GPUMAP: A TRANSPARENTLY GPU-ACCELERATED PYTHON MAP FUNCTION

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Ivan Pachev

March 2017

COMMITTEE MEMBERSHIP

TITLE:	GPUMap: A Transparently GPU-accelerated Python Map Function
AUTHOR:	Ivan Pachev
DATE SUBMITTED:	March 2017
COMMITTEE CHAIR:	Chris Lupo, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	John Clements, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Lubomir Stanchev, Ph.D. Professor of Computer Science

ABSTRACT

GPUMap: A Transparently GPU-accelerated Python Map Function

Ivan Pachev

As GPGPU computing becomes more popular, it will be used to tackle a wider range of problems. However, due to the current state of GPGPU programming, programmers are typically required to be familiar with the architecture of the GPU in order to effectively program it. Fortunately, there are software packages that attempt to simplify GPGPU programming in higher-level languages such as Java and Python. However, these software packages do not attempt to abstract the GPU-acceleration process completely. Instead, they require programmers to be somewhat familiar with the traditional GPGPU programming model which involves some understanding of GPU threads and kernels. In addition, prior to using these software packages, programmers are required to transform the data they would like to operate on into arrays of primitive data. Typically, such software packages restrict the use of object-oriented programming when implementing the code to operate on this data. This thesis presents GPUMap, which is a proof-of-concept GPU-accelerated map function for Python. GPUMap aims to hide 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 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.

There are also software packages that provide simplified GPU acceleration to distributed computing frameworks such as MapReduce and Spark. Unfortunately, these packages do not provide a completely abstracted GPU programming experience, which conflicts with the purpose of the distributed computing frameworks: to abstract the underlying distributed system. This thesis also presents GPU-accelerated RDD

(GPURDD), which is a type of Spark Resilient Distributed Dataset (RDD) which incorporates GPUMap into its `map`, `filter`, and `foreach` methods in order to allow Spark applicatons to make use of the abstracted GPU acceleration provided by GPUMap.

ACKNOWLEDGMENTS

Thanks to:

- My parents, Boian and Magdalena for being extremely supportive during my entire experience at Cal Poly.
- Roxanne and Moose, for helping me stay motivated.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

As general purpose GPU (GPGPU) computing becomes more popular, it will be adopted by more programmers and be used to tackle a wider range of problems. However, creating a GPU-accelerated application or retrofitting an existing application to be GPU-accelerated is not a particularly trivial task. Due to the current state of GPGPU programming, programmers are typically required to be somewhat familiar with the architecture of the GPU in order to effectively create a parallel program that runs on the GPU. For programmers without any understanding of GPU architecture, learning GPGPU programming in order to incorporate GPU-acceleration into their applications may be time consuming.

Programmers who program in lower-level languages may be familiar with concepts like threading and memory management. However, when programming a GPU, there are considerably more intricacies to understand regarding threading and memory management than when programming a CPU-only application. Programmers must understand how threads are structured in thread hierarchies, how to synchronize threads, how shared and global memory are managed, and how to correctly and effectively serialize data.

Programmers who are only familiar with higher-level languages may need to spend even more time and effort to become proficient enough in GPGPU programming to effectively GPU-accelerate applications. Fortunately, there are software packages that attempt to simplify GPGPU programming in higher-level languages such as Java and Python, such as Aparapi [3], Rootbeer [12], and Numba [7]. However, these software packages do not attempt to completely abstract away GPGPU programming. Instead, they also require programmers to be somewhat familiar with the traditional GPGPU programming model which involves some understanding of GPU threads and kernels. Typically, such software packages restrict the use of object-oriented programming when implementing the code to operate on this data. 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.

In order to attempt to abstract GPU programming entirely, this thesis presents GPUMap, a GPU-accelerated map function for Python. GPUMap aims to hide 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.

However, GPUMap has a variety of limitations that make it less flexible than Python's built-in map function. These limitations are primarily caused by the fact that Python is dynamically typed, making it difficult to collect information during runtime analysis. Another considerable source of GPUMap's limitations is that GPUMap does not incorporate the usage of thread-level dynamic allocation as the existing CUDA dynamic allocation scheme does not perform well when threads attempt to allocate memory simultaneously [5]. Because GPUMap does not incorporate dynamic allocation, many built-in Python data structures and functions are not available for use in code that is run using GPUMap.

As the demand for high-performance distributed systems increases, existing cluster computing frameworks will need to be upgraded so that these frameworks can keep up with the future's most demanding problems. A severely 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 currently lack the ability to harness the powerful GPUs belonging to each system. Due to the parallelizable nature of tasks commonly performed using distributed computing, some parts of the tasks may be parallelized even further to be performed on a GPU, allowing demanding tasks such as machine learning [1], image processing, and data analytics to be performed more quickly and with greater efficiency.

Fortunately, although GPU-acceleration is not built in to these frameworks, building GPU-accelerated distributed applications that run on these frameworks is possible using CUDA bindings such as PyCUDA [6] and JCUDA [19]. However, these libraries provide almost no simplifications to GPU programming, requiring programmers to already be familiar with GPGPU programming prior to using them. Furthermore, requiring programmers to explicitly incorporate traditional GPGPU programming into their distributed applications conflicts with the purpose of the distributed computing frameworks, which is to abstract the underlying distributed system.

Luckily, there are also software packages, such as HadoopCL [4], Hadoop+Aparapi [9], Spark-Ucores [16], and HeteroSpark [8] that provide simplified GPU acceleration to distributed computing frameworks such as MapReduce [2] and Spark [21]. Unfortunately, although these packages provide a simplified GPGPU programming experience, they do not do so in a completely abstracted manner, which also conflicts with the distributed framework's goal to abstract the underlying distributed system. This thesis also presents a GPU-accelerated RDD (GPURDD), which is a type of 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.

The remainder of this thesis is structured as follows. Chapter 2 discusses background information on GPGPU computing, Python, and Spark. Chapter 3 discusses related works that attempt to provide simplified GPU acceleration in both normal and cluster computing environments. Chapter 4 explains the design and implementation of both GPUMap and GPURDD. Chapter 5 describes how testing was performed on GPUMap and GPURDD. Chapter 6 presents and analyzes performance benchmarks performed on GPUMap and GPURDD. Chapter 7 elaborates on the limitations of GPUMap. Chapter 8 discusses improvements that may further benefit GPUMap, including methods to address some of GPUMap's limitations. Chapter 9 concludes this thesis and provides a summary of the results.

Chapter 2

BACKGROUND

Prior to discussing work related to GPUMap, the implementation of GPUMap itself, and testing GPUMap, some background information must first be discussed. This chapter provides some basic background information on GPGPU computing, Python, and Spark.

2.1 GPGPU Computing

In the past 12 years, the GPU has evolved from being used just as a graphics processor to being 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 [11]. Nowadays, GPUs are used as general purpose, programmable processors that are used for solving parallelizable problems that have large computational requirements [11]. Although GPUs can be used as general purpose parallel processors, GPGPU programming usually focuses on quickly performing batches of numerical computations, rather than running full-fledged, multithreaded, object-oriented programs.

As GPGPU computing has evolved, the process for programming the GPU has become drastically simplified. The programmer is no longer required to use the graphics pipeline to perform calculations. Instead, the programmer simply defines the program as a set of parallel threads that run the same program to operate on different pieces of data using a single-instruction-multiple-data (SIMD) model [11]. Using this simplified programming model allows programmers to succinctly express, as well as drastically speed up, various parallelizable operations such as map and reduce. In addition, as

GPGPU computing has become easier and more popular, GPUs have been shown to excel in tasks like sorting, searching, database querying, and linear algebra. Nowadays, GPGPU computing is used to solve some of humanity's most complex problems, such as problems involving molecular dynamics [11].

Unfortunately, because GPU threads are not fully featured CPUs, developers must take into account the limitations of the GPU when writing optimal GPGPU programs. GPU threads are simplified threads that can only perform branching, loops, and mathematical operations on primitive data [11]. Due to their simplicity, GPU threads do not perform complex branch prediction and so they may not perform as well in applications that make use of complex out-of-order execution. Furthermore, out-of-order execution also can cause performance decreases when threads attempt to run different instructions concurrently, due to the fact that groups of GPU threads share the same instruction fetch hardware [12]. Because the instruction fetch hardware is shared between threads that are trying to execute different instructions, diverging threads must wait to use the instruction fetch hardware, which results in decreased performance. In addition to sharing the same instruction fetch hardware, GPU threads share the same memory management hardware, making effective memory management slightly more difficult, as memory is fetched in batches and then chunks are delivered to the respective threads [10].

The remainder of this section will discuss some CUDA programming basics, as they are necessary in understanding some of the implementation details of GPUMap.

2.1.1 Thread Hierarchy

In order to organize groups of threads, the threads are organized into a thread hierarchy [10]. The first level of the thread hierarchy is that threads are organized into blocks. Threads can be given a one, two, or three-dimensional index in the thread block. The second level of the thread hierarchy is that blocks can be organized into a one, two, or three-dimensional grids. Typically, the dimensionality of the block and the grid is dependent on the type of problem being solved. Each thread picks a piece of data to operate on based on its position in its block and the position of its block in the grid.

2.1.2 Memory Hierarchy

There are three different types of memory on the GPU that are used for GPGPU computing: thread-local memory, shared memory, and global memory [10]. Thread-local memory is located on 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. Because the shared memory is local to a block, it is faster than global memory. However, shared memory can only be accessed by threads in the block, which means that storing data in shared memory is only useful when the data is only relevant to the block. The largest and slowest type of memory is global memory. Global memory can be accessed by any threads in any block and is accessed in batches when multiple threads request access to global memory simultaneously.

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. Then, space for this serialized data must be allocated in the GPU's global memory. This serialized data must then be copied to the allocated memory block.

Once the program's execution is complete, the data will need to be copied back to the host. The data is copied from the location of the memory block. 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 [10]. The kernel is a C function that is executed in parallel on each thread. Typically a pointer that points to the entire data set that is to be operated on by the GPU is passed to the kernel function. Then, each thread must identify the piece of data to operate on using it's block indices and grid indices. Once the thread's work is complete, the thread should write its results back to the data set in global memory.

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 remainder of this section will discuss object structure in Python, Python's map operation, and Python closures as they are necessary in understanding some of the implementation details of GPUMap.

2.2.1 Object Representation

In Python, objects are stored in a hash table, which is called a dictionary [13]. 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. However, because in Python, all functions and methods are callable objects, 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.

Due to the fact that objects are implemented as dictionaries, an object's fields and methods can be quickly and easily examined and extracted.

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 (or any iterable) L, and applying f to each element in L to produce a new list.

In Python 3, the value returned by this `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

In Python, closures are supported. This means that when functions are defined, they can refer to variables from an outer scope. Due to the fact that functions are objects and can be stored and called later, function objects must store references to these variables, so they can later be accessed when the function object is called.

In order to store these references, Python creates 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. If this dictionary is empty, there are no closure bindings. If there are entries present, the names of the variables and the objects they refer to can be accessed.

2.3 Apache Spark

Apache Spark is an open-source cluster-computing framework that provides useful abstractions for the underlying distributed system, allowing programmers to easily harness the power of a distributed system without needing to understand how the system works [21]. Spark provides similar features as Hadoop's MapReduce, including as usage, scalability, and fault tolerance. Spark provides commonly used parallelizable operations such as mapping, filtering, reducing, and sorting [20]. However, Spark's programming model provides more flexibility than MapReduce's model. In addition, for certain types of tasks, Spark performs much better than Hadoop, including iterative tasks such as machine learning and data mining as well as interactive tasks such as database queries [21].

Spark is able to outperform MapReduce 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 [20]. Spark's workers are long-lived processes that can persist RDD partitions in memory between operations, allowing them to be easily reused in future operations [20]. By retaining RDDs in memory, Spark is able to reduce disk I/O, network I/O, and serialization, improving the performance of iterative and interactive tasks. In contrast, MapReduce writes the output of a job to the distributed filesystem [2]. When the next job begins, the data must be reloaded from the distributed filesystem, causing MapReduce to perform very poorly when multiple jobs are chained together.

In addition to performance benefits, Spark's RDDs also provide fault tolerance. Fault tolerance is achieved because an RDD contains information about how the RDD was computed from existing RDDs, also known as its lineage. Storing the lineage of an RDD allows partitions of the RDD to be easily recomputed from existing RDDs or from the data on the distributed filesystem [20].

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

Chapter 3

RELATED WORK

There have been a few projects that aim to provide simplified GPGPU programming to languages such as Java and Python, including Rootbeer, Aparapi, and Numba. These projects allow programmers to implement GPGPU 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 chapter discusses these related projects.

3.1 Rootbeer

GPGPU applications are commonly programmed in CUDA or OpenCL. However, in order to use these frameworks effectively, the programmer must have some understanding of the underlying GPU [12]. Because writing GPU programs is very different from writing conventional programs, adding GPU acceleration can require a complete restructuring of the existing program.

Rootbeer is a Java framework that aims to automate tedious and time-consuming GPGPU programming steps such as serialization of the input data, creating and launching GPU kernels to operate on the data, and deserialization of the data back into the CPU memory [12]. Rootbeer allows developers to simply 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 uses all available CPU cores to serialize them to the GPU [12].

Rootbeer allows usage of almost all Java features, including arrays, composite objects, dynamic memory allocation, strings, and exceptions. Unfortunately, usage of features such as dynamic memory allocation and exceptions is discouraged by the creator of Rootbeer, as such features may cause significant decreases in performance. In addition, 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 [12].

Rootbeer uses Soot, a Java optimization framework, to inspect the Java bytecode by creating an intermediate representation called Jimple. Inspecting the Jimple intermediate representation allows Rootbeer to boil down complex objects into primitives and operations on these primitives that can be used effectively with the GPU [12]. This data is then used by Rootbeer to generate appropriate CUDA code to achieve the desired functionality that was programmed in Java. This CUDA code is then compiled using CUDA's nvcc, which can then be packed into the jar file containing the entire program [12].

Once the program is actually run and the program must begin to execute the GPU code, the Rootbeer runtime first performs high-performance serialization. Rootbeer's high-performance serialization consists of using custom bytecode to access and collect object fields, rather than resorting to reflection. Then, once all the data is collected and copied to the GPU, the Rootbeer runtime executes the compiled CUDA kernel that is stored in the jar. Once the operation is complete, the Rootbeer runtime automatically deserializes the results so they can once again be used by normal Java code [12].

Overall, Rootbeer provides support for many of Java's features on the GPU, and for many of these features, Rootbeer provides a considerable performance improvement. In addition, Rootbeer provides good support for automatic serialization and deserialization, allowing entire objects to be serialized. However, Rootbeer does not make an effort to abstract all of the concepts of GPU programming. In addition, incorporating Rootbeer into a distributed computing framework such as MapReduce or Spark may be difficult, as building a Rootbeer-enabled application is not very straightforward.

3.2 Aparapi

Aparapi is also a Java library that attempts to automatically GPU-accelerate user-supplied Java code [3]. However, rather than translating the Java bytecode into CUDA, the bytecode is translated into OpenCL.

Unfortunately, Aparapi works on a much smaller subset of Java code than Rootbeer, but aims to provide reasonable performance for all of the supported features. Currently, 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` [3]. Aparapi does not support objects, static methods, recursion, overloaded methods, exceptions, dynamic allocation, synchronization, or switch statements [3]. Part of the reason for these limitations is that OpenCL is less flexible than CUDA, as OpenCL does not allow recursion, function overloading, or dynamic allocation.

