUDA kernels.

At a high-level, we traverse the tree in post-order, convert it into SSA form by translating each node into a simple CUDA expression. These expressions are then simply spliced into a CUDA kernel template that handles bookkeeping and input and output arrays.

In detail, at each node, we collect some meta data in the form of a `Fragment` object that represents the core pieces of an incomplete CUDA kernel: a name, a list of statements describing the computation to be performed, and a dictionary of inputs. Each statement is a one-line CUDA statement. Each statement is generated in SSA form and appended to a list that has been generated recursively while traversing the previous nodes. For example, in the case of a binary expression these statements have form `T out = input1 <op> input2` where `<op>` is the binary operator and T represents the type. DelayRepay provides templates for the different possible expression type e.g. unary and binary function calls.

```
class NumpyEx(DelayArray):
 ...
 def to_kernel(self):
  # Merge statements
  body = concat(self.stmts)
  # Generate kernel argument
     ↪ signature
  input_types =
        arg_types(self.kernel_args)

  # Construct kernel template object
  return cupy.ElementwiseKernel(
      input_args,
      self.return_type,
      body,
      self.name}
  )
```

Listing 4. Kernel generation pseudocode

The dictionary of inputs contains keys possibly referring to fragment names or values represented by `InputFragment` that encapsulate either a reference to an input array or to the output of another GPU kernel.

In our kernel generation we take advantage of the fact that it happens dynamically at runtime. Therefore, we know the values of all non-array variables and simply treat them as constants, folding their values directly in the generated kernel i.e. partially evaluating the scalar part of the program. This avoids complications such as wrapping scalar variables as single-element arrays common in other approaches and also enables further benefits such as constant-folding optimisations by the CUDA compiler.

As each node is visited, a `Fragment` object is created that merges the statement lists and input dictionaries of the node's children. The final fragment returned from the root of the tree encapsulating a complete kernel. For this, we generate the code as shown in Listing 4 for the the element-wise case. The statements in the list are concatenated together and used as the body of a CuPy `ElementWiseKernel`. Our implementation also supports the generation of reductions via CuPy's `ReductionKernel`. These CuPy kernels are lightweight abstractions used to template CUDA kernels and automatically generate code for handling the thread and workgroup IDs.

The compilation process is entirely deterministic i.e. the same NumPy program will generate *exactly* the same CUDA kernel with repeated runs, even with different inputs. This means our implementation is able to leverage caching of generated and compiled kernels.

```
 1 #include <cupy/carray.cuh>
 2 __global__ void
 3   kernel(const CArray<double,1> _in,
 4                CArray<double,1> _out
                     ↪ ,
 5                CIndexer<1> _ind) {
 6  CUPY_FOR(i, _ind.size()) {
 7   _ind.set(i);
 8   const double &in = _in[_ind.get()
        ↪ ];
 9   double &out = _out[_ind.get()];
10   double unfunc1 = sin(in);
11   double binex2 = unfunc1 * unfunc1;
12   double unfunc3 = cos(in);
13   double binex4 = unfunc3 * unfunc3;
14   double binex5 = binex2 + binex4;
15   out = binex5;
16  }
17 }
```

Listing 5. Automatically generated fused Pythagorean identity GPU kernel code.

We generate one such kernel for each fused operation. These kernels are then compiled by CUDA and scheduled for execution.

4.3 Generated Kernel Example

Listing 5 shows the kernel generated for the Pythagorean identity. The generated code makes use of templates and macros provided by CuPy. The `CArray` template is a wrapper for accessing NumPy arrays on the GPU. In the one dimensional case these correspond to raw-pointers. The `CIndexer` provides an abstraction to index into (potentially multi-dimensional) arrays. Finally, the `CUPY_FOR` macro abstracts the use of thread ids away distributing the work across all global threads.

After providing references for accessing the input and output arrays in lines 9 and 10, the next five lines (11–15) correspond to the individual operations of the Pythagorean identity computation: element-wise computation of the `sin` and `cos` functions (lines 11 and 13), computing the power of two (lines 12 and 14), and adding up the intermediate results in line 15. Finally, the computed value is written to the output array via the reference (line 16).

4.4 Execution of CUDA kernels

Since our implementation is built on CuPy, we can benefit from its existing infrastructure for scheduling kernels and managing memory. Our wrapped arrays are automatically allocated on the GPU device, and data is transferred transparently. Similarly, space for result arrays is automatically managed. CuPy hooks GPU memory buffers to Python's garbage collector, so no-longer needed data is automatically deallocated. CuPy also allows us to use a normal Python function call syntax to invoke the kernel with its arguments - no need for explicit function calls to set kernel arguments. Using these tools, we simply call each kernel in order and store its result. Future dependent kernels can access this stored result as needed. The result of the final kernel invocation is returned to the user.

