GPUMap: A Transparently GPU-accelerated Python Map Function

Ivan Pachev
California Polytechnic State University
San Luis Obispo, California, USA
ipachev@calpoly.edu

Chris Lupo
California Polytechnic State University
San Luis Obispo, California, USA
clupo@calpoly.edu

ACM Reference Format:
Ivan Pachev and Chris Lupo. 2017. GPUMap: A Transparently GPU-accelerated Python Map Function. In *PyHPC'17: PyHPC'17: 7th Workshop on Python for High-Performance and Scientific Computing, November 12–17, 2017, Denver, CO, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3149869.3149875

1 INTRODUCTION

GPU computing has been adopted by many programmers to solve a wide range of problems with high performance. Creating a GPU-accelerated application or refactoring an existing application to be GPU-accelerated is not always a trivial task. GPU programmers are often required to be familiar with the architecture of the GPU in order to create an efficient GPU program. For programmers without any understanding of GPU architecture, learning GPU programming in order to incorporate GPU-acceleration into their applications is time consuming and sometimes difficult.

There are some existing efforts to develop software packages that attempt to simplify GPU programming in higher-level languages such as Java and Python. They require programmers to be familiar with the traditional GPU programming model which involves some understanding of GPU threads, memory, and kernels. These software packages usually restrict the use of object-oriented programming. As a result, prior to using these software packages, programmers are required to transform the data they would like to operate on into arrays of primitive data.

Just as individual systems with GPU-computing capability have become more available, so too have high-performance distributed systems. A limiting factor in the computing power of many modern cluster computing frameworks is that, although they are scalable, the frameworks are typically constrained to CPU-exclusive distributed computing applications. This means that these frameworks may lack the ability to harness the powerful GPUs belonging to each system. Compute intensive tasks such as machine learning, image processing, and data analytics will benefit from distributed computation on systems with GPUs.

Building GPU-accelerated distributed applications is possible using CUDA bindings such as PyCUDA [3] and JCUDA [15]. However, these libraries provide almost no simplifications to GPU programming, requiring programmers to already be familiar with GPU programming prior to using them. Furthermore, requiring programmers to explicitly incorporate traditional GPU programming into their distributed applications conflicts with the purpose of the distributed computing frameworks, which is to abstract the underlying distributed system.

There are also software packages that provide simplified GPU acceleration to distributed computing frameworks such as MapReduce [1] and Spark [17]. Although these packages provide a simplified GPU programming experience, they do not do so in a completely abstracted manner.

The contributions of this paper are two-fold. The first contribution is GPUMap, a GPU-accelerated map function for Python. GPUMap hides all the details of the GPU from the programmer, and allows the programmer to accelerate programs written in normal Python code that operate on arbitrarily nested objects made up of primitive data using a majority of Python syntax. Using GPUMap, certain types of Python programs are able to be accelerated up to 100 times over normal Python code.

The second contribution is GPU-accelerated RDD (GPURDD), which is a Resilient Distributed Dataset (RDD) that is used with Spark. GPURDD incorporates GPUMap into its `map`, `filter`, and `foreach` methods in order to allow Spark applications to make use of the simplified GPU acceleration provided by GPUMap. These transformations can be used with normal Python functions and do not require objects in the RDD to be restructured in order to use them.

This paper is organized as follows. Section 2 discusses background information on GPU computing, Python, and Spark. Section 3 discusses related works that attempt to provide simplified GPU acceleration in both individual and cluster computing environments. Section 4 explains the design and implementation of both GPUMap and GPURDD. Section 5 analyzes performance benchmarks that use GPUMap and GPURDD, and Section 6 concludes and provides directions for future research.

2 BACKGROUND

This section discusses necessary background information related to GPUMap, the implementation of GPUMap itself, and testing GPUMap. Some basic background information on GPU computing, Python, and Spark is also presented.

PyHPC'17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5124-9/17/11…$15.00
https://doi.org/10.1145/3149869.3149875

2.1 GPU Computing

GPUs can be used as a powerful parallel processor that is capable of more floating point operations and has higher streaming memory bandwith than even the highest-end CPUs [7]. Although GPUs can be used as general purpose parallel processors, GPU programming usually focuses on quickly performing batches of numerical computations, rather than running full multithreaded, object-oriented programs. Developers must take into account the limitations of the GPU when writing optimized GPU programs.

The following is a discussion of some CUDA GPU-programming basics, as they are necessary in understanding the implementation details of GPUMap.

2.1.1 Thread Hierarchy. Threads are organized into a thread hierarchy [6] consisting of *blocks* and *grids*. Both of these hierarchical levels can be indexed in one, two, or three-dimensions. Each thread operates on a piece of data based on its position in the block and the position of the block in the grid.

2.1.2 Memory Hierarchy. There are three different types of memory on the GPU that are used for GPU computing: thread-local memory, shared memory, and global memory [6]. Thread-local memory is available for each thread and is used for the individual runtime stack of each thread. This memory stores local variables and information about function calls. Shared memory is larger than thread-local memory and is shared between threads in a block. The largest and slowest type of memory is global memory. Global memory can be accessed by any threads in any block and is the only GPU memory that is directly accessible to the host processor.

2.1.3 Serialization. Prior to running any GPU code, the data that will be operated on must be serialized into a stream of primitives. Space for this serialized data must be allocated in the GPU's global memory. Serialized data must be copied from the host to the allocated memory block on the GPU.

Once the execution on the GPU is complete, the data needs to be copied back to the host. Once the data is copied back to the host, the data must be deserialized and inserted into its originating data structures.

2.1.4 Kernels. The program to be run on the GPU is written in the form of a *kernel* [6]. The kernel is a function that is executed in parallel for each thread. A pointer to the data set to be operated on by the GPU is passed to the kernel function.

2.2 Python

Python is a general purpose, interpreted programming language that supports object-oriented programming, functional programming, and procedural programming. The language is dynamically typed, meaning that functions can be passed any types of arguments and can return any type. In addition, fields and variables can store data of any type. Python does this by performing late binding, which means that variable names are bound to their corresponding objects at runtime.

The following is a discussion of object structure in Python, Python's map operation, and Python closures as they are necessary in understanding the implementation details of GPUMap.