In order to use Aparapi, the programmer must first determine which primitive data is necessary to operate on. Then the programmer must then extract this data from their objects and collect the data into arrays of primitives. Once code has been written to prepare the data, the programmer needs to define a class that extends the `Kernel` class, which requires the programmer to implement one method: `run` [3]. In this run method, each thread must determine which piece of data to operate on by computing its thread index. Then, when writing code to launch the kernel, the programmer must specify the number of threads to use for this operation, also known as the execution range [3]. After writing code to launch the kernel, the programmer must provide code that inserts the data contained in the arrays of primitives back into its originating objects.

Aparapi, like Rootbeer, generates the GPU code prior to runtime and then packages the compiled GPU program into a jar file. Then, in order to execute the OpenCL code at runtime, Aparapi has a small layer written in C++ that accepts data from the Java code through JNI, copies data to the GPU, and executes the OpenCL kernel [3]. Once the kernel execution is complete, the data is copied back from the GPU and passed back through JNI back to the Java program. Once the data arrives back to the Java program, the data is inserted back into its originating arrays.

Aparapi provides a considerable performance improvement by allowing Java programmers to GPU-accelerate their data-parallel code. However, the Java features that can be used with Aparapi are very limited, allowing only usage of arrays of primitives. Aparapi provides automatic serialization and deserialization of these primitive arrays, but usage of Aparapi is still somewhat difficult because the data must be arranged in arrays of primitives by the programmer.

3.3 Numba

Numba is a just-in-time (JIT) compiler for Python targeted towards scientific computing [7]. In addition to bringing JIT compiling functionality to Python, Numba also provides limited GPU-acceleration, allowing programmers to operate on NumPy arrays in parallel using the GPU [7]. Using GPU-acceleration through Numba is much easier than writing CUDA code because Numba provides considerable simplifications to the traditional GPGPU programming model.

In order to JIT compile the translated Python code, Numba makes use of LLVM, a compiler with an effective JIT API as well as CUDA support. The goal of Numba is to convert the Python bytecode into LLVM IR. However, converting the bytecode into LLVM IR is a difficult process because many commonly used instructions in the Python bytecode do not map directly to instructions in the LLVM IR. Furthermore, the Python interpreter is a stack-based machine, while LLVM uses registers, adding to the difficulty of translation [7].

Numba begins by inspecting and analyzing the Python bytecode. The goal of this analysis is to extract the control flow information and perform a stack-to-register mapping of the bytecode. Once this analysis is complete, the bytecode is converted into an intermediate representation (IR) called the Numba IR [7]. If inferring every type in the Numba IR is possible, then the Numba IR is converted into LLVM IR and compilation is performed. However, due to the fact that Python is dynamically typed, not all types can be inferred. If Numba cannot infer all types, then the code is run through the Python C-API. This means if a function is intended to run on the GPU, all its types must be inferable or determined at runtime.

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 [7]. This means that object-oriented code cannot be used. Another simplification provided by Numba is automatic serialization and deserialization of NumPy arrays between the host and the GPU [7]. However, manual memory management is recommended in order to improve performance. Unfortunately, not all of the GPGPU constructs are obscured, requiring programmers to access the input data using thread indices and requiring programmer to specify kernel launch parameters when the GPU function is called [7].

Both Rootbeer and Aparapi compile the GPU code in advance, but Numba performs JIT compilation. This means that the Python bytecode must be inspected and translated at runtime, which adds to the work that Numba needs to do while the program is running. However, Numba still manages to provide a considerable performance improvement for code that operates on arrays. In addition, Numba provides a decent interface for Python programmers to interact with the GPU by performing automatic serialization and code translation. However, Numba does not completely abstract the concepts of GPU programming. In addition, Numba is very restrictive in terms of the code that can be GPU-accelerated as object oriented code is not allowed.

3.4 Multi-GPU MapReduce (GPMR)

Although GPUs have been shown to be capable of larger throughput than conventional CPUs, many powerful tool sets and APIs are still only implemented for use with CPUs [18]. As a result, many large-scale solutions to complex problems are bound by the performance provided by CPUs. One such CPU-bound tool set is MapReduce, a popular model used with cluster computing that provides an abstraction for the underlying distributed system.

MapReduce is a simple three-step abstraction that is not difficult to implement on a CPU: Take a set of items and map them in order to create key-value pairs, then sort the pairs by key in preparation for the reduce phase, the reduce the key-value pairs by combining all like-keyed pairs [18].

Using Multi-GPU MapReduce, these three steps also occur, but the data must first be copied into the GPU memory, then a GPU program must be executed to process the data, and then the output data must be copied back from the GPU memory. During the map and reduce phases, each item is assigned to a GPU thread to be processed in parallel. A group of items that is processed in parallel is called a chunk. Finally, after the map and reduce phases are complete the results must be copied from the GPU memory back into the main memory [18]. However, this approach runs into two scalability problems: the first problem is that the data set may exceed the GPU memory and the second problem is that this approach does not scale across nodes. In order to address these problems, the framework must partition both the input data and the map output appropriately [18].

The authors claim that communication, over the network and over the PCI-E bus, is typically the bottleneck. Thus, GPMR includes three optimizations that reduce PCI-E communication and network transfer: combination, partial-reduction, and accumulation. The combine step generates a single value per key to be sent to the reducers, rather than an entire set of key-value pairs. Because the input is divided into chunks, GPMR also includes an optional partial-reduction step which reduces GPU-resident key-value pairs after they are mapped, increasing the amount of work done by the GPU during the map phase, but decreasing PCI-E communication and network transfer time. The final optimization is accumulation. When subsequent chunks are mapped, they are combined with the previous mapped chunk's key-value pairs in GPU memory in order to decrease the number of key-value pairs transferred over the PCI-E bus [18].

The creators of GPMR have successfully created a flexible MapReduce implementation that works on normal clusters fitted with GPUs and have demonstrated that their implementation outperforms the normal CPU implementation of MapReduce, as well as other single-GPU implementations of MapReduce [18]. However, although the framework performs well and is flexible, GPMR does not provide a very good abstraction for the underlying system, which may cause difficulty for users without GPU programming experience.

3.5 Hadoop+Aparapi

Due to the increasing popularity of retrofitting existing distributed computing frameworks to support GPU acceleration, there are several GPU-accelerated implementations of Hadoop's MapReduce. One such implementation is Hadoop+Aparapi. The goal of Hadoop+Aparapi is to provide GPU acceleration for MapReduce while attempting to hide the complexity associated with GPGPU programming [9]. In order to provide support to OpenCL compatible devices, Hadoop+Aparapi leverages Aparapi, a convenient library that converts Java bytecode to OpenCL at compile time and executes the OpenCL binary on the GPU at runtime. However, due to limitations in OpenCL, Aparapi places heavy restrictions on the code that can be successfully converted into OpenCL [9].

Hadoop+Aparapi provides an API in the form of the `MapperKernel` abstract class. Users must override the `MapperKernel` class by defining three methods. The first method, `preprocess`, requires the user to convert the input key-value pairs into primitive data that can be operated on by the GPU. The second method, `gpu`, requires the user to define the operations that need to be performed by the GPU on the primitive input data. The third method, `postprocess`, requires the user to define how to convert the primitive output data that was created by the GPU into the desired output format [9].

Using this method, Hadoop+Aparapi was able to achieve a considerable speed-up over conventional MapReduce that was demonstrated by a benchmark consisting of the N-body simulation. The benchmarks for more complicated N-body simulations showed that Hadoop+Aparapi was able to run the simulation up to 80 times faster than Hadoop's MapReduce [9].

Although Hadoop+Aparapi provides a slightly improved interface over traditional GPGPU programming, the developers were not able to completely abstract the GPGPU component due to limitations in Aparapi when Hadoop+Aparapi was created. Thus, simplification of the Hadoop+Aparapi API was not possible. However, recently, 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 [3].

3.6 Spark-Ucores

In addition to GPU-accelerating Hadoop's MapReduce implementation, there have been attempts to provide GPU-acceleration for Apache Spark. One such attempt is called Spark-Ucores (previously called SparkCL). Spark-Ucores is a modification of spark that provides hardware accelerated processing using unconventional compute cores such as FPGAs, GPUs, and APUs [16].

Similarly to Hadoop+Aparapi, Spark-Ucores uses Aparapi to translate Java code into OpenCL. However, in order to provide support for devices like FPGAs and APUs, the Spark-Ucores team has also forked Aparapi in order to provide support for FPGAs and APUs, resulting in the creation of Aparapi-Ucores [16].

Spark-Ucores provides GPU-accelerated implementations of a few parallel operations on RDDs, including `map`, `mapPartitions`, and `reduce`. However, there are many other operations that Spark can perform on RDDs that can benefit from GPU-acceleration [16].

Spark-Ucores provides a similar three-step process to Hadoop+Aparapi's `preprocess`, `gpu`, and `postprocess` steps. However, Spark-Ucores provides a slightly weaker abstraction for the operations performed by the GPU. Spark-Ucores requires the user to be familiar with GPU programming in order to override the run method from Aparapi's GPU Kernel class, which requires the user to manually divide the work across the GPU threads [16]. Because Spark-Ucores does not have an not an explicit goal to provide an abstraction for its GPGPU components, programmers must have at least some GPU experience. In addition, in order to use Spark-Ucores, the user must restructure their existing Spark code, which may further discourage usage of

Spark-Ucores.

Chapter 4

IMPLEMENTATION

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)` to apply f to each item of L to produce L'.

In order to simplify the implementation of GPUMap, the the following assumptions have been made, which will be justified later in this chapter and in Chapter 7:

- 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.

When the programmer calls `gpumap(f, L)`, the following steps, described in greater detail later in this chapter, 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 $L_{1 \ldots n}$ 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 $L_{1 \ldots n}$.
8. The serialized input list, $L_{1 \ldots n}$, and any closure objects or lists are deserialized and the data is re-incorporated into the original objects.
9. The output list $L'_{1 \ldots n}$ is deserialized and is used to populate a list of objects based on the structure of L'_0.
10. L'_0 is prepended to $L'_{1 \ldots n}$ to form L' as desired and L' is returned.

4.1 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 both inspecting the fields of objects and tracing function execution during runtime.

4.1.1 Function Call Inspection

In order to create translations for the functions and methods that are to be called, some information about these functions and methods must first be collected and stored in a Function Representation or Method Representation object during runtime.

A Function Representation object contains the following information:

- The name of the function.
- A list containing the names of each parameter.
- A list containing the types of each argument.
- The return type of the function.
- A reference to the function object itself.

A Method Representation extends a Function Representation and adds one additional field which is the Python type of the class that the method should be associated with.

Due to the fact that Python is dynamically typed, there is no type information associated with function or method arguments. As a result, the functions or methods must be called and runtime inspection must be performed to determine the types that are passed as arguments. Similarly, anticipating the return types of Python functions or methods without actually calling them is difficult. Thus, the return type of each function or method is also inspected at runtime.

Luckily, because this is a map operation consisting of a function f and a list L, the functions and methods that are called, as well as their argument types and return types, can be determined by applying f to the first element of L while using Python's built-in function tracing functionality. Python's function tracing allows inspection of the current stack frame during a function call or function return. The stack frame at the beginning of a function call contains a list of parameter names as well as the objects that were used as arguments, allowing us to create a mapping between parameter name and argument type which is stored in the Function Representation. During a function return, the type of the returned object can be examined and stored in the Function Representation. If the function returns $None$, then the translated function will have a void return type. If any of these argument or return types are found to be Python floats, they are replaced by a double class that extends float to simplify code translation later, as Python's floats are actually doubles [14].

Functions are required to have a single return type every time the function is called because the Function Representation used to represent the function only stores a single return type. If the function returns other types of values, the Function Representation will not be accurate and the function will not be translated properly. Furthermore, due to the fact that C++ has fixed return types from functions and methods, translated Python functions must have a consistent return type.

If the function call turns out to be a method call, the object that the method was called on can be determined by examining the call's first argument. The type of this object is extracted and is then used to construct a Method Representation.

The Function Representation also contains an object that is the function itself because the function object is used to acquire the source code of the function which is used for code translation. During function tracing, if the call turns out to be a method being called on an object obj, the function object can be acquired by accessing the field of *obj* that has the same name as the method being called. Otherwise, the function object can be obtained by name from the global variables that can be found in the stack frame.

There is a special function, called the top-level function, which is the function that is passed as an argument to `gpumap` so that it can be applied to each element in a list. If the top-level function turns out to be a closure, the closure variables are added as arguments and their types are added as argument types in the top-level function's corresponding Function Representation. In the case where a closure variable is a list, the type that is inserted is a tuple consisting of `list` and the type of the items in the list. The closure variables can be found by inspecting the function object's `__closure__` field. If this field is empty, then the function is not a closure and modifying the top-level function's Function Representation is not necessary.

4.1.2 Object Inspection

There is a variety of information that needs 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 object and contains the following:

- The class's name
- The raw Python type of this class
- A list of the names of each field in the class
- A list of Class Representations for each field's types
- The format specifier string used to serialize members of this class using Python's `struct` module
- Method Representations for each necessary method of the class

Class Representations for the classes that will be used in the translated code must be extracted. These classes include the classes of objects in the input list, in the output list, in the closure bindings, and any objects constructed during the runtime inspection, including any nested objects.

Because a single Class Representation is eventually used to define a CUDA class and used to serialize all objects of the class, objects must be homogeneous, otherwise some objects will not comply with the Class Representation and cannot be properly serialized. Furthermore, objects must be restricted from containing objects of the same type both directly and indirectly. Otherwise, constructing both a Class Representation itself and the format specifier string will result in an infinite recursion.

There are two types of Class Representations, primitive Class Representations and complex Class Representations. Primitive Class Representations are used to represent objects that have corresponding CUDA primitives, such as ints, floats, and booleans. Primitive Class Representations do not have any fields or methods, and their format specifier string consists of a single character. Complex Class Representations represent normal objects that have fields and methods.

The complex Class Representation is a recursive data structure that may contain multiple other Class Representations for each field. As a result, the method of extracting a complex Class Representation is recursive. A complex 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 4.2 depicts a sample extraction of a Class Representation of an object of a class called `classA`. In order to speed up the recursive extraction process, Class Representations are cached and only

```
Extract ($obj, extracted$)
    Inputs : $obj$, An object
             $extracted$, A dict of previously extracted Class Representations
    Output: A Class Representation representing the class of $obj$
    $t \leftarrow$ type($obj$);
    if $t \in extracted$ then
        return $extracted$[t]
    else if isPrimitive(t) then
        return PrimitiveClassRepresentation($t$.__name__, $t$))
    else
        $name \leftarrow t$.__name__;
        $fields \leftarrow$ [ ];
        foreach ($field, value$) $\in$ obj.__dict__ do
            $fields$.append(($field$, Extract($value$, $extracted$)));
        $repr \leftarrow$ ClassRepresentation($name$, $fields$);
        $extracted[t] \leftarrow repr$;
        return $repr$
```

Figure 4.1: Recursively Extracting a Class Representation

need to be extracted once per class. If a Class Representation has previously been extracted, the Class Representation is immediately returned instead of attempting to re-extract the Class Representation.