5 Evaluation

This section evaluates our DelayRepay runtime and compiler against the CuPy system, the Bohrium compiler and runtime system, and the standard CPU Numpy library using a selection of realistic NumPy benchmarks. Since DelayRepay is a drop-in replacement for NumPy our benchmarks are unmodified from their NumPy versions, except for changing the module imports. Our benchmarks are selected to be as realistic as possible and to cover as much of the NumPy API as possible. All benchmarks are of a data-parallel nature making them appropriate for GPUs.

We measure the performance of each library for the warmed-up execution times for each benchmark. 'Warmed-up' means that a previous run has completed, so analysis and GPU kernel compilation has been cached where possible, and we record this warmup time. We also measure the wall-clock time for a single end-to-end run for each benchmark including the time to launch and shutdown the Python interpreter.

Our primary results report benchmark speed-ups against NumPy, CuPy and Bohrium. We then relate the speedups to the fusion of kernels and improved memory usage to gain insights on the reported performance improvements.

5.1 Benchmarks

We have modified and extended an existing Numpy benchmark suite. Many of the benchmarks were originally sourced from StackOverflow, a popular programming question and answer forum, indicating that they are being used in the open source community.

We parameterized the benchmarks to easily select the library we wish to benchmark and control data creation and repetition. A brief description of each benchmark is found in Table 1.

The benchmarks are mostly high-level numerical NumPy programs. This selection aims to exercise as much of the libraries as possible, with benchmarks covering element-wise array operations, reductions, array slicing, and various combinations and repetitions of these.

Table 1. Benchmark descriptions

Benchmark	Description
`arc_distance`	Pairwise arc distance between all points in two vectors
`create_grid`	Creates and reshapes grid
`cronbach`	Cronbach's α function. Bohrium produces an error when running this benchmark
`euclidean_distance_square`	Square of Euclidean distance
`evolve`	Laplacian evolution
`grayscott`	Gray-Scott reaction diffusion model
`harris`	Harris corner detection
`hasting`	Hasting and Powell mode
`l1norm`	L1 Norm
`l2norm`	L2 Norm
`laplacien`	Laplace transform
`log_likelihood`	Log-likelihood of normal distribution
`lstsqr`	Least squares solution to linear system of equations
`nn`	Simple numpy neural network
`pairwise`	Pairwise linear distance
`pythagorean_id`	Computes Pythagorean identity
`repeating`	Fills matrix using the repeat function
`reverse_cumsum`	Reverse cumulative sum of array
`rosen`	Rosenbrock function
`specialconvolve`	Image convolution
`vibr_energy`	Classical mechanics

5.2 Hardware and Software Platform

All benchmarks are run on a server running OpenSUSE LEAP 42.1 with a 12-core Intel Xeon E5-2620 running at 2.00 GHz, 16 GB RAM and a nVidia Tesla K20 with XGB of VRAM. The CUDA version is 8.0.1, the CuPy version is 7.2.0, the NumPy version is 1.18.1, and the Bohrium version is 0.11.0.

5.3 Experimental Methodology

Our benchmark framework runs each benchmark for NumPy, CuPy, Bohrium, and DelayRepay. These dependencies are injected at runtime. For each run, the framework creates the required input data and then performs a warmup run of the benchmark, followed by 10 iterations. When the set of 10 runs is complete, the framework clears all GPU kernel caches: the CUDA compute cache; the CuPy cache of generated and compiled kernels; Bohrium's cached binary CUDA kernels, compiled bytecode and object files; and DelayRepay's internal analysis cache.

For the end-to-end wall-clock benchmarking runs, we fork a new Python process to run the entire benchmark. 10 runs are recorded without any warmup, and we clear all compute and kernel caches as before, but also clear the caches between repetitions of each benchmark, as well as between different benchmarks and libraries.

For both sets of benchmarking runs, we report the mean execution time of the 10 runs.

We profile with Python's built-in `cProfile` tool.

5.4 Performance Results

This section details the results of the experiments described above. The speedups achieved for the core computations are shown in Figure 3 and for the end-to-end execution of the entire benchmark application are shown in Figure 4. The detailed numbers reported in these figures can be found at the end of the paper in Table 2 and Table 3.

5.4.1 Speed-up of Computational Code. Figure 3 shows the overall speedups of the DelayRepay and CuPy computation times against a NumPy baseline. We can make the following observations:

First, we can see that all evaluated benchmarks benefit from the GPU execution. This is not surprising as it confirms the data-parallel nature of NumPy code.

Second, there are no benchmarks where CuPy significantly outperforms DelayRepay.

Third, the mean speedup is generally close to the maximum speedup. Several benchmarks show a significantly lower minimum speedup for all libraries, suggesting a limitation in our warmup approach here. This effect is more pronounced with Bohrium.