2.2.1 Object Representation. Python objects are stored in a hash table called a dictionary [10]. The keys to the dictionary are the names of the fields or methods of an object. The values of the dictionary are the objects or methods referred to by the keys.

When a field or method is accessed in Python, the name of the field or method is looked up in the object's dictionary at runtime. If the name belongs to a field, the object contained in the field is returned. All functions and methods are callable objects in Python, so when the name of a method is looked up in an object's dictionary, the callable object representing the method is returned. This callable object can either be called or passed around as a normal object.

2.2.2 Map Function. One of the functional programming components provided with Python is the `map` function. The `map` function provides an effective abstraction for transforming a list by applying a function to each element by accepting a function f and a list L, and applying f to each element in L to produce a new list.

In Python 3, the value returned by the `map` function is a generator. A generator is like an iterator, but the elements returned by the generator are lazily produced when the generator is iterated upon. Thus, when the map generator is iterated upon, f is applied to each successive element over L, returning the result.

2.2.3 Closures. Python closures are supported, allowing functions to refer to variables from an outer scope. Functions are objects that can be stored and called later, so function objects must store references to these outer scope variables for later access when the function object is called.

Python stores these references by creating a mapping between the variable names and the objects they refer to in the form of a dictionary. This dictionary can be accessed by using the `__clos__` field of a function object.

2.3 Apache Spark

Apache Spark is an open-source, cluster-computing framework that provides abstractions for the underlying distributed system that allow programmers to harness the power of the system without needing to understand how the system works [17].

Spark is efficient for iterative and interactive tasks due to the nature of its underlying data model, the resilient distributed dataset (RDD). A RDD is a read-only, fault-tolerant collection of items that is partitioned across a cluster and can be operated on in parallel [16]. Spark's workers are long-lived processes that keep RDD partitions persistent in memory between operations, allowing them to be easily reused in future operations [16].

Because RDDs are typically operated on in parallel, users should see performance increases provided by GPU-accelerated implementations of RDD operations on the worker nodes.

3 RELATED WORK

There are other projects that aim to provide simplified GPU programming to languages such as Java and Python, including Rootbeer, Aparapi, and Numba. These projects allow programmers to implement GPU algorithms and produce GPU-accelerated applications without needing to write any CUDA or OpenCL code. In addition, there are some projects that attempt to provide GPU-acceleration in both MapReduce and Spark, such as Multi-GPU

MapReduce, Hadoop+Aparapi, and Spark-Ucores. This section discusses these related projects.

3.1 Rootbeer

Rootbeer is a Java framework that aims to automate GPU programming steps such as serialization of the input data, creating and launching GPU kernels, and deserialization of the data back into the CPU memory. Rootbeer allows developers to write their parallel code in Java by implementing the `Kernel` interface which contains only one method: `gpuMethod`. Rootbeer automatically finds all fields that are reachable from the `Kernel` class and serializes them to the GPU [9].

Rootbeer allows usage of almost all Java features, including arrays, composite objects, dynamic memory allocation, strings, and exceptions. However, in order to implement a more complex application, some of the underlying CUDA functionality, such as thread configuration, shared memory and CUDA thread synchronization must be manually specified [9].

3.2 Aparapi

Aparapi is another Java library that attempts to automatically GPU-accelerate user-supplied Java code [2]. However, rather than translating the Java bytecode into CUDA, the bytecode is translated into OpenCL.

Aparapi only supports usage of primitives and allows operation on them using normal Java syntax, as well as many of the functions provided by `java.lang.Math` [2]. Aparapi does not support objects, static methods, recursion, overloaded methods, exceptions, dynamic allocation, synchronization, or switch statements [2].

3.3 Numba

Numba is a just-in-time (JIT) compiler for Python targeted towards scientific computing. Numba provides limited GPU-acceleration, allowing programmers to operate on NumPy arrays in parallel using the GPU [4]. GPU-acceleration with Numba is much easier than writing CUDA code because Numba provides considerable simplifications to the traditional GPU programming model.

Numba does not require programmers to be well-versed in CUDA programming. Instead, the code that is to be translated can be written in a subset of Python. The reason only a subset of Python is allowed is because Numba must be to be able to infer all types in order to generate the proper GPU code [4]. This means that object-oriented code cannot be used.

3.4 Multi-GPU MapReduce (GPMR)

Multi-GPU MapReduce is a flexible MapReduce implementation that works on clusters where the compute nodes have GPUs. Multi-GPU MapReduce outperforms the normal CPU implementation of MapReduce, as well as other single-GPU implementations of MapReduce [14]. Although the framework performs well and is flexible, Multi-GPU MapReduce does not provide a high-level abstraction for the underlying system, which may cause difficulty for users without GPU programming experience.

3.5 Hadoop+Aparapi

Hadoop+Aparapi provides an improved interface over traditional GPU programming [5]. The developers were not able to completely abstract the GPU component due to limitations in Aparapi when Hadoop+Aparapi was created. Thus, simplification of the Hadoop+Aparapi API was not possible. The Aparapi team has added features that may help simplify the Hadoop+Aparapi's API such as the usage of objects and lambda expressions in the GPU kernel code [2].

3.6 Spark-Ucores

Spark-Ucores also uses Aparapi to translate Java code into OpenCL[12]. The Spark-Ucores team has also forked Aparapi in order to provide support for FPGAs and APUs, resulting in the creation of Aparapi-Ucores [12]. Spark-Ucores provides GPU-accelerated implementations of a few parallel operations on RDDs, including `map`, `mapPartitions`, and `reduce`.

Spark-Ucores does not provide an abstraction for its GPU components, so programmers must have at least some GPU experience. In order to use Spark-Ucores, the user must restructure their existing Spark code, which may further discourage usage of Spark-Ucores.

4 IMPLEMENTATION

GPUMap is open-source software, and may be downloaded from the repository at: https://github.com/ipachev/py_gpumap.

The primary goal of GPUMap is to provide transparent GPU-acceleration, which involves automatic serialization, code translation, execution of the translated code, and deserialization. The programmer should be able to write normal Python code that provides a function f and a list L and call `gpumap(f, L)` to produce a list L', the same way that they would normally call `map(f, L)`.

4.1 Requirements

The implementation of GPUMap imposes the following requirements to support GPU execution.