Once the Class Representation has been extracted, its Python `struct` format string must be computed. For a primitive Class Representation, the format string consists of a single format character that corresponds to the CUDA primitive represented by the Class Representation, as dictated by the Python `struct` module. For complex Class Representations, the format specifier string is computed recursively. For each field name and Class Representation pair in the parent Class Representation,

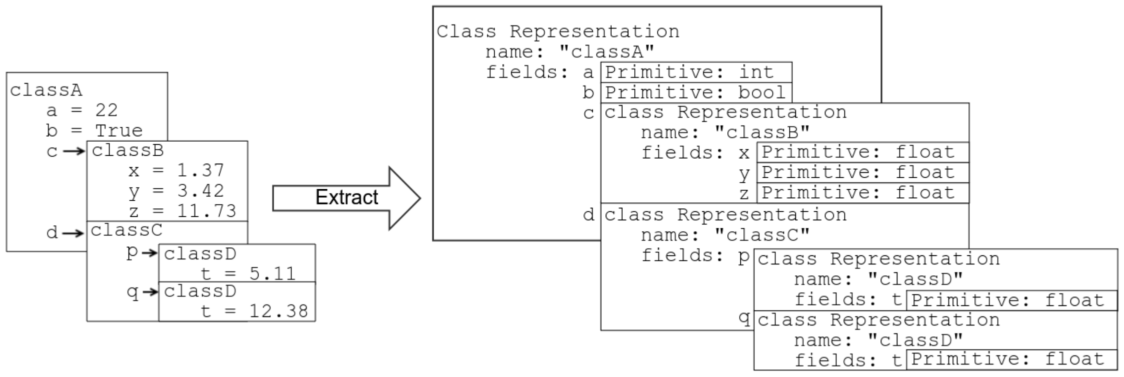

Figure 4.2: Sample Extraction of a Class Representation

the nested Class Representation's format string is inserted into the resulting format string.

In order to create a proper CUDA class definition for the class represented by the Class Representation, the class's method prototypes as well as method implementations must be included. This means that the Class Representation should also keep track of the Method Representations of the necessary methods. These Method Representations are inserted into the corresponding Class Representations after the function call inspection has been performed and all the Method Representations have been created. Methods that are not called during the runtime inspection phase are not included, as they did not have Method Representations created.

4.2 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 section will first discuss how to use references to emulate the scope of Python variables, and then it will explain how classes and functions are generated.

```
GetFormat ($repr, primitives$)
    Inputs : $repr$, A Class Representation
             $primitives$, A dict mapping primitive types to format specifiers
    Output: A String describing the entire struct format specifier of $repr$
    $format \leftarrow$ "";
    if $repr.format$ is not None then
        return $repr$.format
    else if $repr$ is a PrimitiveClassRepresentation then
        $format \leftarrow primitives[repr.\text{type}]$;
    else
        foreach $(field, nested\_repr) \in repr.fields$ do
            $format \leftarrow format +$ GetFormat($nested\_repr$, $primitives$);
    $repr$.format $\leftarrow format$;
    return $format$
```

Figure 4.3: Recursively computing a Class Representation's format specifier

4.2.1 Emulating Python References

In order to emulate Python's pass-by-reference behavior, all objects must be passed as references to functions, even when they are rvalues. In addition, objects must be returned by reference as often as possible. However, due to the fact that dynamic allocation is not being used, objects may need to be be returned as copies rather than as references to allow objects to move to a wider scope. The following paragraphs will explain useful information about C++11's rvalue references and then discuss how they are applied in passing arguments and return values between functions.

Rvalue references Typically, rvalues are stored by the compiler when they are computed and are immediately discarded. An rvalue reference is a special type of reference that is used to extend the lifespan of an rvalue by preventing the compiler from discarding the rvalue so that the rvalue can be used as an lvalue, meaning that the rvalue can be used as a local variable or as a function argument. Once an rvalue is bound to a named rvalue reference, either by passing the rvalue as a function argument or by assigning it, the rvalue reference behaves like an lvalue reference and is passed to functions as an lvalue reference. This allows differentiation between when an rvalue or an lvalue is passed as a function argument. In the case an rvalue is passed as an argument, the argument type is an rvalue reference. Otherwise, the argument type is an lvalue reference. In addition, rvalue references can also have existing objects or existing lvalue references assigned to them, making them very flexible. When assigning an rvalue reference, is is not necessary for any space to be allocated. If an existing object or reference to an object is assigned into the rvalue reference, the rvalue reference now points at the object. If an rvalue is assigned to an rvalue reference, the rvalue already exists in memory as the compiler allocated space to compute it. However, the rvalue is not immediately discarded by the compiler. Instead, the rvalue continues to reside in memory, and the rvalue reference refers to the object. An rvalue reference can also be used in place of a local object, meaning that the rvalue reference can be assigned into a local object, passed as a normal object argument, or returned from a function that returns a normal object. When an rvalue reference is assigned into a normal object, the object referred to by the rvalue reference is copied into the slot reserved for the normal object using a move constructor.

Passing Arguments In order to mimic Python's pass-by-reference behavior, all objects must be passed as arguments by reference, including both local variables and rvalues. This means that functions must be allowed to modify both lvalues and rvalues when they are passed into a function. However, C++ does not allow passing an rvalue as an lvalue reference while still allowing the rvalue to be modifiable. One way around this is to overload each function to take either an rvalue reference or lvalue reference for each non-primitive parameter. Unfortunately, this results in 2^n overloads for a function that has n non-primitive parameters. There is likely some work to be done to figure out which of these prototypes are actually necessary so only they are considered when generating code.

Returning Objects Due to the fact that GPUMap does not using dynamic memory management, emulating Python's behavior of returning objects from functions is slightly more difficult. If a return object was originally passed in as an lvalue reference, meaning that the return object exists outside the scope of a function, the object must be returned as an lvalue reference. In the case of methods, if a method is called on an object O and the method's return value is a field of O or even O itself, the return object must be returned as an lvalue reference due to the fact that O and all of its fields already exist outside the scope of the method. However, if the return object was passed in as an rvalue reference, created as a local variable, or is an rvalue, then the return object does not exist outside the scope of the function. As a result, the return object must be returned as a copy in order to allow the object to move to a wider scope. This means that for functions that eventually return a non-primitive argument, they must have different return types depending on the signature. Luckily, local variables holding objects are all declared as rvalue references, allowing either references or objects to be assigned into them.

4.2.2 Generating Classes

The information used to produce all necessary CUDA C++ class definitions is conveniently located in the extracted Class Representations. As described previously, a Class Representation contains a class's name, a list of fields and their types, as well as a list of Method Representations describing the prototypes that must be included in the class definition. The Method Representations contain a method's name, return type, and parameter names and types.

The first part of generating class definitions consists of creating a list of forward declarations for each class. This allows us to sidestep the issue of needing to define classes in any particular order and allows us to generate a wider variety of Python code. Without these forward declarations, some Python code cannot be translated. For example, suppose that class A has a field of type B and class B has a method that has a parameter of type A. This would cause a circular dependency and would require A to be defined before B and B to be defined before A. However, using forward declarations, the compiler is aware of both A and B before either of them are actually defined. Creating these forward declarations is a straightforward process and involves iterating through the name fields of all the extracted Class Representations.

Once the forward declarations are present, the next step is to generate the actual class definitions for each Class Representation that was extracted. In order to mimic the fact that in Python all fields and methods are public, all the fields and method prototypes in the generated CUDA C++ class definition are placed under the public modifier.

The first component of the class definition is the class's field declarations. The field declarations are obtained by iterating through the name and Class Representation pairs describing the fields and their types in the Class Representation for which the class definition is being generated. For non-primitive fields, instead of storing a pointer to another object as would be preferable in normal C++, the entire object is stored in the parent object, requiring objects to be stored in contiguous memory. This helps speed up serialization and deserialization by keeping objects in contiguous memory, but requires that objects are prevented from directly or indirectly containing objects of the same type, as previously mentioned. Storing objects in contiguous memory also allows simpler allocation of local variables, as performing parallel dynamic allocations from many GPU threads simultaneously is not ideal [5]. In addition, having contiguous objects requires that all objects must be homogeneous and must have all fields present. This means that objects of a particular type type cannot have null fields if other objects of the same type have data present in these fields.

The second component of the class definition is either a declaration or implementation of a default constructor. If a Python class has a constructor that does not take any parameters, then a prototype is simply declared and an implementation is provided later, during the method translation process. Otherwise, the default constructor is given an empty implementation and is automatically used by the CUDA compiler to initialize an empty object so that memory can be copied into the newly created object from another object that is passed by value either as a function argument or as a function return value. This empty default constructor is also automatically used by the compiler when an object is assigned into an existing object, such as when an object is written to the output list of the map operation. A copy constructor that takes a reference and initializes an object based on that reference is also necessary. The copy constructor is implicitly defined by the compiler and assigns each field from a reference into the newly created object. This copy constructor is used when a function or method A's return statement consists of a call to another function or method B that returns a reference. If A returns an object copy while B returns a reference, the object referred to by the reference returned from B must be copied, using the copy constructor, into the return value location in A's stack frame so that the object can be returned by A as an object copy. In addition to a copy constructor, a move constructor is also necessary. Luckily, the compiler also implicitly defines a move constructor and no additional work is necessary. The implicitly defined move constructor assigns each field of the object referred to by the rvalue reference to their corresponding fields of the newly created object. The move constructor is used when functions attempt to return an rvalue reference as an object copy.

The third component of the class definition is the method prototype declarations. These method prototype declarations are determined by iterating over each Method Representation in the Class Representation for which the class is being generated. Each Method Representation contains the name, return type, and parameter types of the method represented by the Method Representation. In order to attempt to mimic Python's pass-by-reference behavior for non-primitive objects, including both rvalues and lvalues, there must be multiple prototypes for each method. For each non-primitive parameter of a method, there must be a prototype that declares the parameter as a normal reference, as well as a prototype that declares the parameter is an rvalue reference, because methods must be prepared to take either rvalues or lvalues as arguments, as described previously in Section 4.2.1. In addition, depending on the scope of the return value of the function, the return value may either be returned as an lvalue reference or an object copy. This step must be done after function generation, as the abstract syntax trees must be inspected in order to determine whether an argument was returned, so that some of the prototypes can be given lvalue reference return types.

4.2.3 Generating Functions

In order to allow the generated functions to arbitrarily reference each other, prototypes for each function must be created by iterating through each of the existing

Function Representations and obtaining the function's name, parameter names, parameter types, and return type.

Generating the function prototypes is very similar to generating the method prototypes found in class declarations and requires generating 2^n prototypes for a function with n non-primitive parameters, as discussed in Section 4.2.1, so that each function can take both rvalues and lvalues as arguments. Every possibility of a non-primitive argument being an lvalue or rvalue reference is generated easily using Python's `itertools.product` function. Furthermore, whether the return type is a reference depends on whether an argument is returned, also discussed in Section 4.2.1.

After the prototypes are generated, generating the function definitions can be performed. For each different prototype for the same function, a corresponding function definition is created. Luckily, the definition of each overload of the function is the same, with the exception of the return type and parameter list. The reason is because in C++, rvalue and lvalue references can have their fields accessed and methods called using the exact same syntax.

Generating function definitions also requires iterating through the existing Function Representations to create abstract syntax trees (ASTs) of the Python functions that are to be translated. The function object in the Function Representation can be used to obtain the source code of the function, provided that the source file is available. Python's `inspect` module has a function called `getsource`, which takes the function object as an argument and returns the source code of the function represented by the function object. Then, Python's `ast` module can be used to parse an AST of from the function source so that the function can be systematically translated by recursively traversing the AST.

In order to traverse a Python AST, an implementation of `ast.NodeVisitor` is necessary. Implementing an `ast.NodeVisitor` involves implementing a variety of node handlers, which are methods of the form `visit_<NodeType>` where `<NodeType>` is replaced by the name of the type of node to be handled by that particular node handler. The node handler is passed one argument, which is the node that the node handler must handle. These node handlers often have recursive calls to visit which is a generic method provided by the `ast.NodeVisitor` that is used to call the appropriate node handler for nodes inside the current node. The `ast.NodeVisitor` implementation returns translated C++ code snippets from the each of its `visit_<NodeType>` methods, which are then forwarded by the `visit` method to any callers. This allows CUDA C++ code to be constructed by recursively navigating an AST. All of the C++ code for the function is generated by calling the `visit` method on the root of the AST parsed from the function source code.

The first useful AST node that is encountered during traversal is a FunctionDef node. The FunctionDef node contains the name of the function, the names of the parameters, and the body of the function as a list of AST nodes. As the function name and parameter names are already contained in the function's Function Representation, only the list of AST nodes representing the body of the function must be examined. Each item in the body list can be a literal, a variable, a statement, an expression, or a control structure. Due to the fact that Python has a similar syntax to C++, many language constructs, including object field accesses, if statements, while loops, various operators, return statements, and function/method calls map easily to their C++ counterparts. Unfortunately, there are two commonly used Python syntax elements that do not map so easily to C++ syntax elements: variable assignments and for-loops. In addition, because Python's built-in functions are not able to be translated, they must be either be mapped to suitable counterparts in CUDA C++, or they must be hand implemented in C++. The approaches used to accommodate variable assignments, for-loops, and built-in functions are discussed in the remainder of this section.

Variable Assignments The primary difference between variable assignments in Python and in C++ is that in Python, declaring variables is not necessary as they can be directly assigned without prior declaration. Furthermore, when Python variables are assigned for the first time, they do not have a type declaration, which is necessary in C++. Rather than attempting to perform type inference on the expression that is assigned into a variable, when a variable is used for the first time, its type is simply declared as `auto&&`, for the sake of convenience. By declaring the variable as `auto`, the CUDA compiler is forced to evaluate the resulting type of the expression that is to be assigned. Furthermore, by declaring the variable as an rvalue reference, with `&&`, either an object copy or a reference can be assigned to the variable. Declaring variables as rvalue references simplifies assigning a function's return value into the variable because a function can either return object copies or object references depending on the function and the way the function is called, as discussed in Section 4.2.1. No additional work is necessary to handle these cases separately as the work is automatically done by the CUDA compiler.

However, primitives must be handled slightly differently than objects. In the case that a primitive is assigned for the first time, the variable must be declared as a normal `auto` variable and not an rvalue reference, in order to properly mimic Python's behavior regarding primitive assignment. If an existing primitive variable is assigned into the rvalue reference, reassigning the rvalue reference will modify the existing primitive variable, which is undesirable. Thus, when assigning primitives, they must be assigned by value rather than by reference, meaning that a non-reference type must be declared.

Choosing to create a normal variable or an rvalue reference variable is determined by whether an assignment is assigning a primitive or an object. The following rules can help determine whether the assigned value is a primitive or an object:

- If the item being assigned is an argument to the current function, the item's type can be found in the current function's Function Representation
- If the item being assigned is the return value of a function or method, the item's type can be found in the corresponding function's Function Representation
- If the item being assigned is an existing primitive, the item is already known to be a primitive
- If the item being assigned is an integer, float, or boolean literal, the item is a primitive
- If the item being assigned is the result of a boolean operation, binary operation, or comparison, then the item is a primitive
- If the item being assigned is an element of a list, the item's type can be found by checking the type of data items stored in the list
- If the item being assigned is the field of an object, the object's corresponding Class Representation can be checked to determine the field type

For Loops The only type of for-loop supported in Python is a for-each loop, which makes heavy use of Python's iterators. In contrast, CUDA C++ does not support for-each loops and does not support iterators. One solution to bridge the gap between Python iterators and normal loops in CUDA C++ is to implement iterators from scratch in C++. In order to properly mimic Python for-each loops, the iterators need to be able to perform two tasks:

- An iterator must be able to produce new elements.
- An iterator must be able to check whether there is a new element to be produced.