Finally, DelayRepay outperforms CuPy on a number of benchmarks including `arc_distance`, `pythagorean_id`, `vibr_energy` and `harris` where DelayRepay achieves about a twice as high speedup over NumPy as CuPy does. DelayRepay outperforms Bohrium on some benchmarks, while Bohrium outperforms DelayRepay on others. We note that Bohrium significantly underperforms DelayRepay, CuPy and NumPy in several benchmarks. Across all benchmarks, DelayRepay achieves a geometric mean speedup of 24.05× compared to 15.11× by CuPy - a 60% increase in the mean speedup - and 10.51× by Bohrium - a 130% increase in the mean speedup. DelayRepay achieves a maximum speedup of 377.40× compared to 74.06× by CuPy - a 409% increase - and 203.16× by Bohrium - an 86% increase.

5.4.2 Speed-up of End-to-End Execution. Figure 4 shows the speedup of the end-to-end execution that includes the time used for executing the non-NumPy Python code. This results show where the use of the GPU pays off. We can make the following observations:

First, not all benchmarks benefit from executing on the GPU when including the additional time the benchmark takes to execute on the Python interpreter.

Second, there are still a number of benchmarks that achieve large (>4×) speedup compared to NumPy when executed with DelayRepay (e.g., `log_likelihood`, `vibr_energy`, and `pythagorean_id`).

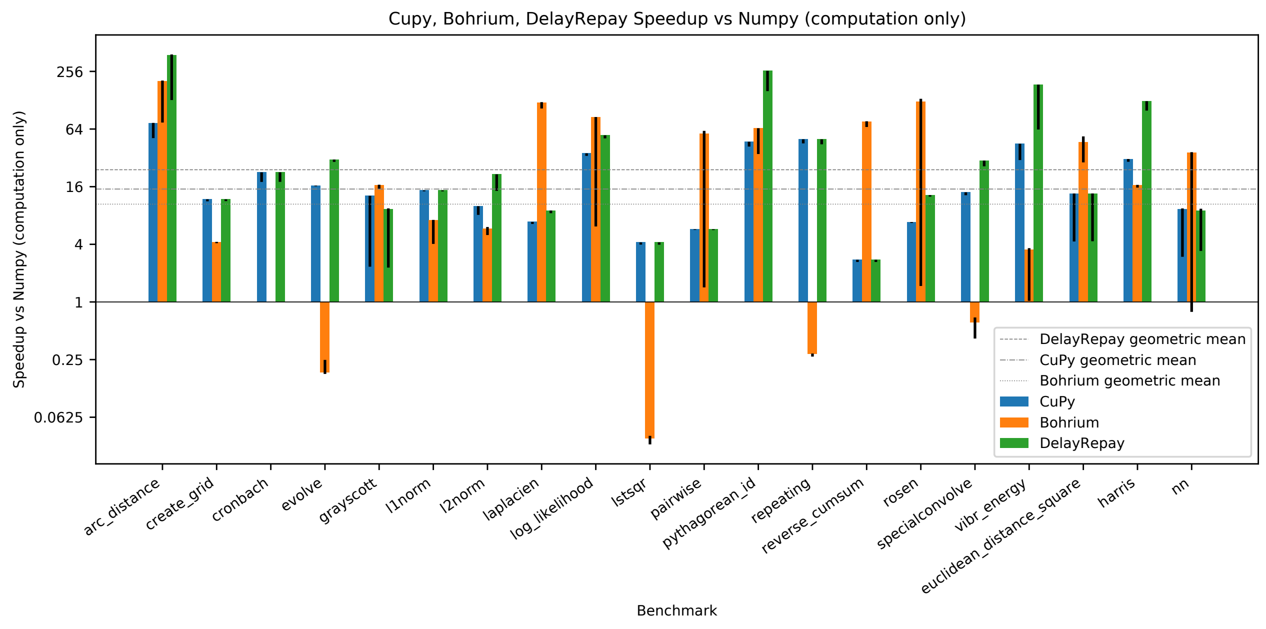

Figure 3. Speedup of computational code of CuPy and DelayRepay over the NumPy baseline. Higher is better. DelayRepay clear outperforms NumPy on all and CuPy and Bohrium on most benchmarks. Error bars represent the minimum and maximum speedup achieved. Bohrium produced an error when running `cronbach`.