- Objects must contain only integers, floating point numbers, booleans, or other objects.
- Objects of the same class must have the same fields and the same types in their corresponding fields, i.e. must be homogeneous.
- Objects cannot contain members of the their own class either directly or indirectly.
- Lists must contain only one class of objects.
- Functions or methods must be passed arguments of the same type every time they are called.
- Functions or methods must return the same type every time they are called.
- When a function is called, the function must call the same functions or methods every time.

4.2 Invocation

When the programmer calls `gpumap(f, L)`, the following steps are taken in order to perform the desired map operation:

(1) f is applied to the first element of L, L_0, to produce L'_0 and runtime inspection is performed to analyze every function call.

(2) The fields of L_0 and L'_0 are inspected to collect data about the classes of L_0 and L'_0.
(3) If f is a closure, any objects that are included in the closure bindings are also inspected and information is collected about their classes.
(4) CUDA C++ class definitions are created for the necessary classes by using the information collected during runtime inspection and object inspection.
(5) Any functions and methods, including constructors, that are called when applying f to L_0 are translated into CUDA C++.
(6) All of the elements of $L_{1...n}$ are serialized to the GPU. Any of the objects or lists that have closure bindings in f are also serialized.
(7) The map kernel, which includes all class, function, and method translations, is compiled and executed on the GPU, applying the translation of f, f', to each element in the serialized version of $L_{1...n}$.
(8) The serialized input list, $L_{1...n}$, and any closure objects or lists are deserialized and the data is re-incorporated into the original objects.
(9) The output list $L'_{1...n}$ is deserialized and is used to populate a list of objects based on the structure of L'_0.
(10) L'_0 is prepended to $L'_{1...n}$ to form L' as desired and L' is returned.

4.3 Implementation Details

There are many details and important algorithms used in the implementation of GPUMap. Some of the most pertinent steps are described here, but the full discussion of the implementation is beyond the scope of this paper. For full details of the implementation, the reader is referred to Reference [8], and the source repository on GitHub.

4.4 Runtime Inspection

Prior to doing any code translation or serialization, some data must be collected about the functions and methods that will need to be called, as well as the objects that will need to be operated on. This data is acquired through inspection of the fields of objects and tracing function execution when the given function is applied to the first element in the list. Runtime inspection consists of call and object inspection, which are performed to extract representations of functions, methods, and objects.

Several pieces of information need to be known about a class of objects in order to generate the appropriate CUDA class definitions and properly serialize objects of that class. This information is stored in a Class Representation.

The Class Representation is a recursive data structure that may contain multiple other Class Representations for each field. A Class Representation is extracted by examining all of the fields of a sample object, *obj*, by iterating through *obj*'s fields as a normal Python dict, using `obj.__dict__`. This dict contains a mapping from *obj*'s field names to the objects contained in those fields. For each entry in this dict, the field name is recorded, and a Class Representation is extracted and recorded for the object contained in the field. Figure 1 depicts a sample extraction of a Class Representation of an object of a class called `classA`.

4.5 Code Generation

In order to operate on a list L by applying a function f to each element in the list on the GPU, the necessary CUDA C++ class definitions and function/method definitions must be generated. This process is non-trivial, and requires the emulation of Python's pass-by-reference behavior by passing all objects as references to functions.

Classes, methods, and functions are generated from the data structures extracted from runtime inspection.

Built-in Functions. Support was added for built-in functions such as math functions, `len`, `print`, and others. Some of these built-in functions have existing counterparts in CUDA C++, such as the math functions. Other functions that do not have existing counterparts, such as `len`, are implemented in C++ and are supplied in a header during compilation. Due to the fact that the names of built-in Python functions do not always match up with built-in CUDA C++ functions, translating the names of built-in Python functions may be necessary.

4.6 Kernel Generation

The final step of code generation is finalizing the CUDA kernel function that will be executed on the GPU. The kernel function is a function that is executed by each GPU thread. In order to parallelize the map process, the GPU thread will apply the top-level function to a different list item.

4.7 Serialization

When calling `gpumap(f, L)` with a function f and a list L, L and any closure variables of f must be serialized and copied to the GPU. The list L and any closure variables are not cached on the GPU and must be serialized for every call to `gpumap`, although this may be addressed by future work to improve performance.

After the translated code is executed, L and f's closure variables must be copied back to the host and deserialized.

4.7.1 Serializing Objects. Prior to copying an object to the GPU, the object must be serialized into the proper format so that the object can be processed by the translated CUDA code. Serializing a Python object involves collecting all of its non-contiguous parts and collecting them in a contiguous section of memory as normal binary data, as depicted in Figure 2.

In order to collect all the data in an object, including the data in its nested objects, the object's fields can be recursively examined. The order in which the fields are accessed must be in the same order as the class definition that is created during the class generation phase.

4.7.2 Serializing Lists of Objects. The process for serializing an entire list is similar to serializing a single object so that objects, whether or not they originate from a list, can be accessed and manipulated the same way, using the same C++ class definition. The same is true for deserialization. However, in the case of deserializing the output list of the map operation, the objects do not yet exist in the Python code, so a slightly different approach must be taken that also involves object creation, so the data can be unpacked into objects.