These tasks are performed by calling `next` or `has_next` on the iterator, respectively.

The iterator is used by first instantiating the iterator and then using it in conjunction with a while loop. The condition used in the while loop is simply a call to the method used to check whether there is a new element to be produced. In the body of the while loop, the first step is to create a local variable with the same name as the Python loop's target variable and assign the next element that is produced by the iterator to this variable. The remainder of the loop body consists of a translation of the body of the original Python loop.

GPUMap implements two types of iterators, allowing two types of Python for-each loops to be translated: a loop iterating over a range of numbers, and a loop iterating over a list. Their implementations will be discussed in the next few paragraphs.

Iterating Over A Range In order to iterate over a range of integers, an iterator that produces integers within a desired range must be implemented. If the next element is outside of the desired range, the call to `has_next` returns false and the loop terminates. If the next element is inside the desired range, `has_next` returns true and the loop begins the next iteration. At the beginning of the iteration, the call to `next` returns the next element. This next element is assigned to a local variable with the same name as the variable that is used in the Python for-loop.

During the first call of `next`, the returned value is the number at the beginning of the range. At each call to `next`, the returned value is stored as the last returned value. That way, during a subsequent call to `next`, the last returned value can be incremented by the step size and can be returned. If the sum of the previously returned value and the step size is outside the desired range, `has_next` returns false and `next` is no longer called because the loop terminates.

Iterating Over A List In order to iterate over a list, an iterator that produces references to each element in the list must be implemented. The iterator must produce references rather than copies because modifications to the list elements must be possible from translated functions.

Lists are stored in a class that has two fields: an integer describing the length of the list, and a pointer to an array containing the list elements.

Upon initialization of the iterator, the iterator is given a reference to the list object and the iterator's current index is initialized to zero. Upon each subsequent call to `next`, the iterator stores a reference to the element at the current index, increments the current index and returns the stored reference. If the current index is equal to the length of the list, `has_next` returns false.

Calling Built-in Functions Programmers should be able to call some commonly used 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.

The built-in Python functions that are supported during translation must be mapped to the names of the functions that must be called from the translated code. This mapping is stored in the form of a series of nested dicts as depicted in Figure 4.4. The reason built-in function names are stored in a nested dict structure is because functions from the same packages can be grouped together. By specifying the packages from which the functions originate, the likelihood that a function will be incorrectly

```
built_in_functions = {
    "len": "len",
    "print": "print",
    "math": {
        "exp": "exp",
        "sin": "sin",
        "cos": "cos",
        "tan": "tan",
        "ceil": "ceil",
        "floor": "floor",
        "sqrt": "sqrt",
        "pow": "pow",
        "log": "log",
        "log10": "log10",
        "log1p": "log1p",
        "log2": "log2",
    }
}
```

Figure 4.4: A mapping of supported built-in functions

mapped to a built-in function is decreased. The final output after providing a series of keys is the name of the C++ function that should be called.

When a Call AST node is found, the appropriate visit method, `visit_Call`, is called on the implementation of `ast.NodeVisitor`. The Call node that is passed to the `visit_Call` method contains a AST node describing the name of the function or method that is called, and a list of AST nodes for each argument that is passed to the function. The name node is evaluated and a string is created to determine the corresponding C++ built-in function if such a function exists, by using the algorithm in Figure 4.5 to traverse the dict shown in Figure 4.4. This algorithm splits the string describing the name of the function or method by the period character and checks to see whether each piece of the split string is in the built-in function dict, using each piece as a key to traverse the nested dicts. If the end result is a string, then a built-in function was matched and the resulting string is used in place of the original function name in the translated code. If None was returned, this indicates that no translation is necessary and the existing function name is used without translation.

```
GetBuiltinName (name, builtin)
    Inputs : name, A string containing the name of a function or method
             builtin, A nested dict containing built-in function names
    Output: The name of the built-in function that should be called in the
            translated code or None if no such function exists
    split_name ← name.split(".");
    temp ← builtin;
    foreach piece ∈ split_name do
        if piece ∉ temp then
            return None
        else
            temp ← temp[piece];
    if temp is a string then
        return temp
    else
        return None
```

Figure 4.5: An algorithm used to determine built-in function correspondence

4.2.4 Generating Methods

Method translation is almost exactly the same as function translation. Method Representation objects contain a reference to the Python class object associated with the method, in addition to the information contained by a Function Representation. The class object is used to provide the class's name so that the class name can be included in the first line of the method declaration. All of the syntax is translated the same way, except that every time a reference to `self` is made in the Python code, the reference is translated to `(*this)`. In addition, there is one additional check to make sure methods do not attempt to add fields to their objects by comparing the field that is assigned to the list of available fields in the Class Representation representing the class of the object. If a field is added by a non-constructor method, then an exception is thrown indicating that this is not allowed.

4.2.5 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. To determine which list item each thread should operate on, the thread ID given to each thread is used to index the input list.

The CUDA kernel function is a function that takes a pointer to the serialized input list. If the top-level function does have a return value, i.e. does not return None, then a pointer to the space allocated for the output list is also passed to the kernel function. If the top-level function is a closure, then the kernel's arguments also contain pointers to the serialized closure variables of the top-level function. Serialization of closure variables is discussed in more detail in Section 4.3.

Class Representations used to describe the items in the input list, the items in the output list, and the closure variables must be used to fill in the parameter types and template arguments in the kernel function's parameter list. For each of the parameters of the kernel function, if the parameter is a list pointer type, then its template argument must be filled in with the proper type so that the compiler knows what type of objects or primitives are included in the list. If the parameter is not a list pointer, then the proper type name must be inserted as the parameter type based on the name field of its Class Representation.

The first step of the kernel function is to compute the thread ID. Because the input list is a list and lists are one-dimensional, only one-dimensional blocks arranged in a one-dimensional CUDA grid are necessary. Thus, the thread ID can be computed as follows:

```
int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
```

If the thread ID is less than the length of the list, the thread will apply the top-level function to the item in the input list, passing all of the closure variables to the top-level function as well. Each of these items are passed by reference so that they can be modified by the top-level function or any functions called by the top-level function. If the top-level function has a return type, then the output of the call to the top-level function is copied to the location in the output list described by the thread ID.

4.3 Serialization

When calling `gpumap(f, L)` with a function f and a list L, prior to actually running the translated code, 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. This section will discuss how data is converted from Python objects to a serialized format, how the serialized data is copied to and from the GPU, and how the serialized data is deserialized back into Python objects.

4.3.1 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 4.6.

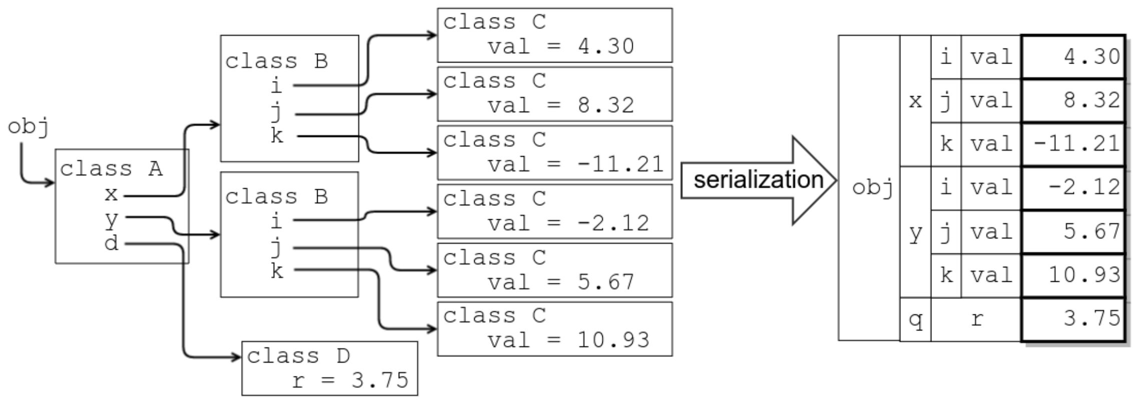

Figure 4.6: A normal Python object and its serialized counterpart

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. However, 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. Otherwise, the data will not be in the proper order to match up with the class definition and will not match up with the precomputed format specifier in the Class Representation. Luckily, because order of the fields in the class definition is determined by the order of the fields in the class's corresponding Class Representation, the Class Representation and its nested Class Representations can be used to traverse the object and collect its data in the correct order. The algorithm in Figure 4.7 is used to recursively collect all the data of an object and any nested objects.

The algorithm must be provided with an object to extract data from and a Class Representation representing that object. The algorithm iterates through all the field names and field types in the Class Representation. If the field type is a primitive Class Representation, then the data in the field is of a primitive type and must be collected. Otherwise, if the field type is a complex Class Representation, this

```
ExtractData ($obj, class\_repr, data\_items$)
    Inputs: $obj$, An object
            $class\_repr$, A Class Representation representing the object
            $data\_items$, A list to contain the extracted data items
    foreach $field, repr \in$ zip(class_repr.field_names, class_repr.field_types) do
        if repr is a PrimitiveClassRepresentation then
            $data\_items$.append($obj$.__dict__[$field$]);
        else
            ExtractData($obj$.__dict__[$field$], $repr$, $data\_items$);
```

Figure 4.7: An algorithm to recursively collect all the data in a Python object

indicates that there is an object to be recursively examined in the field. In this case, ExtractData is recursively called and a reference to the list of data items that is currently being built is passed along.

Using this algorithm, the data is guaranteed to be in the same order as expected by the C++ class definition to be used with the serialized object. Then, the data can be converted into its serialized form by using Python's `struct` module. The `struct` module has a method `pack`, which is used to pack binary data. In order to use `struct.pack` to pack a list of data items, a format specifier string must be provided. The format specifier string for the object can be found in the Class Representation, as the string was precomputed during the construction of the Class Representation. Calling `struct.pack` with the format specifier string and the list of data items produces a byte array object containing the binary representation of the list of data items. This byte array that can be passed to PyCUDA's `to_device` function which allocates appropriate memory on the GPU for the byte array, copies the data into the allocated memory, and returns a `DeviceAllocation` object. This `DeviceAllocation` object is effectively a pointer to GPU memory and can be passed into the CUDA kernel as an argument.

Once the serialized data has been copied to the GPU and the kernel has been executed, the serialized objects must be copied back to the host and then deserialized. In order to copy the serialized objects back to the host, a byte array with the same size as the serialized object must first be allocated. The size of the serialized object can be determined by using `struct.calcsize`, which takes a format specifier string and outputs the number of bytes needed to store a struct with the specified format. To copy the serialized data from the GPU into the empty byte array, PyCUDA's `memcpy_dtoh` function is used. The `memcpy_dtoh` function takes a byte array to store the copied data and a DeviceAllocation that describes the location of the data on the GPU. This `memcpy_dtoh` function is passed the empty byte array and the existing `DeviceAllocation` object that was created when copying the serialized data to the GPU.

After the serialized data is copied back to the host, the data is still contained in a byte array and must be unpacked. The first step of deserialization is using `struct.unpack` to convert the byte array into a list of data items using the format string from the Class Representation. Now, the data items must placed back into the fields of the object from which they originated, using the algorithm in Figure 4.8.

This algorithm requires an object to unpack the data items into, *obj*, a Class Representation for the class of *obj* to guarantee correct order of iteration over the fields of *obj*, the list of data items that must be unpacked into *obj*, and the number of data items unpacked from the data items list (initially zero). The algorithm iterates over all the fields in the Class Representation, recursively stepping into the field of an object if a non-primitive Class Representation is found. If a primitive Class Representation is found while iterating over the Class Representation's fields, the

```
InsertData ($obj, class\_repr, data\_items, num\_unpacked$)
    Inputs : $obj$, An object
             $class\_repr$, A Class Representation representing the object
             $data\_items$, A list containing the data items to unpack into $obj$
             $num\_unpacked$, The number of items unpacked from $data\_items$
    Output: The number of data items unpacked so far
    foreach $field, repr \in$ zip(class_repr.field_names, class_repr.field_types) do
        if repr is a PrimitiveClassRepresentation then
            $obj$.__dict__[$field$] $\leftarrow$ $data\_items[num\_unpacked]$;
            $num\_unpacked \leftarrow num\_unpacked + 1$;
        else
            $num\_unpacked \leftarrow$ InsertData($obj$.__dict__[$field$], $repr$, $data\_items$,
             $num\_unpacked$);
    return $num\_unpacked$
```

Figure 4.8: An algorithm to recursively insert a data list into a Python object

next item from the data item list must be inserted into the field described by the primitive Class Representation.

4.3.2 Serializing Lists of Objects

Now that serialization and deserialization of a single item is possible, serializing and deserializing entire lists of items becomes straightforward. The process for serializing an entire list should be 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. In addition, the process for deserializing an entire list should be similar to deserializing an a single object. 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.

In the translated code, lists have a length and a flexible array member that contains all of the serialized items. In addition, because there can be a list of any type of items, the list class must be a template class. Lists are defined as shown in Figure 4.9.

```
template <class T>
class List {
    public:
        int length;
        T items[];
}
```

Figure 4.9: List struct used to access serialized data

Due to the fact that the list items are contained in a flexible array member, all of the items in a list can be serialized by serializing each item individually and concatenating the results. The list definition must have a length member, as this is used to iterate over the list using the technique described in Section 4.2.3. In addition, the list length is used to determine how many threads must execute the top-level function on items in the input list, as described in Section 4.2.5.

```
SerializeList ($list, class\_repr$)
    Inputs : $list$, A list of objects
             $class\_repr$, A Class Representation for the objects in $list$
    Output: A byte array containing the serialized version of all the objects
    $data\_lists \leftarrow$ map(ExtractData, $list$);
    $serialized\_items \leftarrow$ map(lambda $l$: struct.pack($class\_repr$.format, $l$), $data\_lists$);
    $byte\_arrays =$ [struct.pack("i", len($list$))] + $serialized\_items$;
    return bytes.join($byte\_arrays$)
```

Figure 4.10: An algorithm to create a byte array from a list of python objects

A byte array that matches the list struct definition is created by using the algorithm shown in Figure 4.10. The byte array contains the serialized version of the list length, as well as the serialized versions of the list items. First, the data is extracted from each object in the list using the method used to extract data from a single object from Figure 4.7. Then, the data lists containing the data of each object are converted into byte arrays using `struct.pack` with the appropriate format string from the Class Representation that represents the objects. These byte arrays are collected in a list. Then, a byte array that contains the binary representation of the list length is created and added to the front of the list that contains the byte arrays for each object. The last step is to concatenate all the byte arrays, including the list length and each serialized object, to produce a byte array that can be copied to the GPU.

Similarly to allocating a single object on the GPU, as described in Section 4.3.1, PyCUDA's `to_device` is used to allocate the byte array. The call to `to_device` returns a DeviceAllocation object that can later be used to copy the data back to the host after the data has been operated on by the translated code.

Once the translated code has finished running, the serialized lists must be copied back to the host and the data items must be unpacked back into their corresponding objects. In the case that the list is an output list of the map operation, corresponding objects not yet exist, so the objects must first be created.