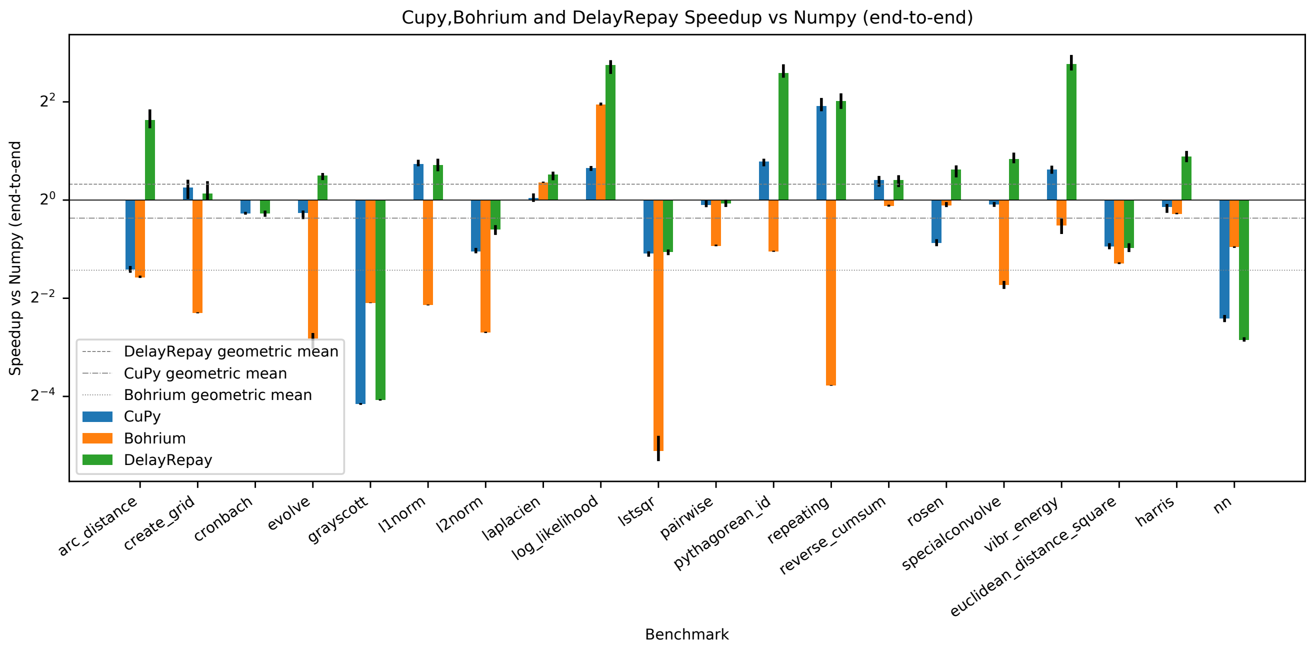

Figure 4. Speedup of end-to-end benchmarks of CuPy and DelayRepay over the NumPy baseline. Higher is better. Most benchmarks benefit from GPU execution. Error bars represent the minimum and maximum speedup achieved. With DelayRepay multiple benchmarks are profitable that are not using CuPy (`arc_distance`, `evolve`, `rosen`, `specialconvolve`, and `harris`) or Bohrium (`arc_distance`, `repeating`, `vibr_energy`, `evolve`, `create_grid`, `l1norm`, `pythagorean_id`, `repeating`, `reverse_cumsum`, `rosen`, `specialconvolve`, and `harris`). Bohrium produced an error when running `cronbach`.