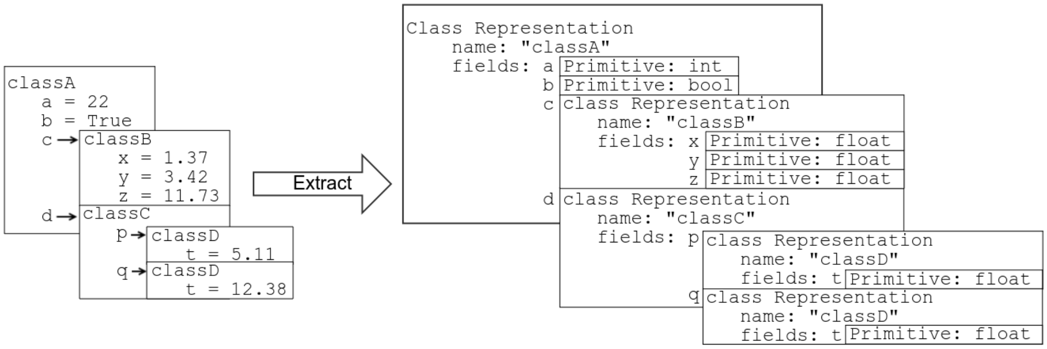

Figure 1: Sample Extraction of a Class Representation

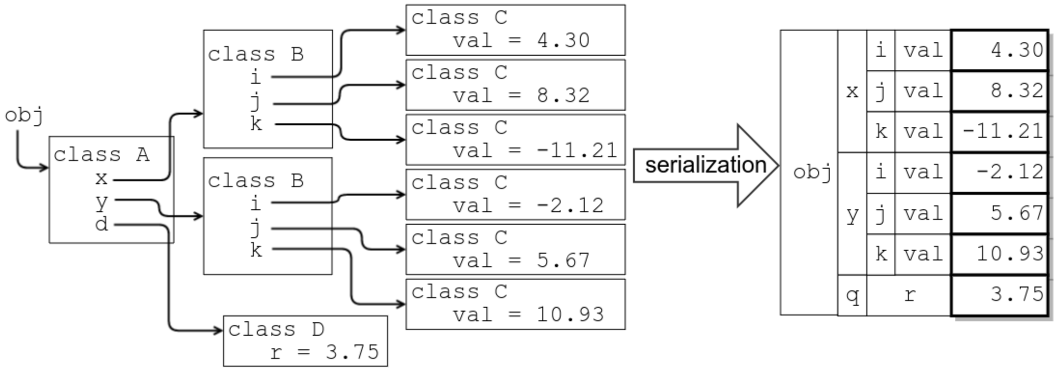

Figure 2: A normal Python object and its serialized counterpart

For each object to be instantiated in the output list, the first output object, which was created during runtime inspection, is deep copied and is used as a skeleton for the new object.

The object is then inserted into the output list. Once the correct number of objects have been created, populated, and inserted into the output list, the output list is returned.

4.8 Integration in Spark

GPUMap can be integrated in a variety of transformations and actions that can be performed on Spark RDDs. This section describes the implementation of GPURDD, which is a class that extends RDD and provides alternative implementations of map, filter, and foreach. The restrictions described at the beginning of this section regarding lists also extend to Spark RDDs.

4.8.1 Map. The `map` method on a Spark RDD allows the programmer to perform a transformation on the RDD by individually applying a function f to each element of the RDD to produce a new RDD containing transformed elements.

The existing implementation of RDD's `map` method defines a function g that takes an iterator of the input type of f and returns an iterator of the output type of f. Because f is stored in the closure bindings of g, f can be passed along with g. This closure g is then passed to `mapPartitions` so that f is applied to each element in a partition. The `mapPartitions` method accepts a function that takes an iterator of the input type and produces an iterator of the transformed type, making g an acceptable candidate.

The closure g that is implemented inside the body of `map` returns a map generator that applies the function passed to `map`, f, to each element returned by the iterator supplied to g. Map generators are created by using Python's built-in `map` function. Each time the map generator is iterated upon, the generator lazily applies f to a new element from the iterator passed to g and produces the output. In order to obtain an iterator to an entire partition, g is passed to `mapPartitions`, where g will be given the partition iterator. The call to `mapPartitions` returns a handle to an RDD that will eventually contain the items transformed using g, once the RDD is evaluated, and this is returned from the RDD's `map` method.

In order to incorporate GPUMap into GPURDD's `map` function, the implementation of g must be altered to create g'. The function g' must still take an iterator of f's input type and return an iterator of f's output type. The application of f on many items from the iterator must be evaluated immediately in parallel, rather than producing a map generator to evaluate the application of f to each element from an iterator in sequence. This means that access to all the elements in a partition simultaneously is necessary, which can be achieved by exhausting the iterator into a list. This list can then be passed into GPUMap, along with f, in order to apply f in parallel and produce a transformed list. Because GPUMap outputs a list and not an iterator, g' must return an iterator over the list, rather than just the list itself. The last step of GPURDD's map method is to return the return value of the call to mapPartitions, which is a handle to an RDD that will eventually contain the values that will have been transformed using g'.

Once an action is performed on the resulting GPURDD in order to evaluate all of the transformations, g' will be called with an iterator to the partition elements in order to transform them. Although the partition iterator is exhausted by g', Spark's lazy evaluation model is still preserved because g' is passed to mapPartitions. The mapPartitions method will only call g' when the time comes to evaluate the RDD. RDDs are evaluated when an action, such as collect, count, or foreach is performed on them.

4.8.2 Foreach. The foreach method on a Spark RDD allows the programmer to apply a function f to each element of an RDD without transforming the RDD. Instead, the foreach method is used to produce side-effects in the elements of the RDD.

The existing implementation of foreach makes use of the RDD's mapPartitions method, similarly to the way map method does. The foreach method simply defines a function g that iterates through a partition iterator, applying f to each item of the partition, ignoring the return value of the call to f. In order to comply with the fact that any function passed to mapPartitions must return an iterator, g returns an empty iterator. This function, g, is passed to mapPartitions, which creates a handle to an empty, dummy RDD. The reason an empty RDD is created is because g returns an empty iterator. When this dummy RDD is evaluated, f is applied to each element of the source RDD. In order to force evaluation of the dummy RDD, the foreach method calls the count method of this dummy RDD. The handle to the dummy RDD is not returned from foreach, as a handle to the dummy RDD is not useful.

GPUMap preserves side-effects, and can be effectively incorporated into foreach. The approach taken to incorporate GPUMap is similar to the approach taken with GPURDD's map method. A function g' is defined that takes an iterator over a partition and returns an empty iterator. The body of g' simply consists of exhausting the partition iterator into a list, and calling gpumap with f and the list created by exhausting the iterator. The return value of the call to gpumap can be discarded as it is not useful.

Then, g' is passed to mapPartitions to create a handle to a dummy RDD and, similarly to RDD's foreach method, evaluation of the dummy RDD is forced using the handle's count method in order to apply g' to each partition.

4.8.3 Filter. The filter method on a Spark RDD allows the programmer to transform an RDD by removing elements from the RDD by applying a function f to each element that returns a boolean indicating whether or not to keep the element.