In order to copy the data back to the host, a byte array with the proper size must first be prepared. The proper size of the byte array can be computed using `struct.calcsize` on the format string in the Class Representation that describes each object in the list. This provides the size of a single object, so in order to obtain the size of the entire list, the result must be multiplied by the total number of objects in the list, as they are all guaranteed to be the same size. Then, the size of the serialized list length must be added, which can be computed using `struct.calcsize` on the string “i”. Once a byte array with this size has been created, the serialized list can be copied from the GPU into the allocated byte array by calling PyCUDA’s `memcpy_dtoh` and passing the byte array as well as the DeviceAllocation object that was returned during allocation of the serialized list.

Once the serialized list has been copied back to the host, the list must be deserialized. Deserializing a list is a slightly more complex process than serializing the list because the boundaries of different objects are not immediately clear. However, the information in the Class Representation can be used to extract all of the data items from the byte array and can then be used to insert the data items into the correct fields of their corresponding objects. The first step is to create a format string to use with `struct.unpack` that can create a list containing all the data items in the byte array. To create the format string, Class Representation’s format string can be concatenated repeatedly for each element in the list. Then “i” must be added to the beginning of the resulting string to account for the list length at the beginning of the list struct. Then, `struct.unpack` can be called on this string and the copied byte array to create a list of data items that contains the data of all of the objects in the list. Once the list of all the data items is created, the algorithm in Figure 4.11, which makes use of the algorithm in Figure 4.8, is used to insert the data items into their corresponding objects.

The algorithm in Figure 4.11 first initializes *unpacked_count* to 1, in order to skip the list length that was deserialized as this value should not be unpacked into any objects in the list. Then, the algorithm iterates over each object in the list and unpacks the data items into the object. The *unpacked_count* must be specified so that `InsertData` knows where to find the correct data items for the current object. `InsertData` inserts the data into each object according to order of the fields specified in the Class Representation, as discussed previously. Then, the index in the data item list is stored so that it can be used by the subsequent `InsertData` call in the

```
InsertListData ($list, class\_repr, data\_items$)
    Inputs: $list$, A list of objects for $data\_items$ to be unpacked into
            $class\_repr$, A Class Representation for the objects in $list$
            $data\_items$, The data items to be unpacked into the objects of $list$
    $unpacked\_count \leftarrow 1$;
    foreach $obj \in list$ do
        $unpacked\_count \leftarrow$ InsertData($obj$, $class\_repr$, $data\_items$, $unpacked\_count$)
```

Figure 4.11: An algorithm to insert a deserialized list's data items into the originating list

next iteration.

In the case that the list is an output list, there are no existing objects to be unpacked into. When calling `gpumap(f, L)` with a function f and a list L, in order to perform runtime inspection, f is applied to the first item in L, L_0 to produce L'_0. This gives us a sample object that is representative of the objects in the output list L', as well as a Class Representation of the objects in the output list. This object, L'_0, is used in conjunction with Python's object deep-copy functionality to create a list of objects to be populated with data from the deserialized output list. However, due to performance reasons `copy.deepcopy` is not used. Instead, Python's `pickle` module is used as it provides better performance. Python's `pickle` module allows Python objects to be serialized so they can be saved and re-instantiated at a later time.

The algorithm in Figure 4.12 describes how an output list is created from the deserialized data items using a sample object as a template. The output list length must be specified so that the algorithm can create the proper number of output objects using the sample object as a template. The algorithm first starts by initializing an empty output list, which will contain the output objects as they are created. Then, a pickled representation of the sample object is created using `pickle.dumps` that can later be repeatedly unpickled to instantiate output objects as they are needed. Similarly to the algorithm in Figure 4.11, *unpacked_count* is initialized to 1 in order to skip over the list length. The first item in the deserialized data items list is the length of the list due to the layout of the list struct, shown in Figure 4.9. This list length is used to determine the number of objects to be instantiated in the output list.

```
InsertListData (sample_obj, class_repr, data_items)
    Inputs : sample_obj, An object used as a template for creating the output list
             class_repr, A Class Representation for the sample_obj
             data_items, The data items to be unpacked into the output list
    Output: A list containing newly created objects from the data in data_items
    output_list ← [ ];
    pickled_sample_obj ← pickle.dumps(sample_obj);
    output_list_length ← data_items[0];
    unpacked_count ← 1;
    foreach i ∈ range(output_list_length) do
        obj ← pickle.loads(pickled_sample_obj);
        unpacked_count ← InsertData(obj, class_repr, data_items, unpacked_count);
        output_list.append(obj);
    return output_list
```

Figure 4.12: An algorithm to insert a deserialized output list's data items into a new list

For each object to be instantiated, a copy of the sample object is instantiated using `pickle.loads`. Then, the data in the object is populated using `InsertData`, described in the algorithm in Figure 4.8, storing the current position in the data item list. The object that was just populated by `InsertData` 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.

4.4 Integration in Spark

GPUMap can be integrated in a variety of transformations and actions that can be performed on Spark RDDs. The transformations that have been implemented in order to test the viability of incorporating GPUMap into Spark are map, filter, and foreach. 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 chapter regarding lists also extend to Spark RDDs.

4.4.1 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.

This 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. However, rather than producing a map generator to evaluate the application of f to each element from an iterator in sequence, the application of f on many items from the iterator must be evaluated at once, in parallel. 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. However, 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 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.

4.4.2 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.

Because GPUMap does preserve side-effects, GPUMap 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.

4.4.3 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.

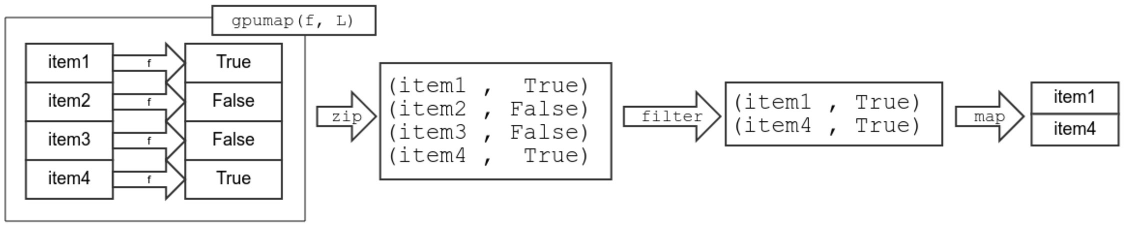

Figure 4.13: GPURDD filter method

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 4.13.

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`.

Chapter 5

VALIDATION

In order to determine the viability of GPUMap, some tests to see whether GPUMap works properly must first be performed. To help guarantee correct functionality, GPUMap needs to properly:

- Perform runtime inspection to extract the necessary data
- Translate Python code to CUDA C++ code
- Serialize input lists and closure bindings.
- Deserialize input lists, closure bindings, and output lists.

In addition, another requirement of GPUMap is that GPUMap works with native Python code. If a function f and a list L comply with GPUMap's restrictions, regardless of whether `gpumap(f, L)` or `map(f, L)` is called, the results should be the same. This means that GPUMap's functionality can be tested by creating a variety of test programs, running them with both `gpumap` and `map`, and comparing the results.

This section will discuss the tests used to validate GPUMap's functionality. These tests consist of a suite of language translation unit tests, a serialization test, two Python tests incorporating GPUMap, and four Spark tests incorporating GPURDD. In addition, the Python and Spark tests are also used as performance benchmarks, which are discussed in Section 6.

5.1 Language Translation Tests

Testing the language translation components of GPUMap can be done through a series of unit tests that attempt to translate each different type of Python AST nodes. These tests are used to test the `FunctionConverter` class, which extends `ast.NodeVisitor`, as well as the `MethodConverter` class, which extends `FunctionConverter`. In the case that the AST node represents an unsupported feature, FunctionConverter or MethodConverter's `visit` method should raise a SyntaxError with a small explanation of what went wrong.

Each test contains at least one snippet of Python code that is parsed using `ast.parse`. If possible variations of an AST are present, several code snippets are provided and parsed in an attempt to create different configurations of the same AST node. Then, the resulting AST is passed to either `FunctionConverter` or `MethodConverter`'s `visit` method. The return value from this `visit` method should be the same as the desired translated CUDA C++ code. If the AST contains unsupported features, the tests check that a SyntaxError is raised.

The tests can be found in Appendix A and are grouped into two parts. The first part, `TestFunctionConverter`, tests `FunctionConverter`'s ability to translate all different types of AST nodes. The second part, `TestMethodConverter`, only tests the functionality added or overridden by `MethodConverter` as a result of extending `FunctionConverter`.

5.2 Serialization Test

In order for the data to be properly operated on by the translated code, the data must be arranged in a contiguous chunk of memory. Once the data has been operated on, the data must be properly inserted back into the objects from which the data originated. Thus, in order to test GPUMap's ability to reliably serialize and deserialize data a serialization integration test is performed. The test is used for testing GPUMap's serialization capabilities involving serializing and deserializing both lists of primitives and lists of objects into their originating lists and also into new lists. The source code of the test can be found in Appendix B.

GPUMap must be able to deserialize data back into the data's originating objects because retaining side-effects on closure variables and input lists is necessary in order to properly mimic Python's map function. In addition, GPUMap must also be able to deserialize data into a list of new objects because this occurs when producing an output list as a result of transforming the input list.

The first test, which tests serialization of lists of objects, starts by generating a list containing objects of `TestClassA`, which consist of randomized integers and nested objects of the classes `TestClassB`, and `TestClassC`. The list is then serialized and deserialized back into itself. If the list is properly serialized and deserialized, the objects in the list must contain the same data as they were initialized with. Prior to deserialization, a deep copy is made to check the deserialized elements against. In addition, the serialized data is also deserialized into a list of new objects. This new list is compared against the deep copy in order to ensure that the data was properly deserialized.

The second test performs the same thing as the first test, but using randomly generated integers rather than object hierarchies in order to test GPUMap's serialization functionality with primitives. The list is serialized and then deserialized back into the same list, making sure the elements are unchanged. Then, the serialized data is deserialized into a new list, making sure the elements are the same as in the original list.

A final test is done to check whether the data can be operated on using CUDA code. The same objects from the first test are serialized and copied to the GPU. A small test kernel is run that attempts to write a value to a particular field in each object that is computed using arbitrary operations on the other fields in the object. After the kernel is run, the serialized data is copied back to the host and the data is deserialized into a list of new objects. The fields that are unchanged are checked to ensure they remain the same between the old list and the new list. The changed field is checked to make sure that it contains the correct value.

5.3 N-body Test

The first Python test that attempts to exercise the entire functionality of GPUMap is an all-pairs n-body implementation based on the outer loop approach from [17]. This is a comprehensive test that tests many of the features of GPUMap. The implementation of this test can be found in Appendix C.

The test begins by a creating a list of randomized bodies, with randomized positions and velocities, and makes two copies of this list: one for use with `gpumap` and one for use with `map`. Then, two simulations are instantiated: a `CPUSimulation` and a `GPUSimulation`, which both extend the `Simulation` class. Each of these simulations overrides the `Simulation.advance` method, which is where either `gpumap` or `map` should be called. In order to run the simulation, the `run` method of the simulation is called, which simply calls its `advance` method repeatedly for the desired number of time steps.

Both simulations make use of a `Vector3` class that contains methods to add and subtract other vectors, scale the vector, and compute the length of the vector. The simulations also make use of a `Body` class that contains the mass of the body, as well as two `Vector3`s, one for position and one for velocity.

This `advance` method is called to move the simulation forward one time step.

The `advance` method defines a function `calc_vel`, which takes an integer telling the function which body to calculate the velocity for. The list of bodies is included as a closure binding to `calc_vel` and is indexed using the integer that is passed in to determine the current body. In addition to the list of bodies, the time step width and a distance padding value (to avoid division by zero) are also included as closure bindings to `calc_vel`. Once the current body is determined, `calc_vel` iterates over all bodies in the body list in order to calculate their impact on the velocity of the current body. For each iteration, a distance vector between the position vectors of the bodies is calculated using the `Vector3.sub` method. Then, the length of the distance vector is calculated using `Vector3.distance`, which internally calls `math.sqrt` and `math.pow`. Both functions `math.sqrt` and `math.pow` are translated to their corresponding CUDA counterparts when running through `gpumap`. Then the velocity of the current body is modified as a result of calls to `Vector3.scale`. Then `map` or `gpumap` is called with `calc_vel` and a list of consecutive integers to update the velocity of each body.

Once the `calc_vel` function has been applied to each body, the updated velocity must be used to alter the position of the body, using the `update` function defined in `advance`. The `update` function takes a body and updates the body's position by adding a displacement vector that is calculated by scaling the current velocity vector by the time step. Then `map` or `gpumap` is called with the `update` function as well as the list of bodies.

All of the classes, functions, and lists used in the calls to `gpumap` and `map` are either shared or are exactly the same, in order to demonstrate that `gpumap` works with unaltered code. In addition, the n-body validation test exercises GPUMap's closure functionality for both single variables and lists by including the body list, the time step, and the padding in `calc_vel`'s closure bindings. The n-body test also tests looping over a list that is included as a closure binding. Side effects both on closure bindings (through `calc_vel`) and on the input list (through `update`) are also tested.

The test also tests passing both rvalues and lvalues to functions and methods, as they are called with both existing variables and the results of other calls. In addition, the test also tests mapping built-in Python functions such as `math.sqrt` to their corresponding CUDA counterparts.

The n-body test is run for between 2 and 8192 bodies by powers of two. Each of these tests is run for 10 time steps. After each run, the results of the `GPUSimluation` and `CPUSimluation` are compared. This involves comparing each component of the position and velocity vectors, as well as the masses of each corresponding pair of bodies produced by the `GPUSimluation` and `CPUSimluation`. Due to doubles being handled slightly differently by CUDA and by Python, the results are checked to be approximately the same.

5.4 Bubble Sort Test

The parallel bubble sort test is a simple test that sorts each list in the input list. This test exercises GPUMap's ability to operate on lists of lists of primitives, iterate over ranges of numbers, and properly assign primitives into each other. The implementation of the bubble sort test can be found in Appendix D.

The test involves creating a list L of lists of n random integers and calling the function `bubblesort` to sort each of the lists in L. The test is performed for values of n from 2 to 8192 by powers of two. Copies are made of L to be passed to both `map` and `gpumap`. In addition, a copy of L is also sorted using Python's `sorted` function to be compared against the lists outputted by both `map` and `gpumap` to assert that they are properly sorted.

5.5 Shell Sort Test

The parallel shell sort test attempts to sort each list in a list using shell sort instead of bubble sort. The shell sort test runs through Spark and incorporates the use of a GPURDD. The reason shell sort was chosen over faster algorithms like merge sort or quick sort is because these algorithms are recursive and CUDA does not support recursion well. Although an iterative variation of merge sort and quick sort can be written, they require allocation of a temporary list, which is currently not supported in GPUMap. The source code for the shell sort test can be found in Appendix E.

The test exercises GPUMap's ability to operate on lists of lists, translate while loops, iterate over ranges, use logic operators, use augmented assignments, use integer division, and use expressions as list indices. In addition, the shell sort test exercises GPUMap's ability to be incorporated into Spark through the use of the GPURDD's `foreach` function.

The test first creates an RDD consisting of a list of integers of length n. Then, these integers are mapped to lists of random numbers of length 10000 to produce an RDD L with containing n lists of 10000 elements each. A copy of L, L', is produced by mapping L with an identity function. Then, `foreach` is called on L, passing the `shellsort` function, which sorts the lists in L using the CPU on the workers. Afterwards, L' is converted into a GPURDD by passing L' into the constructor of GPURDD. Then the GPURDD's `foreach` method is called with `shellsort`, which sorts the lists in L' using the GPUs on the workers. Once both L and L' have had their lists sorted, lists are collected on the driver to make sure they are properly sorted.