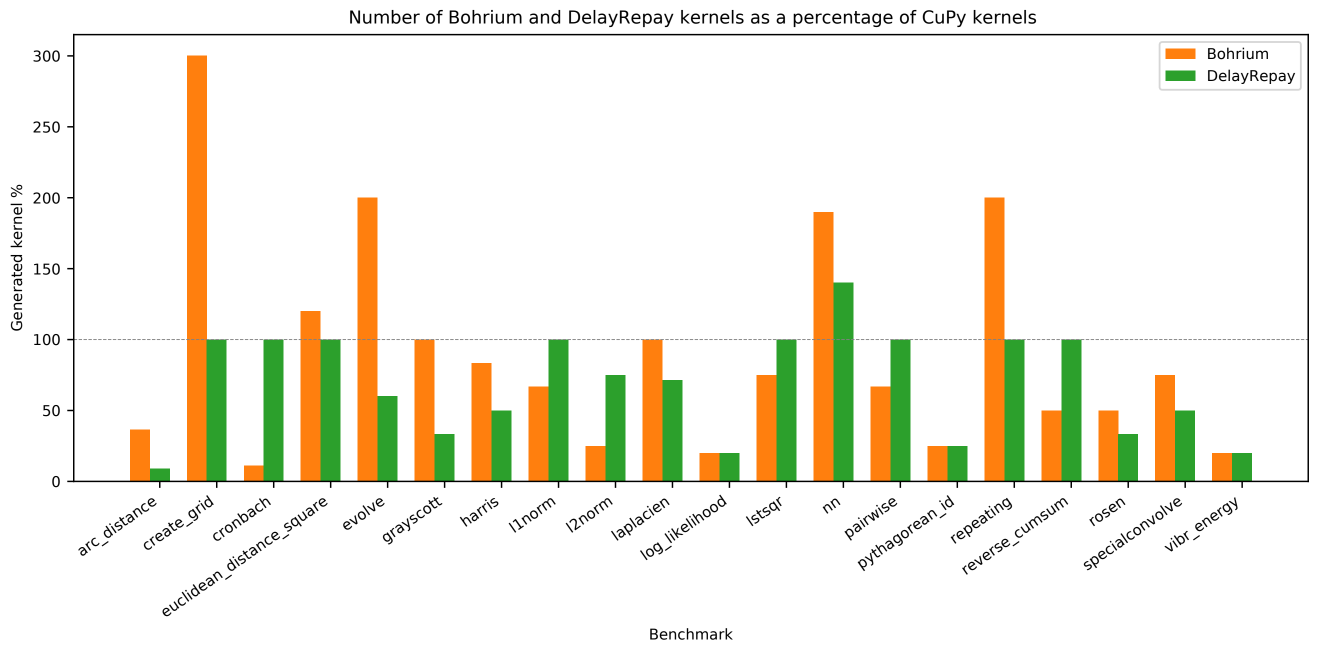

Figure 5. Relative number of kernels generated by Bohrium and DelayRepay as a fraction of number of kernels generated by CuPy. Lower is better.

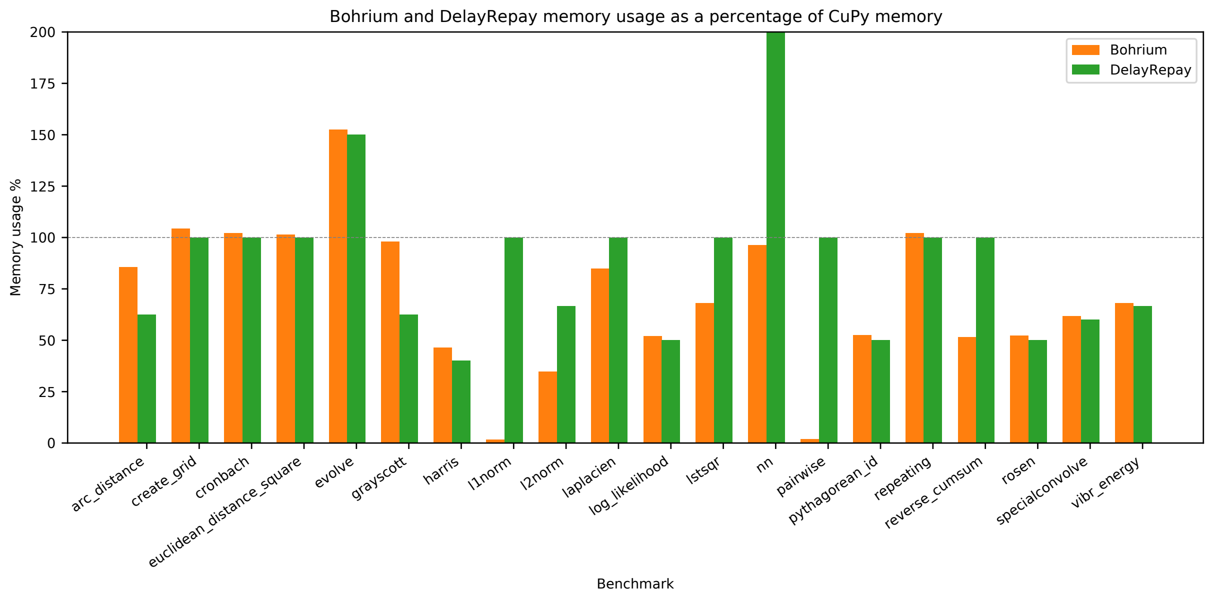

Figure 6. Relative memory usage of Bohrium and DelayRepay as a percentage of CuPy. Lower is better.

Third, the observed speedups are much more stable than for the computation-only benchmarks.

Finally, there are a number of benchmarks that become profitable to be executed on the GPU with DelayRepay that are not when using CuPy or Bohrium, with Bohrium producing a number of extreme slowdowns.

This includes `arc_distance`, `evolve`, `rosen`, `specialconvolve`, and `harris` for CuPy; and `arc_distance`, `repeating`, `vibr_energy`, `evolve`, `create_grid`, `l1norm`, `pythagorean_id`, `repeating`, `reverse_cumsum`, `rosen`, `specialconvolve`, and `harris` for Bohrium. These are the benchmarks where the additional optimisation of kernel