The filter method is implemented very similarly to how map is implemented, incorporating the use of mapPartitions. This method defines a closure g that takes an iterator that provides elements of the RDD and returns an iterator that provides elements that did not get filtered. The closure g calls Python's built-in filter function with f to create an iterator that produces items from an iterable for which a f returns true and simply returns this iterator. Then, g is passed to mapPartitions, which provides g with an iterator over a partition, so that the elements of the partition can be filtered. An RDD handle is returned by the call to mapPartitions.

The purpose of incorporating GPUMap into GPURDD's filter method is to attempt to speed up the evaluation of f on each element of the partition. Due to the fact that when using GPUMap, the input list and output list must have a one-to-one correspondence, GPUMap cannot be directly used to filter the elements. However, the results of applying f to each item can be computed using GPUMap and can be subsequently used to remove elements.

In order to implement GPURDD's filter method, a function g' must be created to be used with mapPartitions, similar to GPURDD's implementation of map. First, the iterator passed to g' must be exhausted to produce a list of items that can be operated on in parallel. Then, gpumap is called with f and the list of items to produce a list of boolean values indicating whether or not to keep an entry in the list of items. Once the list of items and the list of booleans are available, Python's zip iterator can be used to provide tuples consisting of the item itself and its corresponding boolean. Then, Python's built-in filter function is used to create a filter iterator from the zip iterator. This filter iterator will not return tuples where the second field of the tuple, the boolean, is false. Then the tuples returned by the filter iterator can be converted back into items by using a map generator. A map generator is created by using Python's built-in map function that maps a tuple yielded by the filter iterator to the tuple's first field, which is the item itself. This map generator serves as an iterator over the filtered items and is returned by g'. This process is illustrated in Figure 3.

Then g' is passed to mapPartitions and the resulting RDD handle is returned. Once an action is performed on the resulting RDD, then the RDD will be evaluated and g' will be called on an iterator over each partition, as with GPURDD's implementation of map.

5 EXPERIMENTS

In order to determine the types of workloads that can be accelerated using GPUMap, performance benchmarks are performed. The tests used for performance benchmarking GPUMap are the n-body test and the bubble sort test. The tests used for performance benchmarking GPURDD are the shell sort test and the pi estimation tests. These benchmarks use a variety of algorithms with different time complexities, allowing us to examine the viability of GPUMap or GPURDD in these different scenarios.

The experimental setup consists of machines fitted with:

- Intel Xeon E5-2695 v3 CPU @ 2.30GHz
- NVIDIA GeForce GTX 980
- 32 GB Memory
- CentOS 7

The NVIDIA GeForce GTX 980 has 4GB of memory and 2048 CUDA cores running at 1126 MHz. The GTX 980 supports CUDA Compute Capability 5.2 which allows up to 1024 threads per block and a maximum one-dimensional grid size of $2^{31} - 1$. Although the maximum number of threads per block is 1024, the benchmarks all use a block size of $512 \times 1 \times 1$. The grid size is $\lceil n/512 \rceil \times 1 \times 1$ where n is the size of the input list. This configuration allows GPUMap to achieve an occupancy of 100%, meaning that all 2048 CUDA cores on the GTX 980 are able to be used concurrently.

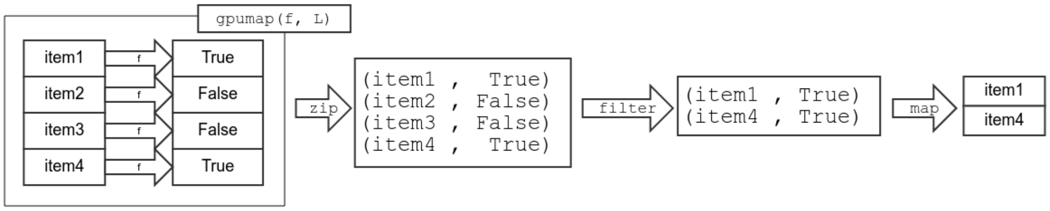

Figure 3: GPURDD filter method

Each machine shares the same GPU to run a graphical user interface as well as CUDA kernels. As a result, the maximum possible CUDA kernel runtime is constrained, which in turn limits the maximum possible data sizes that can be used.

In the cases where just GPUMap is benchmarked, only a single machine is used. In the cases where GPURDD is benchmarked, a Spark cluster consisting of 10 of these machines is used. The Spark cluster is configured so that each worker node only has one worker process and the worker process is only allowed to use one core. This configuration is used because there is only one GPU per machine and so executing GPUMap simultaneously from different processes is not possible.

The remainder of this section will discuss the different benchmarks and perform a performance evaluation. Overall, the results demonstrate that both GPUMap and GPURDD are not viable for $O(n)$ algorithms but provide considerable performance improvements to algorithms with larger time complexities.

5.1 N-body Benchmark

The n-body test is an all-pairs n-body implementation based on the outer loop approach from [13]. This test is used as a performance benchmark for GPUMap. This is an $O(n^2)$ algorithm due to the fact that on each step of the simulation, for each body, each other body must be considered.

Prior to running the simulation, a warmup on 256 bodies is performed using both Python's map and GPUMap. Then, the simulation is run starting with two bodies, all the way up to 8192 bodies by powers of two. For each number of bodies, the simulation is run five times and the average execution time is computed.

Figure 4 shows the speed-up achieved by using GPUMap over normal Python running the exact same code. With less than 256 bodies, there is no speed-up and GPUMap is not viable. However, starting with 1024 bodies, there is a considerable speed-up of slightly above 12 times. With 8192 bodies, the program is able to execute about 249 times faster.

The reason for this increase in performance is that because this all-pairs n-body simulation is an $O(n^2)$ algorithm, the amount of work needed to be done increases faster than the data that needs to be serialized for increasing body count. Thus, for larger numbers of bodies, the processing duration is not outweighed by serialization.

Figure 5 shows a breakdown of the run times of different stages of GPUMap for the different input data sizes. These stages are:

- First call, where runtime inspection is performed by applying the given function to the first item in the list.
- Code generation, where the appropriate CUDA code is generated and compiled using the information acquired from first call, as well as AST inspection.
- Serialize, where the closure variables as well as remaining objects in the list are inspected, serialized, and copied to the GPU.
- Run, where the CUDA kernel is executed in order to operate on the data in parallel.
- Deserialize, where the serialized data is copied back from the GPU and unpacked into its originating objects.