5.6 Pi Estimation Test

The Pi Estimation test attempts to estimate the value of pi using a Monte Carlo simulation. This test is adapted from the existing Spark example for estimating Pi. The test consists of picking random coordinates with a uniform distribution in a 2x2 square which is centered at the origin. Each random coordinate is checked whether the coordinate is in the unit circle. Then, the proportion of points that land inside the unit circle can be computed to estimate the ratio of the area of the circle to the area of the surrounding square. Once this ratio is determined, the ratio can be multiplied by the area of the square to estimate the area of the circle. The value of pi is equal to the area of the unit circle because the unit circle has a radius of 1.

There are three different versions of the pi estimation test. The first version is a normal pi estimation test that uses GPURDD's `map` method to check whether each point is inside the unit circle. The second version of the pi estimation test uses GPURDD's `filter` method to check whether each point is inside the unit circle. The final version of the pi estimation test arranges the random chosen points into an RDD of lists and performs a combined map and reduce step on each list of random points. Each of these tests consists of performing the same operation and calling the same code from both the CPU and GPU and then comparing the runtime. Code that is used across all the tests can be found in Appendix F. These variations of the pi estimation test will be explained in greater detail in the following sections.

5.6.1 Using Map

The first version of the pi estimation test simply tests GPURDD's `map` method by using `map` to determine whether each randomly generated coordinate pair in an RDD is inside the unit circle. The source for the first pi estimation test can be found in

Appendix F.1.

The test uses a function called `inside_unit_circle` that takes a coordinate pair object and returns 1 if the coordinate pair is inside the unit circle and 0 otherwise. This means that after `map` is called, the resulting RDD contains 1s and 0s. This resulting RDD can reduced to determine the number of randomly generated coordinate points residing inside the unit circle, which can then be used to estimate pi.

The test starts by creating an RDD of n integers, where n is the number of desired coordinate points to use for estimation. Then, each integer is transformed into a random coordinate pair, to create an RDD R. This RDD of random coordinate pairs, R, is then cached so the RDD can be used multiple times, once for the CPU test and once for the GPU test, without being recomputed.

The CPU test simply consists of creating an RDD by calling R's `map` with the `inside_unit_circle` function. The resulting RDD's `count` method is then called in order to force evaluation and the runtime of the call to `count` is measured. The GPU test first creates a GPURDD from R and then calls `map` with the `inside_unit_circle` function to produce an RDD of 1s or 0s. The resulting RDD's `count` method is called in order to force evaluation and the runtime of this `count` call is measured.

The contents of the RDDs produced by the CPU test and the GPU test should be identical, and this is verified in order to test whether GPURDD's `map` method works properly.

5.6.2 Using Filter

The second version of the pi estimation test tests GPURDD's `filter` method by using `filter` to remove the points from the list that are not in the unit circle and then counting the size of the resulting RDD to determine the number of random points inside the unit circle. The source code for the filter version of the pi estimation test can be found in Appendix F.2.

The test uses a function called `inside_unit_circle`, but instead of returning a 1 or 0 like in the previous test, the function returns a boolean value indicating whether the point is inside the unit circle. This function can be used with `filter` because the function returns a boolean indicating whether elements in the RDD should be kept. The resulting RDD after removing points outside the unit circle can be counted in order to determine the number of points that are in the unit circle, allowing us to estimate pi.

Similarly to the map version of the pi estimation test, this test creates an RDD of n integers and transforms these integers into random coordinate pairs. Then, two RDDs are prepared: one which will call RDD's `filter` method with the `inside_unit_circle` function upon evaluation, and one which will call GPURDD's `filter` method with `inside_unit_circle` upon evaluation.

In order to force evaluation on these RDDs, the `count` method is called on each. The time taken for each of these `count` calls is measured.

Then, the RDDs are collected and compared to make sure that both RDDs retained the same points after being filtered.

5.6.3 Using Parallel Reduction

The final version of the pi estimation test attempts to combine the map and reduce steps as much as possible in order to do more work toward estimating pi on the GPU. The source for this final pi estimation test can be found in Appendix F.3.

Instead of creating an RDD of integers and then transforming the RDD into an RDD of random coordinate points, the RDD of integers is transformed into an RDD of lists of random coordinate points. When `map` is applied to this RDD containing lists of coordinate points, each list will be passed to a function called `count_and_reduce`. The `count_and_reduce` function takes a list of coordinate pairs and outputs a single integer value representing the number of points in the list that were inside the unit circle. This yields a RDD of integers, which can be further reduced on the CPU in order to compute the final proportion of coordinate points that landed inside the unit circle so that pi can be calculated. Each integer in this RDD represents the number of coordinate points in the corresponding list that were inside the unit circle.

Two RDDs are prepared from the RDD containing lists of random coordinate points. Upon evaluation, one RDD will be evaluated using the normal `map` method, and the other will be evaluated using GPURDD's `map` method. In order to force evaluation of each of these RDD's, the `count` method is used, as in the prior pi estimation tests.

In order to validate that GPURDD is behaving properly, the partially reduced RDDs that are created as a result of calling `map(count_and_reduce)` are compared.

Chapter 6

EXPERIMENTS

In order to determine the types of workloads that can be accelerated using GPUMap, performance benchmarks are performed using some of the tests described in Chapter 5. 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.

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 chapter will discuss the different benchmarks and perform a performance evaluation. Overall, the results demonstrate that both GPUMap and GPURDD are not viable for $\mathcal{O}(n)$ algorithms but provide considerable performance improvements to algorithms with larger time complexities.

6.1 N-body Benchmark

As discussed in Section 5.3, the n-body test is an all-pairs n-body implementation based on the outer loop approach from [17]. This test is also used as a performance benchmark for GPUMap. This is an $\mathcal{O}(n^2)$ 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 6.1 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.

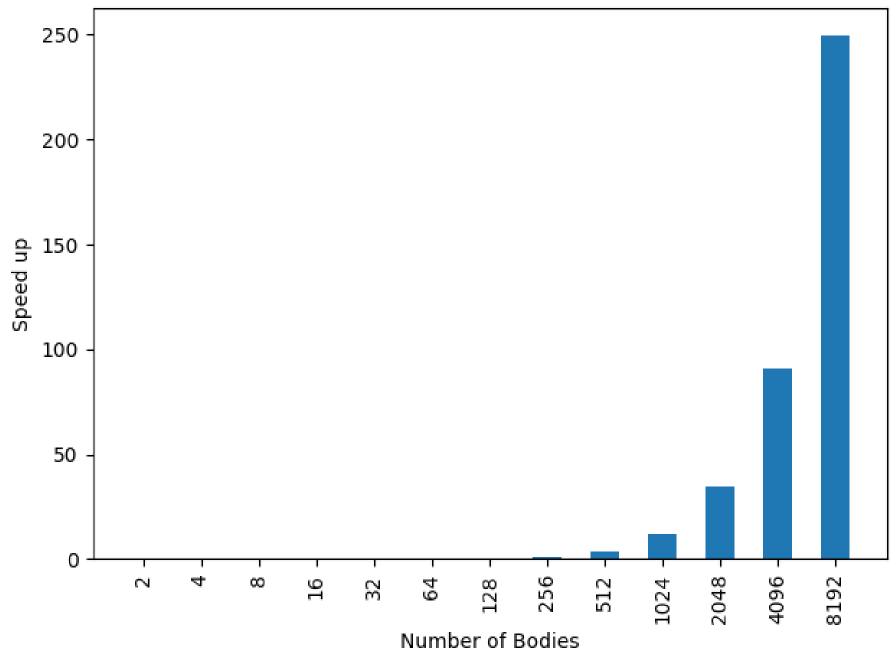

Figure 6.1: Speed-up using GPUMap with the N-body benchmark

The reason for this increase in performance is that because this all-pairs n-body simulation is an $\mathcal{O}(n^2)$ 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 6.2 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.

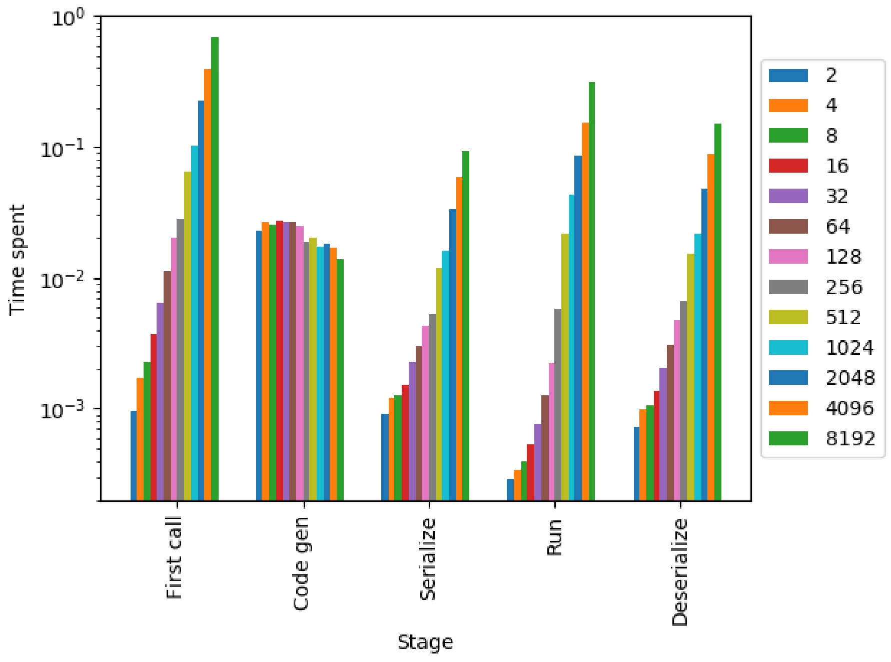

Figure 6.2: Stage duration using GPUMap with the N-body benchmark

As expected, due to the fact this algorithm is $\mathcal{O}(n^2)$, 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 $\mathcal{O}(n)$. 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.

6.2 Bubble Sort Benchmark

As discussed in Section 5.4, the bubble sort test is a simple test that involves sorting multiple lists of randomly generated integers in parallel using bubble sort, an $\mathcal{O}(n^2)$ 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 6.3 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

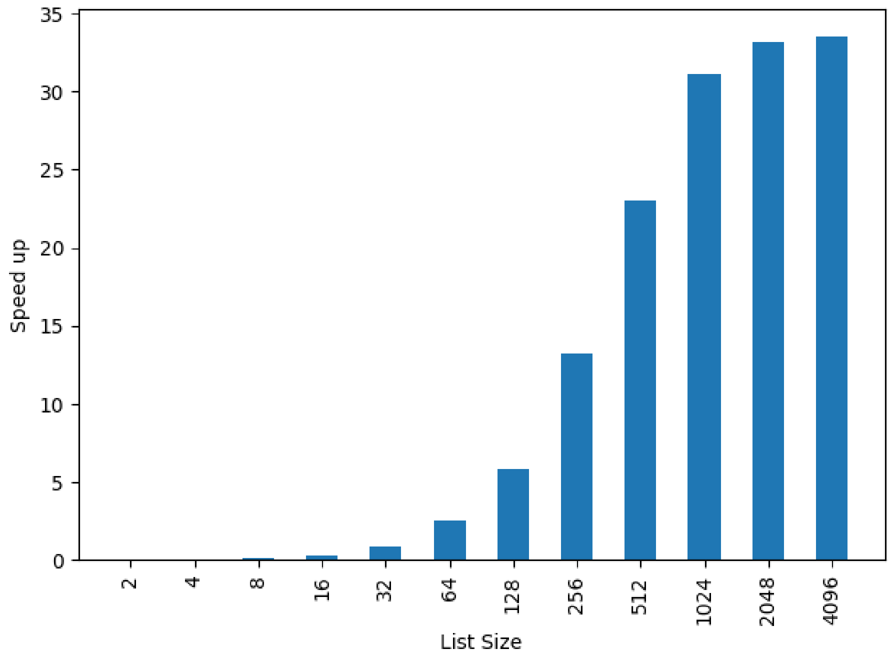

Figure 6.3: Speed-up using GPUMap to sort 1000 lists using bubble sort

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 6.4 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 $\mathcal{O}(n^2)$ 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 $\mathcal{O}(n)$, 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.

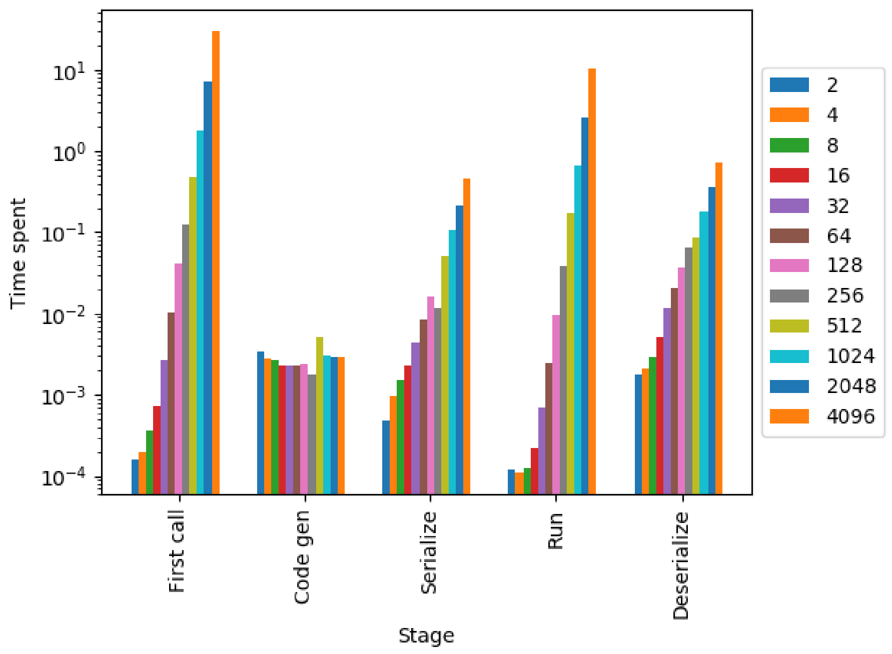

Figure 6.4: Stage duration using GPUMap to sort 1000 lists using bubble sort

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 6.5.

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 $\mathcal{O}(n^2)$ 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.

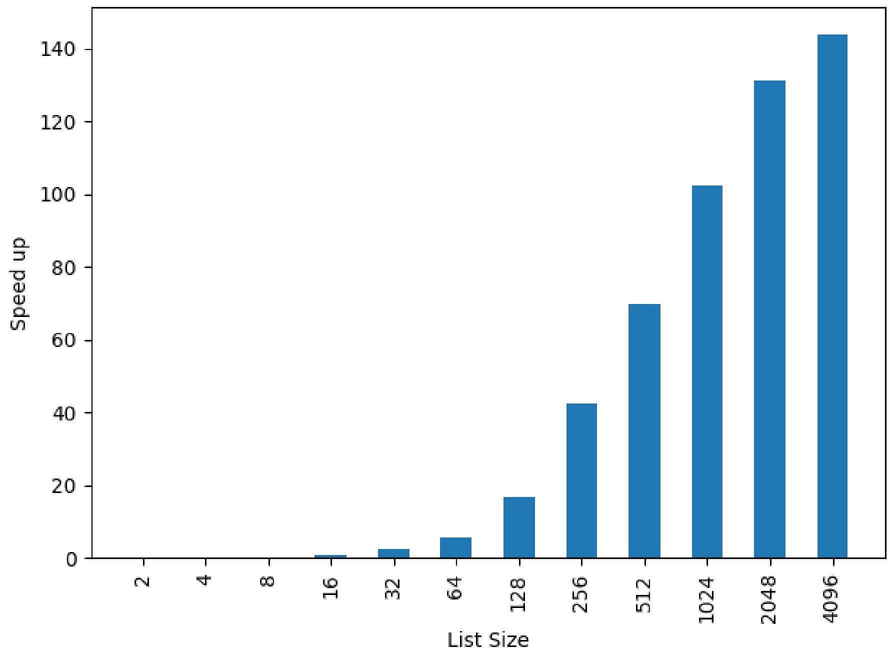

Figure 6.5: Speed-up using GPUMap to sort 5000 lists using bubble sort

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.