Table 2. Computation-only benchmark results

Benchmark	Numpy (s)	CuPy (ms)	Bohrium (ms)	DelayRepay (ms)	Input Size (MB)
arc_distance	1.96	26.45	9.64	5.19	320.00
create_grid	0.62	52.21	145.52	52.22	0.00
cronbach	4.16	184.21	N/A	184.26	0.06
euclidean_distance_square	1.18	87.21	25.30	87.13	2400.00
evolve	1.98	120.51	10801.32	64.49	3200.00
grayscott	2.52	196.39	151.98	271.78	2304.12
harris	3.11	99.84	186.63	24.88	800.00
l1norm	3.15	215.66	442.72	215.66	665.40
l2norm	0.91	91.03	156.73	42.32	0.00
laplacien	3.26	475.70	26.92	361.24	476.79
log_likelihood	7.48	210.02	88.04	136.07	7.18
lstsqr	2.29	547.66	60668.87	547.65	1568.00
nn	0.87	93.46	24.01	96.42	800.00
pairwise	1.59	277.14	27.83	277.18	1536.00
pythagorean_id	3.56	75.65	54.79	13.74	808.00
repeating	2.24	44.82	7826.68	44.85	1624.00
reverse_cumsum	1.63	590.95	21.34	591.04	105.71
rosen	1.68	245.51	13.53	128.78	1.36
specialconvolve	1.85	132.55	3031.52	61.74	666.89
vibr_energy	4.21	93.90	1191.17	22.63	80.00

fusion enabled by our delayed execution strategy pays off compared to the eager execution strategy employed by CuPy, while our lightweight analysis and code generation approach pays off against Bohrium's heavyweight bytecode generation approach.

On average, using the geometric mean, CuPy is slower than NumPy with a speedup of 0.77×, Bohrium is significantly slower than NumPy with a speedup of 0.37×, while DelayRepay achieves a speedup of 1.25×. DelayRepay achieves a maximum speedup of 6.79× compared to 3.75× by CuPy and 3.85× by Bohrium.

5.5 Analysis

Figure 5 and Figure 6 provide some insights into the performance achieved by DelayRepay over CuPy. Figure 5 shows the percentage of kernels generated by DelayRepay and Bohrium compared to CuPy for each benchmark. Figure 6 show the memory usage of DelayRepay and Bohrium as a percentage of CuPy's memory usage.

We can see in Figure 5 that for benchmarks with large performance advantage over CuPy such as `arc_distance` or `log_likelihood` DelayRepay executes a fraction of the GPU kernels that CuPy does by applying the fusion optimization. This results also in a reduced memory usage versus CuPy, as shown in Figure 6, as less intermediate results have to be stored.

The direct comparison of Bohrium in Figure 5 is less useful as it is not based on CuPy, and we have no knowledge of what an unfused Bohrium baseline looks like. However, we observe that generally, Bohrium produces more CUDA kernels than DelayRepay when DelayRepay outperforms Bohrium and *vice versa*. DelayRepay uses marginally less memory than Bohrium for most benchmarks.

The `nn` benchmark is an outlier where our fusion implementation produces more kernels than CuPy, resulting also in a higher memory usage and poor performance compared to CuPy. This is a result of the constant folding optimisation described in Section 4.2: each round of the algorithm uses different weight values, which results in DelayRepay generating a new kernel each time, the only difference in each being the constant values. CuPy generates only one kernel and reuses it, passing in the scalar values as parameters. Work is ongoing into an heuristic that would disable this optimisation in such cases.

The `grayscott` benchmark is an interesting case where fusion results in poorer performance compared to CuPy. This particular benchmark is a stencil-like computation which performs operations on sub-arrays of a larger matrix. Our fusion algorithm produces a single huge kernel that takes over two hundred input arrays. We suspect the compilation time and the memory overhead result in a slowdown here.

The insights presented in Figure 5 and Figure 6 confirm that fusion is the reason for the performance benefits observed by DelayRepay compared to CuPy.

Table 3. End-to-end benchmark results

Benchmark	NumPy (s)	CuPy (s)	Bohrium (s)	DelayRepay (s)	Input Size (MB)
arc_distance	2.02	5.35	6.00	0.66	320.00
create_grid	0.68	0.57	3.36	0.62	0.00
cronbach	3.69	4.45	N/A	4.45	0.06
euclidean_distance_square	1.40	2.68	3.40	2.73	2400.00
evolve	2.08	2.50	14.67	1.48	3200.00
grayscott	2.56	45.44	10.92	42.95	2304.12
harris	2.81	3.10	3.41	1.53	800.00
l1norm	2.71	1.63	11.91	1.66	665.40
l2norm	0.98	2.01	6.34	1.48	0.00
laplacien	3.75	3.66	2.93	2.61	476.79
log_likelihood	7.75	4.92	2.01	1.15	7.18
lstsqr	1.87	3.96	64.55	3.88	1568.00
nn	0.96	5.10	1.86	6.91	800.00
pairwise	1.63	1.74	3.09	1.71	1536.00
pythagorean_id	3.62	2.10	7.47	0.60	808.00
repeating	2.33	0.62	32.00	0.58	1624.00
reverse_cumsum	2.06	1.56	2.23	1.56	105.71
rosen	1.70	3.11	1.83	1.10	1.36
specialconvolve	1.88	1.99	6.23	1.06	666.89
vibr_energy	3.85	2.50	5.48	0.57	80.00

6 Related Work

GPU Accelerated Drop-In NumPy Replacements. There are a number of related projects that act as a drop-in replacement for NumPy, utilizing the GPU to improve performance.