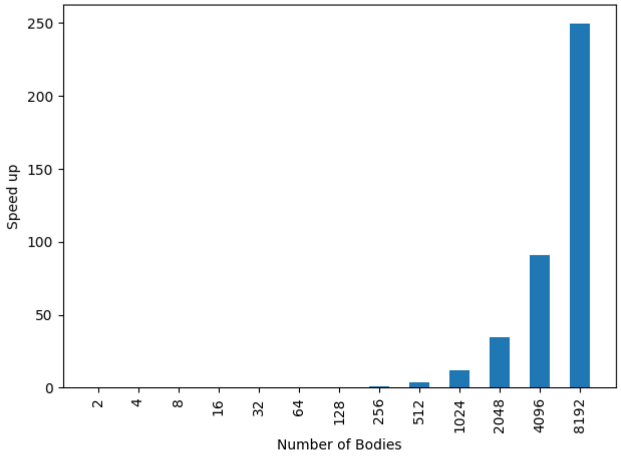

Figure 4: Speed-up using GPUMap with the N-body benchmark

As expected, due to the fact this algorithm is $O(n^2)$, the first call and run stages' durations increase much more quickly than the serialization and deserialization stages' durations with increases in body count. Serialization and deserialization durations increase with increases in body count, but not as quickly as first call and run durations due to the fact that serialization and deserialization are only $O(n)$. In addition, code generation seems relatively constant across changes in input data size because the code that is generated is independent of the input data size.

Overall, GPUMap is able to produce a considerable performance improvement of 249 times over the exact same n-body simulation code running through Python on the largest tested data set. However, GPUMap is not useful for n-body simulations with very small data sets as GPUMap actually causes a slowdown.

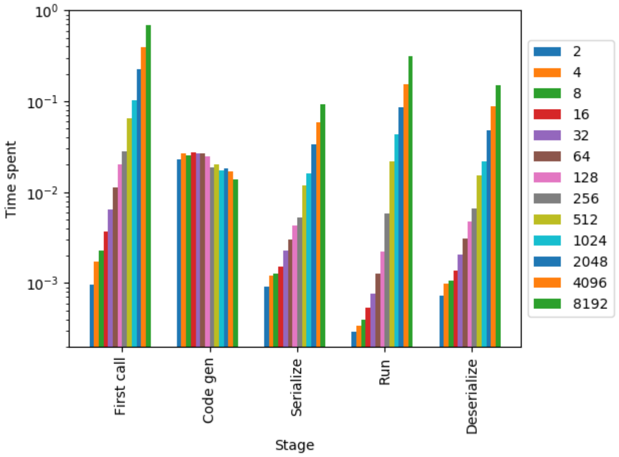

Figure 5: Stage duration using GPUMap with the N-body benchmark

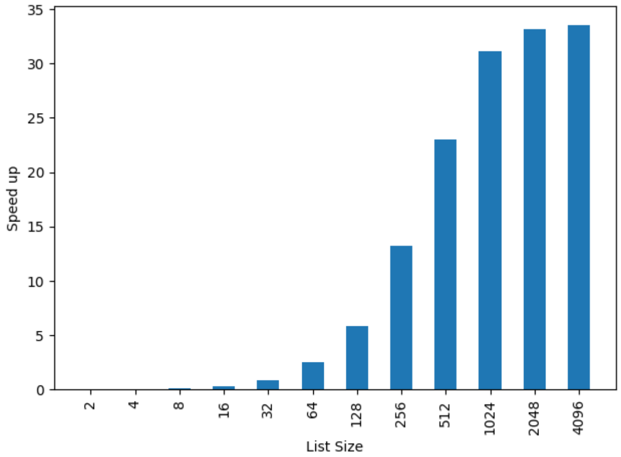

Figure 6: Speed-up using GPUMap to sort 1000 lists using bubble sort

5.2 Bubble Sort Benchmark

The bubble sort test is a simple test that involves sorting multiple lists of randomly generated integers in parallel using bubble sort, an $O(n^2)$ sorting algorithm. This test is also used as a performance benchmark for GPUMap.

When sorting multiple lists in parallel using the GPU, some lists may be more poorly sorted than others, causing certain threads take longer to complete. This means that the execution time of all the threads on the same block is always as long as the time taken to sort the most poorly sorted list, resulting in more consistent worst-case scenario performance for bubble sort.

Prior to performing the benchmark, a warmup is performed by using both Python's map and GPUMap to sort 1000 lists of 256 elements each. The benchmark consists of sorting 1000 lists of particular length, starting with lists of size 2 all the way to lists of size 4096 using both Python's map as well as GPUMap. Then another benchmark is performed that sorts 5000 lists of length from 2 to 4096 by powers of 2. For each data size, the benchmark is run 5 times and an average runtime is computed.

Figure 6 shows the performance increase that can be obtained from using GPUMap to sort 1000 lists in parallel with bubble sort over normal Python running the exact same code, sorting the same lists. For lists of size less than 32, GPUMap is not viable as the serialization duration outweighs the kernel runtime duration. For lists of size 32 or greater, GPUMap is able to outperform normal Python's map function. The largest speed-up, 33.5 times, is obtained for the lists of size 4096. However, this speed-up is not much greater than the 33.2 times speed-up for the lists of size 2048. The results show that with increasing list length, the speed-up converges asymptotically. One possible reason for this asymptotic convergence is due to the fact that sorting 1000 lists in parallel does not fully exploit the parallel nature of the GPU.

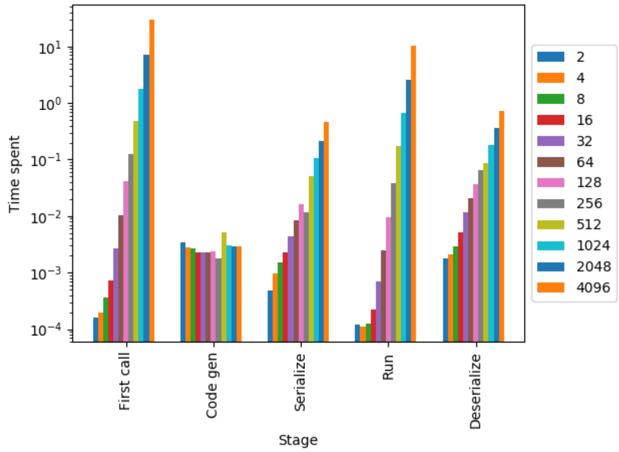

Figure 7: Stage duration using GPUMap to sort 1000 lists using bubble sort

Figure 7 shows a breakdown of the run times of different stages of GPUMap for the different input data sizes for the bubble sort benchmark. Similarly to the n-body benchmark, due to the fact that this is an $O(n^2)$ algorithm, the durations of the first call and run stages increase much quicker than the durations of the serialize and deserialize stages with increases in input data size. As previously mentioned, serialization and deserialization is $O(n)$, so the durations of these stages do not increase as quickly with increases in input data size. Furthermore, the duration of code generation is independent of the input data size as the same code is generated each time.