6.3 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 $\mathcal{O}(nlog^2n)$ [15]. 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 $\mathcal{O}(m)$ 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 6.6, 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.

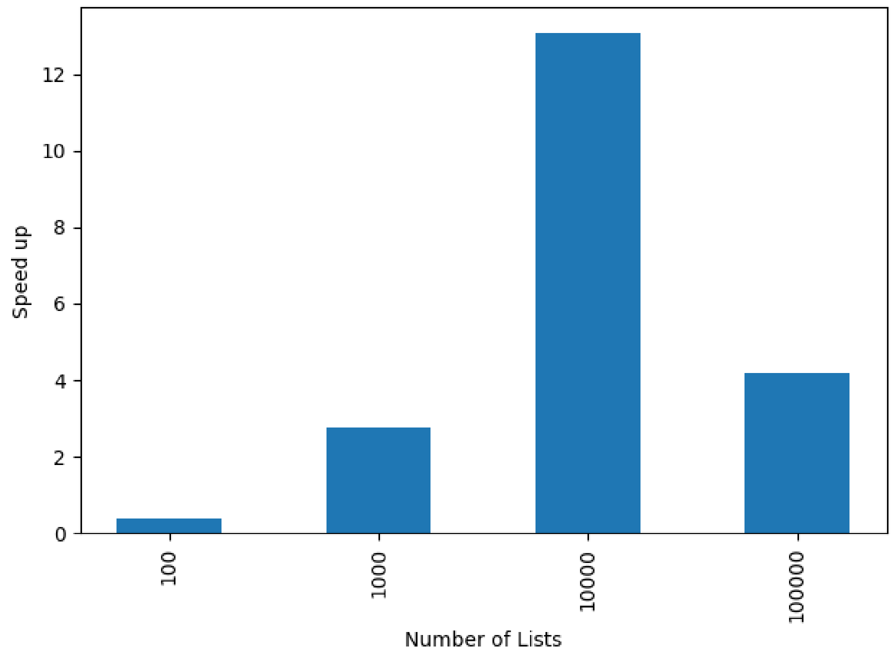

Figure 6.6: Speed-up using GPURDD to sort lists of size 10000 using shellsort

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 6.7.

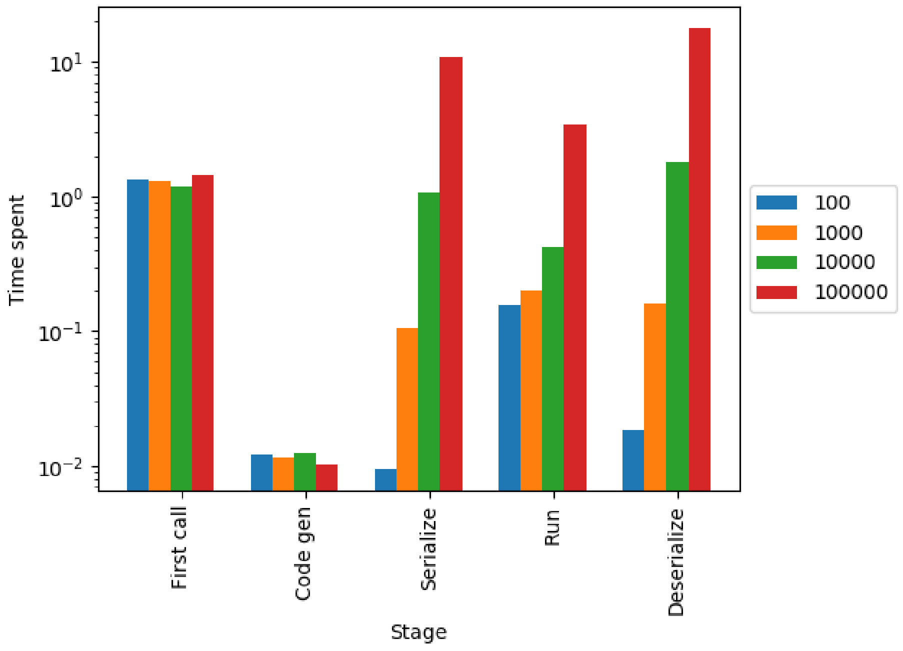

Figure 6.7: Stage duration using GPURDD to sort lists of size 10000 using shell sort

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 $\mathcal{O}(m)$ on the number of lists, the serialization costs cannot be outweighed with processing time.

6.4 Pi Estimation Benchmark

The pi estimation tests, previously discussed in Chapter 5.6, are used to test the functionality as well as the performance of GPURDD. Each of these tests consists of an $\mathcal{O}(n)$ algorithm that checks whether n randomly chosen points lie inside a unit circle. Each type of benchmark is tested with different sized data sets. The data sets used for running each type of benchmark consist of RDDs of randomly chosen points, ranging in size from 10,000 to 1,000,000,000 by powers of 10. Each benchmark divides the RDDs into 10 partitions, one partition for each machine in the Spark cluster.

Due to the fact that these different algorithms are $\mathcal{O}(n)$, GPUMap is not able to provide improved performance due to the high cost of serialization. The remainder of this section will discuss the different pi estimation tests and provide a more detailed performance evaluation for each.

6.4.1 Using Map

The first of the pi estimation benchmarks involves using GPURDD's and RDD's `map` methods to map each point in the RDD to either a 1 or a 0 depending on whether the point is inside or outside the unit circle, respectively. The performance of these map operations is measured and compared.

As mentioned previously, this is an $\mathcal{O}(n)$ algorithm, and for each random point in the data set, the constant number of steps to be performed is very small. This means that GPURDD has difficulty competing with a traditional CPU implementation, as the serialization and deserialization cost is also $\mathcal{O}(n)$.

Figure 6.8 shows that no matter the input data size, GPURDD is not able to achieve better performance than a traditional CPU implementation, for the simple reason that the constant number of steps required to process each element directly is

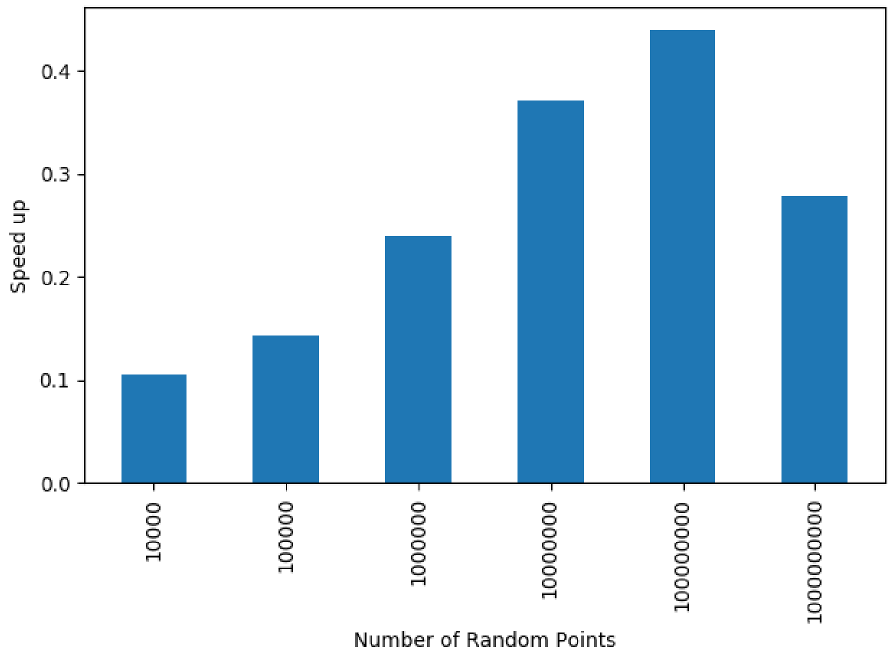

Figure 6.8: Speed-up using GPURDD to check whether points are in the unit circle

smaller than the constant number of steps required to serialize and then deserialize each element. In addition, the result from the largest data set demonstrates considerable performance decreases because the data in a single partition is so large that the Spark workers experience excessive page faults due to the added memory usage of GPUMap. Storing the data set in memory along with its serialized form and other metadata used by GPUMap is able to consume the entire available memory of the worker nodes and nearly one-third of each worker's 64 GB swap partition.

Figure 6.9 shows the average durations of each GPUMap stage across all the systems in the cluster for each size data set. The figure shows that, regardless of the data set size, the serialization and deserialization costs outweigh the kernel run time. For many sizes of data sets, the serialization and deserialization costs are entire orders of magnitude larger than kernel duration. The first call stage has a constant duration because for each call of GPURDD's `map`, the function is applied to the first element of the RDD, which takes the same amount of time regardless of data size.

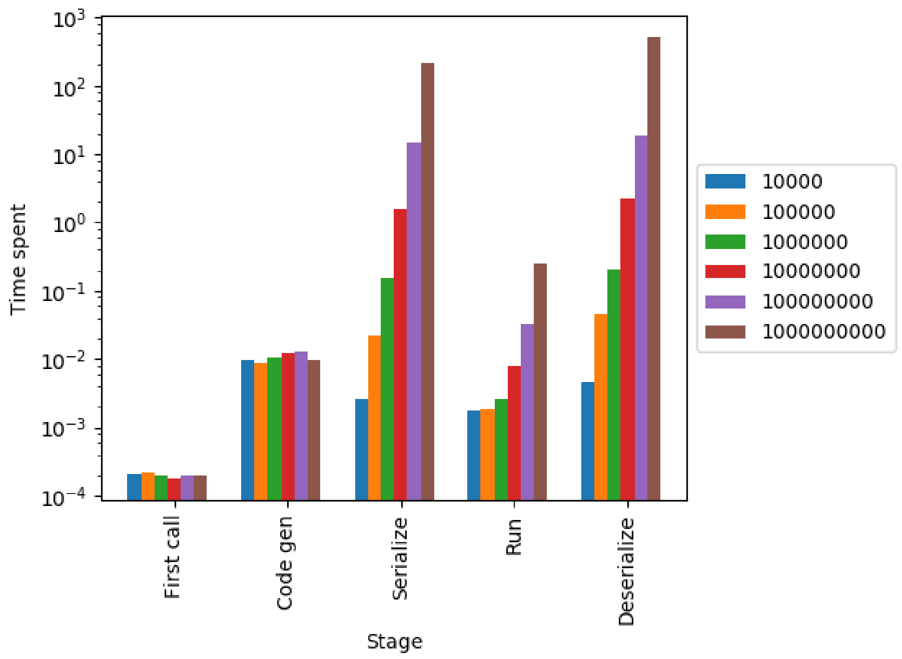

Figure 6.9: Stage duration using GPURDD to check whether points are in the unit circle

Similarly, because the code generated is independent of the data size, the duration of the code generation stage does not increase with data set size.

6.4.2 Using Filter

The second pi estimation benchmark uses GPURDD's and RDD's `filter` methods to remove points that lie outside the unit circle, and measuring the performance of these operations. Once the points outside the unit circle are removed, the remaining points can be counted in order to estimate pi. This is also an $\mathcal{O}(n)$ algorithm, as each point needs to simply be checked whether the point lies inside the unit circle.

The results of this benchmark, shown in Figure 6.10 are very slightly worse than the results of the benchmark using GPURDD's `map` because GPURDD's `filter` needs to do $\mathcal{O}(n)$ more work to prune the results after the filtering function has been applied to each element in the RDD. However, this extra work is not reflected in the execution

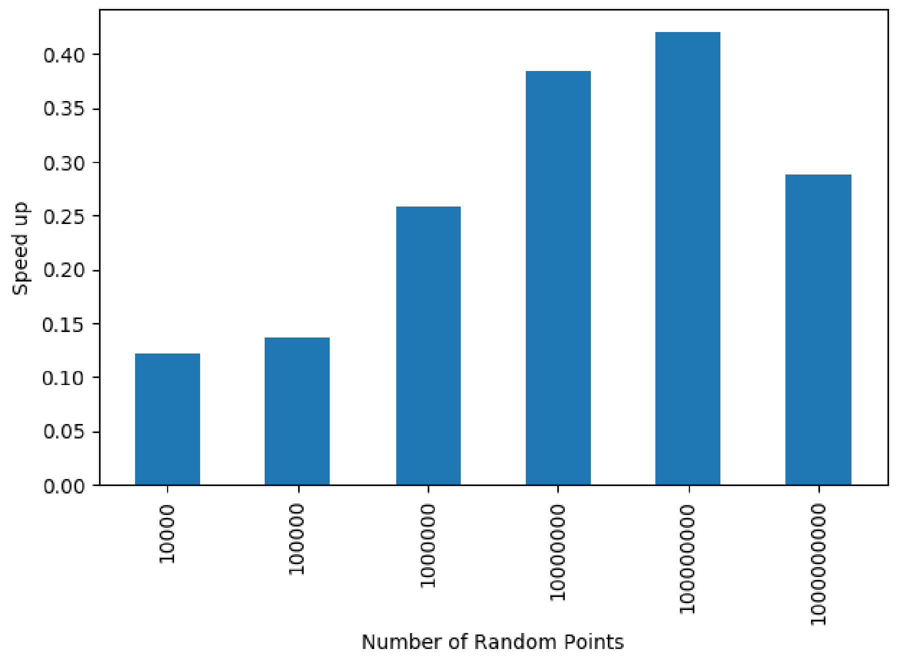

Figure 6.10: Speed-up using GPURDD to filter points that are not located in the unit circle

time of GPUMap, as the work is done outside of GPUMap.

6.4.3 Using Parallel Reduction

The final pi estimation benchmark divides the data set of n randomly chosen points into lists of size 100. Then for each list, a count of the points that lie inside the unit circle is calculated, using GPURDD and RDD's `map` methods and measuring their performance. Although this is still a $\mathcal{O}(n)$ algorithm, the motivation behind this test is to try to give the GPU threads slightly more work to do, in hopes of being able to slightly outweigh the serialization cost. This approach is similar to the partial-reduction technique used in GPMR [18].

The test produces almost the exact same speed-ups as the first pi estimation benchmark that uses map to map each random point to a 0 or a 1. However, in this case, fewer GPU threads are used to check whether points belong in the unit circle.