CuPy [20], introduced earlier in Section 2.1, is a drop-in replacement for NumPy on GPUs, and forms the basis for our work on DelayRepay. CuPy off-loads operations onto GPUs and supports multi-GPU execution, but without a drop-in NumPy interface. It requires programmers to manually manage multi-GPU data movement and synchronization.

Bohrium [15], introduced earlier in Section 2.2, is a drop-in replacement for NumPy that accelerates programs on GPUs by performing lazy evaluation of Numpy programs before fusing operators. Bohrium supports CPUs and GPUs.

Legate NumPy [3] by NVIDIA provides a drop-in replacement for GPUs using kernel fusions to accelerate NumPy code. It is built on the top of the Legion task-based runtime system, which achieves high performance and scalability on a wide range of supercomputers [4]. However, the implementation of Legate relies heavily upon the Legion specific programming model and runtime system [4] - decreasing its utility as a drop-in replacement for NumPy - whereas DelayRepay only requires the freely-available CuPy Python library and a standard CUDA installation. Legate NumPy also does not perform kernel fusion and has an eager evaluation model. Furthermore, Legate is aimed specifically at distributed computing workloads on high-performance computing clusters.

Kernel Fusion. There has been a large body of work looking into fusion of GPU kernels. Arash et al. [1] present an analytical model that considers input data characteristics and available GPU resources to estimate near-optimal settings for kernel launch parameters to optimize machine learning workloads. Kernel fusion can also reduce energy consumption and improve power efficiency on GPU architectures [28]. Both of these introduce kernel fusion methods to achieve higher hardware utilization and reduce the total energy optimization without performance loss, in the field of Machine Learning. In contrast, PyDelay tries to optimise NumPy programs by tracking arrays without programmer intervention, and is application agnostic.

Several domain specific languages aimed at performance support array operation fusion or loop fusion, including Delite [26], Accelerate [18], Build to Order BLAS [5], Firedrake [23] and Taskgraph [2].

There has also been work on fusion of array operations in non-GPU contexts. Single-Assignment C [24] and the work of Shei et al. on MATLAB [25] both use array operation fusion.

Much of this work is similar in spirit to earlier optimisation techniques such as loop fusion [13] and automatic parallelisation using the polyhedral model [8].

GPU Code Generation for Dynamic Languages. Besides simple wrappers such as PyOpenCL and PyCUDA [14] there exist more sophisticated GPU code generation systems. Numba [17] is a GPU code generator for Python that uses JIT compilation techniques, but users must be aware of the underlying GPU programming model. Harlan-J [22] is a JavaScript extension for data parallelism combined with an OpenCL JIT compiler. As with Numba, Harlan-J requires users to write specific parallel aware code for targeting the GPU. Similar to Numba, Copperhead [7] is another tool for accelerating Python code with GPUs. Copperhead requires the programmer to write GPU kernels in a restricted Python subset. AlPyNa [12], parallelises ordinary nested Python loops onto GPUs via Numba. This requires programmers to rewrite existing NumPy programs as loop-based Python.

Some higher level approaches have taken advantage of the parallel semantics of operations performed over arrays. Fumero et al. [10] describe an advanced GPU compilation system for the R programming language that makes use of the Graal virtual machine technology to generate GPU kernels at runtime based on dynamically captured traces. Also building on top of Graal, Tornado [9] provides GPU capabilities for Java programs at runtime.

Improving Dynamic Language Native Extension Performance. Although not aimed at GPUs, there has been work on improving the performance of calling native code from dynamic languages. SQPyte [6] improves performance of calling native database code from a Python implementation through the use of an embedded JIT compiler. Grimmer et al. [11] show significant improvement in Ruby native extension performance by combining a Ruby and C interpreter that share a common IR. Weld NumPy [21] uses the approach of a common runtime and intermediate representation to optimise calling between different languages and libraries.

7 Conclusion and Future Work

We have presented DelayRepay, a drop-in replacement for NumPy that implicitly accelerates Python code on GPUs. We have shown that our kernel fusion techniques enable DelayRepay to outperform our closest competitors, CuPy and Bohrium, by about 60% and 130% on average respectively, and by a maximum of 409% and 86% respectively. We have achieved this while only requiring one trivial code change for users of the original NumPy version, demonstrating that the significant performance gains available on GPUs can be made available to scientific Python programmers, at essentially no additional programming effort. We have shown that while DelayRepay was built for CUDA, its design allows for other backends to be plugged in.

There are many avenues of exploration and improvement. For example, our approach of building a NumPy expression tree coupled with dynamic information available at runtime allows potential optimisations, such as dead code elimination, to be made before generating the CUDA kernels. Similarly, we believe it will be interesting to investigate dynamic schedule ordering of kernels to exploit better locality properties. We believe it is worth investigating how this approach might improve NumPy performance on the CPU. Our lightweigth AST approach should compare favourably with existing approaches such as Weld. Finally, by further developing our analysis of the correlation between application properties and achieved speed-up, we aim to develop simple dynamic heuristics which can filter out cases where DelayRepay is not beneficial. For example, a heuristic that would break up a fused kernel when the number of input arguments is greater than a threshold value could improve the performance of the `grayscott` benchmark.