The GPU is capable of running more than 1000 threads simultaneously, so a second benchmark is performed that involves sorting 5000 lists rather than 1000 lists. The purpose of this benchmark is to examine different usage scenarios that may allow GPUMap to take better advantage of the parallel processing power of the GPU. The GPU is able to achieve a much better speedup when sorting these 5000 lists in parallel, as shown in Figure 8.

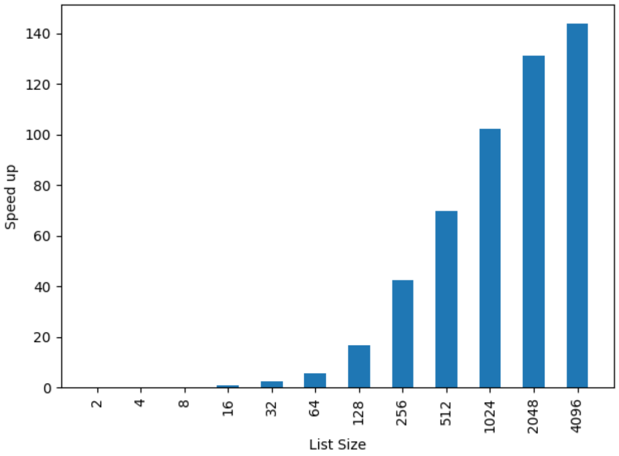

Figure 8: Speed-up using GPUMap to sort 5000 lists using bubble sort

With a larger number of lists to operate on in parallel, the GPU is able to use more of its available threads simultaneously. As a result, the maximum possible speed-up from increasing data size increases from about 34 times in the benchmark with 1000 lists, to about 145 times in the benchmark with 5000 lists.

Due to the fact that bubble sort is an $O(n^2)$ algorithm, using bubble sort with GPUMap to sort multiple lists simultaneously produces considerable performance improvements, as the processing time is not overshadowed by the serialization time. However, the performance improvement seems to be somewhat dependent on the data size, and unfortunately, reshaping the data into a shape that maximizes performance may not be possible.

5.3 Shell Sort Benchmark

The shell sort test benchmarks the sorting of multiple lists in parallel, similarly to the bubble sort. However, this test uses the shell sort algorithm, which has a worst case time complexity of $O(nlog^2n)$ [11]. In addition, this test designed for use with GPURDD rather than GPUMap, and the list count is varied rather than the list size. Varying the list count rather than the list size makes this parallel list sort an $O(m)$ where m is the number of lists.

Prior to running the benchmark, a warm up is performed using 100 lists of length 100 partitioned into 10 partitions. Then, the tests are run in order measure the performance of sorting 100, 1000, 10000, and 100000 lists of length 10000, partitioned into 10 partitions. The benchmark measures the time taken for the each set of lists to be sorted using both RDD's `foreach` method and GPURDD's `foreach` method. As shown in Figure 9, for this benchmark, GPURDD's `foreach` performs at least as well as RDD's `foreach` on each of these tests, with the exception of the smallest data set due to the added overhead of GPUMap. The sorting smallest data set, 100 lists of size 10000, results in poor performance because GPUMap is not able to take full advantage of the GPU as there are only 100 lists to sort, so only 100 GPU threads can be used. The most significant speed-up, 13 times, occurs when there are 10000 lists of size 10000 being sorted.

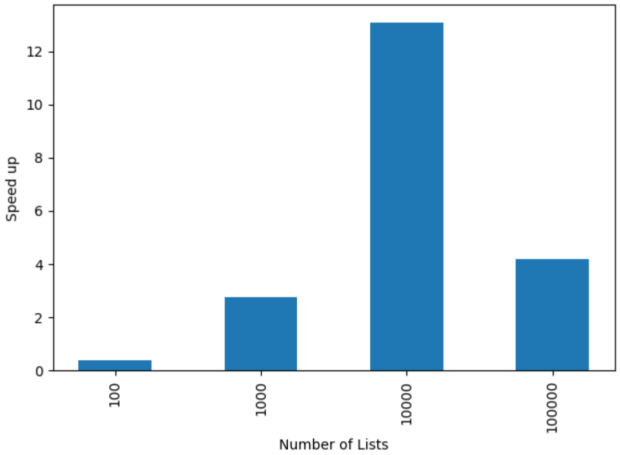

Figure 9: Speed-up using GPURDD to sort lists of size 10000 using shellsort

The reason this benchmark results in smaller speed-ups than the bubble sort benchmark is because this algorithm has better time complexity than bubble sort. In addition, the number of lists is varied rather than the number of elements in each list, causing larger numbers of lists to linearly increase the amount of work to do. This means that the kernel duration is not able to outweigh the first call and serialization stages' durations. For bubble sort's benchmarks involving larger data sets, the kernel had a longer duration than serialization, thus resulting in a larger impact on the overall duration. However, on the shell sort tests, the total duration was impacted by the first call, serialization, and deserialization stages more significantly than by the kernel run stage, as depicted in Figure 10.

Overall, GPURDD may provide a slight performance improvement over a normal RDD when using an algorithm such as shell sort with a large enough data set. However, performance improvement is dependent on the shape of the data, which can make depending on GPURDD for performance improvements less feasible. In addition, due to the fact that in this experiment the work to be done is $O(m)$ on the number of lists, the serialization costs cannot be outweighed with processing time.

6 CONCLUSIONS AND FUTURE WORK

This paper presents GPUMap, a Python map function that aims to allow programmers to GPU-accelerate certain types of programs with no extra effort or knowledge of GPU computing.. GPUMap works with normal, object-oriented Python code and does not require users to do any serialization or write kernels. This paper also presents GPURDD, which is a type of Spark RDD that incorporates GPUMap into its `map`, `filter`, and `foreach` methods to provide simplified GPU acceleration in Apache Spark.

GPUMap performs automatic serialization, automatic code generation, automatic kernel execution, and automatic deserialization in

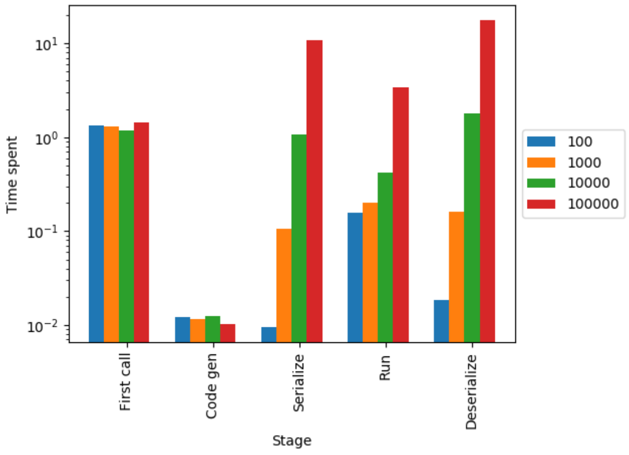

Figure 10: Stage duration using GPURDD to sort lists of size 10000 using shell sort

order to attempt to provide the programmer with a GPU-accelerated map function that does not require any extra effort to use.

GPUMap achieves a considerable performance improvement for certain types of programs. For compatible algorithms that have considerably larger time complexity than $O(n)$ and a large enough data set, GPUMap may provide performance improvements. During benchmarking of GPUMap using the $O(n^2)$ n-body algorithm, GPUMap was able to produce a speed up of 249 times on the largest data set. However, for algorithms with $O(n)$ time complexity or better, GPUMap will likely not yield any considerable speed-ups due to $O(n)$ serialization complexity.

GPURDD was created to incorporate the simplified GPU-acceleration provided by GPUMap into Apache Spark. GPURDD's `map` and `foreach` methods use GPUMap to apply a given function to each item in a partition. In the case of the `filter` method, GPUMap is used to apply the filtering function to each item in a partition to determine whether each element should be kept. The elements that should be not kept are then pruned outside of GPUMap.

6.1 Future Work

There are some Python language features, data structures, and built-in functions that are unsupported in code translated by GPUMap, primarily because GPUMap does not make use of CUDA's thread-level dynamic allocation as it does not perform well when many threads attempt to allocate memory simultaneously. This means that variable length data structures such as lists, dicts, and strings are unsupported by GPUMap. However, GPUMap supports limited usage of lists that are included as input list elements or closure variables. By using an alternative thread-level dynamic allocation scheme, it may be possible to incorporate dynamic allocation into GPUMap so that many more Python features can be implemented.

There are further performance improvements that can be made to GPUMap by caching input lists and closure variables and parallelizing the serialization process, which may help GPUMap perform better overall by decreasing the serialization time. Because GPURDD incorporates GPUMap, all of the limitations of GPUMap carry over to GPURDD.

REFERENCES

[1] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[2] Gary Frost and Mohammed Ibrahim. 2015. *Aparapi Documentation.* Technical Report. AMD Open Source Zone. https://aparapi.github.io/http://developer.amd.com/tools-and-sdks/open-source/

[3] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih. 2012. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Comput.* 38, 3 (2012), 157–174. https://doi.org/10.1016/j.parco.2011.09.001

[4] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM '15).* ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2833157.2833162

[5] Y. Lin, S. Okur, and C. Radoi. 2012. *Hadoop+Aparapi: Making heterogenous MapReduce programming easier.* Technical Report. hgpu.org. http://www.semihokur.com/docs/okur2012-hadoop_aparapi.pdf

[6] NVIDIA Corporation. 2017. *CUDA C Programming Guide.* Technical Report. NVIDIA Corporation. http://docs.nvidia.com/cuda/cuda-c-programming-guide

[7] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. 2008. GPU computing. *Proc. IEEE* 96, 5 (2008), 879–899.

[8] Ivan Pachev. 2017. *GPUMap: A Transparently GPU-Accelerated Map Function.* Master's thesis. California Polytechnic State University.

[9] Philip C Pratt-Szeliga, James W Fawcett, and Roy D Welch. 2012. Rootbeer: Seamlessly using GPUs from Java. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on.* IEEE, Liverpool, United Kingdom, 375–380.

[10] Python Software Foundation. 2017. *Data Model.* Technical Report. Python Software Foundation. https://docs.python.org/3/reference/datamodel.html

[11] Robert Sedgewick. 1996. Analysis of Shellsort and Related Algorithms. In *Proceedings of the Fourth Annual European Symposium on Algorithms (ESA '96).* Springer-Verlag, London, UK, UK, 1–11. http://dl.acm.org/citation.cfm?id=647906.739656

[12] Oren Segal, Philip Colangelo, Nasibeh Nasiri, Zhuo Qian, and Martin Margala. 2015. SparkCL: A Unified Programming Framework for Accelerators on Heterogeneous Clusters. *CoRR* abs/1505.01120 (2015). http://arxiv.org/abs/1505.01120

[13] Artjoms Šinkarovs, Sven-Bodo Scholz, Robert Bernecky, Roeland Douma, and Clemens Grelck. 2014. SaC/C formulations of the all-pairs N-body problem and their performance on SMPs and GPGPUs. *Concurrency and Computation: Practice and Experience* 26, 4 (2014), 952–971.

[14] J. A. Stuart and J. D. Owens. 2011. Multi-GPU MapReduce on GPU Clusters. In *2011 IEEE International Parallel Distributed Processing Symposium.* 1068–1079. https://doi.org/10.1109/IPDPS.2011.102

[15] Yonghong Yan, Max Grossman, and Vivek Sarkar. 2009. JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA. In *Euro-Par 2009 Parallel Processing: 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings.* Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 887–899. https://doi.org/10.1007/978-3-642-03869-3_82

[16] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12).* USENIX, San Jose, CA, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[17] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. *HotCloud* 10 (2010), 10–10.