Acknowledgments

The authors would like to thank the reviewers for their helpful feedback. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), [NSERC RGPIN-2020-05889], the Canada CIFAR AI Chairs Program, and the Defense Advanced Research Projects Agency (DARPA) Software Defined Hardware (SDH) program.

References

[1] Arash Ashari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, and P. Sadayappan. On optimizing machine learning workloads via kernel fusion. In Albert Cohen and David Grove, editors, *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 173–182. ACM, 2015.

[2] Christian Batory, Lengauer Don, Charles Consel, and Martin Odersky. *Domain-Specific Program Generation.* Springer, 2004.

[3] Michael Bauer and Michael Garland. Legate NumPy: Accelerated and distributed array computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–23, 2019.

[4] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[5] Geoffrey Belter, Elizabeth R. Jessup, Ian Karlin, and Jeremy G. Siek. Automating the generation of composed linear algebra kernels. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*. ACM, 2009.

[6] Carl Friedrich Bolz, Darya Kurilova, and Laurence Tratt. Making an Embedded DBMS JIT-friendly. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:24, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[7] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In Calin Cascaval and Pen-Chung Yew, editors, *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pages 47–56. ACM, 2011.

[8] Paul Feautrier. Some efficient solutions to the affine scheduling problem. i. one-dimensional time. *Int. J. Parallel Program.*, 21(5):313–347, 1992.

[9] Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. Dynamic application reconfiguration on heterogeneous hardware. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution*

Environments, VEE 2019, page 165–178, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Juan José Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. Just-in-time GPU compilation for interpreted languages with partial evaluation. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8-9, 2017*, pages 60–73. ACM, 2017.

[11] Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. Dynamically composing languages in a modular way: supporting C extensions for dynamic languages. In Robert B. France, Sudipto Ghosh, and Gary T. Leavens, editors, *Proceedings of the 14th International Conference on Modularity, MODULARITY 2015, Fort Collins, CO, USA, March 16 - 19, 2015*, pages 1–13. ACM, 2015.

[12] Dejice Jacob, Phil Trinder, and Jeremy Singer. Python programmers have GPUs too: Automatic python loop parallelization with staged dependence analysis. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2019, page 42–54, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *Languages and Compilers for Parallel Computing, 6th International Workshop, Portland, Oregon, USA, August 12-14, 1993, Proceedings*, volume 768 of *Lecture Notes in Computer Science*, pages 301–320. Springer, 1993.

[14] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012.

[15] Mads RB Kristensen, Simon AF Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: Unmodified NumPy code on CPU, GPU, and cluster.

[16] Mads Ruben Burgdorff Kristensen, Simon Andreas Frimann Lund, Troels Blum, and James Avery. Fusion of parallel array operations. In Ayal Zaks, Bilha Mendelson, Lawrence Rauchwerger, and Wen-mei W. Hwu, editors, *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, PACT 2016, Haifa, Israel, September 11-15, 2016*, pages 71–85. ACM, 2016.

[17] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery.

[18] Trevor L. McDonell, Manuel M. T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional GPU programs. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 49–60. ACM, 2013.

[19] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

[20] Royud Nishino and Shohei Hido Crissman Loomis. CuPy: A NumPy-compatible library for NVIDIA GPU calculations. *31st Confernce on Neural Information Processing Systems*, page 151, 2017.

[21] 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)*, page 45, 2017.

[22] Uday Pitambare, Arun Chauhan, and Saurabh Malviya. Just-in-time Acceleration of JavaScript. Technical report, 2013.

[23] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. Firedrake: Automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software*, 43(3), December 2016.

[24] Sven-Bodo Scholz. Single assignment C: efficient support for high-level array operations in a functional setting. *J. Funct. Program.*, 13(6):1005–1059, 2003.

[25] Chun-Yu Shei, Adarsh Yoga, Madhav Ramesh, and Arun Chauhan. MATLAB parallelization through scalarization. In *15th Workshop on Interaction between Compilers and Computer Architectures, INTERACT 2011, San Antonio, Texas, USA, February 12, 2011*, pages 44–53. IEEE Computer Society, 2011.

[26] 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*, 13(4s), April 2014.

[27] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 2020.

[28] Guibin Wang, YiSong Lin, and Wei Yi. Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 344–350. IEEE, 2010.

[29] Haicheng Wu, Gregory Diamos, Jin Wang, Srihari Cadambi, Sudhakar Yalamanchili, and Srimat Chakradhar. Optimizing data warehousing applications for gpus using kernel fusion/fission. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 2433–2442. IEEE, 2012.