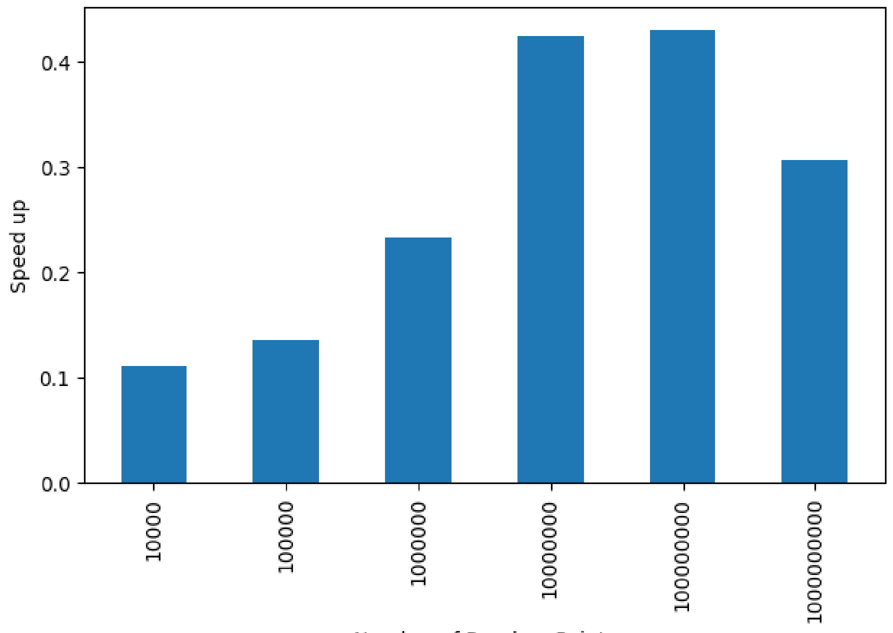

Figure 6.11: Speed-up using GPURDD to count the points that are located in the unit circle

However, there is no considerable change in the kernel runtime between the map test and this test, as can be seen by comparing the run stages from Figure 6.9 and Figure 6.12. As a result, by using this method of grouping points in a list and doing a partial reduction, outweighing the serialization cost is still not possible. In addition, because the data size is the same for this test and the map test, serialization and deserialization take approximately the same time. Furthermore, because serialization and deserialization are easily the longest stages of GPUMap for the pi estimation tests, these stages dictate the overall duration of GPUMap. Thus, this test and the map test can be expected to achieve similar performance.

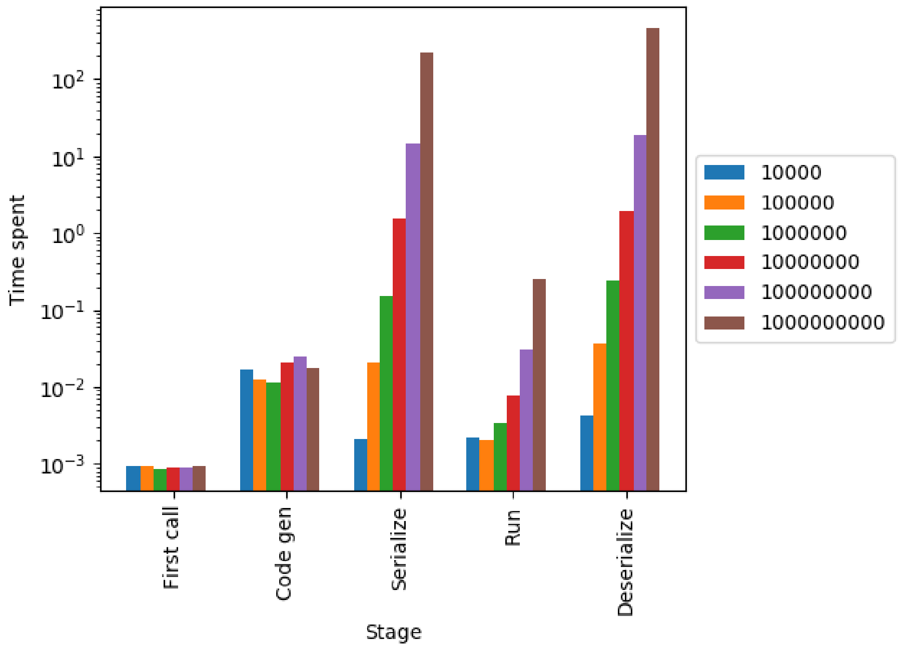

Figure 6.12: Stage duration using GPURDD to count the points that lie in the unit circle

Chapter 7

LIMITATIONS

Although GPUMap aims to be transparent, GPUMap does not support all the essential Python features and unfortunately restricts the possible programs that can be accelerated using GPUMap. Some of these limitations are possible to address with additional implementation time or by redesigning components of GPUMap. However, some of these limitations may not be possible to overcome due to CUDA constraints and hardware constraints.

Many of the limitations of GPUMap stem from the facts that GPUMap lacks dynamic allocation, GPUMap's runtime inspection does not capture everything, and GPUMap's serialization could be revised. There are also other limitations in GPUMap, such as lack of exceptions and lack of synchronization.

7.1 Lack of Dynamic Allocation

A primary reason for many of GPUMap's constraints is that dynamic allocation is currently not used in GPUMap due to the fact that regularly allocating memory dynamically from all threads at once does not result in good performance [5]. Using dynamic allocation would ease the implementation of many more Python built-in language features, as well as help mimic Python behavior more accurately.

Commonly used Python features that depend on dynamic allocation are data structures such as lists, sets, and dicts. The reason dynamic allocation is useful for data structures is because they can be of arbitrary size. In C++, storing entire arbitrary length structures on the runtime stack or returning them from functions is not possible. In order to help improve the usability of GPUMap, GPUMap allows very limited use of lists. Lists can only be passed as closure bindings to the top-level function or as elements of the input list, as these are dynamically allocated on the GPU from the host. They cannot be constructed in the code and they cannot be returned, as the final size of a list is not known at the time of list creation. However, with dynamic allocation, lists could be as big as necessary because passing or returning a reference to the list object is sufficient. Using GPUMap, lists can also be accessed, iterated over, and the contents can even be modified. However, items cannot be added or removed from a list, as the list must be constant size. In addition, lists are the only data structures that are even partially supported by GPUMap.

Currently, GPUMap requires entire objects, including their constituent objects, to be stored in contiguous memory. In order to navigate these objects, a class definition is used. As a result, all objects of a class must contain the same types in their corresponding fields and all member data must be present in each object. In addition, objects cannot be contain arbitrary types, as the types are forced by the class definition. Dynamic allocation could help improve the flexibility of GPUMap by removing the constraint that objects must be stored in contiguous memory, allowing them to be scattered across the heap. Scattering objects would allow much better ability to mimic Python objects, as all variables could be references. However, with dynamic allocation, a basic garbage collector will also be necessary.

Currently, GPUMap requires that local variables, object fields, and list items can only be reassigned to objects of the same type. However, dynamic allocation can allow local variables, object fields, and data structures to store references to object containers that contain objects of arbitrary type, which would make GPUMap much more flexible. This would also lift the restriction that objects cannot contain objects of the same type, as objects would only need to contain references.

Allowing objects of arbitrary size using dynamic allocation can also expand the possible data types that can be used. Currently, GPUMap only supports nested objects that consist of integers, floats, and booleans, which causes all objects to have a well-defined size. However, if storing, passing, and returning references to objects rather than entire object copies is possible, objects of arbitrary size can easily be stored, passed, and returned.

7.2 Limitations from Runtime Inspection

In GPUMap, runtime inspection is performed by analyzing all functions and methods that are called as a result of applying the top-level function to the first item in the list passed to GPUMap. However, if certain functions or methods are only called for some of the list items, then these functions or methods may be missed during function tracing. As a result, these missed functions or methods may not be translated and will cause issues when trying to compile the generated code. Furthermore, due to the way information is stored regarding each called function or method, calling functions or methods with different argument types or returning objects of different types from the same function is not possible.

GPUMap's ability to inspect the structure of objects also results in further limitations. Due to the fact that one Class Representation is used to represent all objects of a class, all objects of a class must conform to this Class Representation to be properly used with GPUMap. However, this does not effectively represent Python objects because objects of the same class may actually differ greatly in structure.

Another problem caused by runtime inspection is that doing function call tracing is not possible during debugging and unit testing. This is because the functions used by the debugger and the unit testing harness are also traced by function call tracing. Some exclusion rules may be created to ignore these functions, but accommodating different debuggers and unit testing harnesses may be difficult. In addition, there is no effective way to debug the code that is run on the GPU.

7.3 Limitations from Serialization

Serialization seems to be the current biggest performance bottleneck with GPUMap. As is common with GPGPU programming, if the data size is too large and the work done on the data is too small, serialization time outweighs processing time. As a result, using GPU-acceleration is not useful in such cases. However, there is certainly room for improvements to the serialization method used by GPUMap. Currently, GPUMap makes use of Python's `struct` module for serialization in order to for serialized objects to be directly compatible with CUDA.

One limitation that is caused by using `struct` is that `struct` can only serialize primitive data. This means that objects must be made only of primitives and these primitives must be recursively extracted in the correct order from each object during serialization. Then during deserialization, `struct` is used to deserialize the data and then the deserialized data must be recursively inserted into each object in the correct order. There may be more optimal ways to serialize and deserialize the data that do not involve recursive extraction and insertion.

Currently, in order to help improve performance, the translated code operates directly on the data serialized using `struct`. As a result, objects are contained in contiguous memory. Recursively allocating objects and inserting pointers into the serialized versions of other objects in order to create scattered objects is possible. However, creating such scattered objects is not viable because this will require several CUDA allocations for each object, which may result in decreased performance. Contiguous objects also means that objects must have all fields present, otherwise they will not match up with the C++ class definition used to access them. Furthermore, having contiguous objects introduces difficulty when including arbitrary size objects inside of other objects, as an arbitrary size object cannot match up with a C++ class definition, which has a fixed size. Serializing the data in a different format may be more optimal if objects on the GPU are scattered across the heap rather than contained in contiguous memory.

7.4 Other Limitations

Currently, there are many built-in functions in Python that do not require dynamic allocation and may be implementable in CUDA C++. However, without the C++ standard library available, implementing these functions from scratch can be time consuming. Thus, due to time constraints, many built-in functions are not implemented in GPUMap. There are also some types of Python syntax that are currently not able to be translated by GPUMap. More information about the supported Python syntax can be found in the language translation validation tests in Appendix A.

In addition, exceptions are not supported in CUDA, and so they are not currently supported in GPUMap. However, implementing exceptions from scratch and forwarding them back to Python may be possible using a method similar to Rootbeer [12].

Another limitation in GPUMap is that synchronization is not supported. Synchronization may be particularly useful when accessing data passed in using the top-level function's closure bindings, which is stored in the GPU's global memory. Currently, the only time GPU threads are guaranteed to be synchronized is between calls to `gpumap`.

Chapter 8

FUTURE WORK

There are a variety of improvements that can be made to improve the performance and flexibility of GPUMap. Some of these performance and flexibility improvements require simply adding features to the existing feature set of GPUMap. Other improvements may require considerable research and may require entire components of GPUMap to be revised. In addition, new developments in the field of GPGPU computing may be able to be utilized in order to simplify the implementation, improve the flexibility, or improve the performance of GPUMap. Overall, these improvements can be grouped into three categories: improving serialization, improving code generation, and adding language features.

8.1 Improved Serialization

Currently, the most significant bottleneck in GPUMap is serialization. Serialization takes considerably more time than code generation and kernel execution for many types of problems. Thus, in order to improve the performance of GPUMap, this is one of the most important problems to tackle. There are a variety of approaches to improving serialization, which will be discussed in the remainder of this section. Some of these serialization improvements may be able to incorporate the use of emerging technology, such as unified memory and direct memory access.

8.1.1 Multithreaded Serialization

Perhaps the most straightforward method of considerably improving serialization performance is to spread the workload across multiple cores. The current serialization process involves taking a list of Python objects and recursively examining the fields of each object in the list. This is an embarrassingly parallel workload because each object can be recursively examined and serialized independently of any other object in the list, and the results can simply be concatenated afterwards.

However, speeding up a program using mulithreading is slightly more complicated in Python. Because Python's memory management is not thread safe, Python must have a global interpreter lock, meaning that threads cannot execute concurrently. Thus, making a multithreaded solution will not yield any performance increase. In Python, a common option is to create a multiprocess application instead of a multithreaded application. Python provides an easy way to create a multiprocess application using a process pool. Instead of using `map` to compute the serialized version of each object in the list, `multiprocessing.Pool`'s `map` method can be used. Creating a `Pool` instantiates a specified number of Python interpreters that are idle and ready to receive code and data to operate on.

In addition, serialization may be able to be done asynchronously from function tracing and code generation, as serialization is not dependent on either task, which can further speed up GPUMap's preparation prior to running the kernel.

8.1.2 Serialize Only Necessary Data

Currently, GPUMap serializes all the data associated with an object. However, it is possible that not all of this data is used by the code executed on the GPU. After examining the ASTs of the code that is to be translated to run on the GPU, it can be determined which fields of which types of objects are necessary to be serialized. This means that the unused fields can be pruned from the Class Representations so they are never serialized or deserialized. This will reduce the work needed to be done during serialization and deserialization, which can improve performance.

In addition, serializing only the necessary data will slightly increase the flexibility of GPUMap, because even objects that contain unsupported data types can still be used with GPUMap, as long as these unsupported members of the objects are not referenced in the code that is run by GPUMap.

8.1.3 Cache Serialized Data

If multiple of the same calls are made to `gpumap` subsequently, repeatedly serializing and deserialize the same list occurs. However, GPUMap's performance may considerably improve if this repeated serialization can be avoided altogether by allowing the list to reside in GPU memory between calls to `gpumap`.

The list should only be allowed to reside on the GPU if the list has not been modified by any Python code between the subsequent calls to `gpumap`. Unfortunately, detecting whether the list was modified between the `gpumap` calls may be difficult. One possible solution requires the programmer to specify whether they would like to cache the list on the GPU so that the list can be re-used in subsequent calls to `gpumap`, requiring the programmer to understand the meaning of caching a serialized list on the GPU. However, this approach does not allow the GPGPU programming to be completely abstracted.

8.1.4 Unified Memory

In CUDA 6.0, NVIDIA released an improvement to the basic GPGPU programming model called unified memory [10]. Unified memory allows host memory to be accessed by the host and the GPU transparently, without needing to do any explicit serialization or deserialization. According to NVIDIA, unified memory allows complex data structures that are connected using pointers, such as linked lists, to be manipulated by both the host and the GPU without needing to be copied back and forth.

Currently, PyCUDA does not have proper support for unified memory and only provides experimental support for unified memory involving NumPy arrays. However, in the future if fully featured unified memory is available, traversing Python data structures and objects on the GPU without needing to convert these objects into a GPU-friendly format may be possible. This means that spending time recursively examining and serializing Python objects and data structures may be avoided completely, which can improve performance. Instead, by implementing logic to traverse these objects and data structures the way Python does in CUDA C++, the objects can be used as-is. However, currently, CUDA's unified memory requires programmers to allocate space in a CUDA managed memory block. This means that forcing Python to allocate memory for these data structures in such a managed memory block may be difficult.

In the future, if manipulating host memory from the GPU becomes even easier, such as by allowing direct memory access (DMA) to the host memory from the GPU, manipulating Python objects and other data structures from the GPU may also become easier.

In addition to potentially increasing performance, unified memory may increase the flexibility of GPUMap because of the possibility to manipulate objects that are not easily serializable using the current method employed in GPUMap.

8.2 Improved Code Generation

Currently, the method used for code generation is one of the causes of GPUMap's limited flexibility. As discussed in Chapter 7, each function can only have one set of input types and only one return type due to GPUMap's runtime inspection. Furthermore, objects of the same class must have the same data types in their corresponding fields. Another limitation of the function tracing technique is that there are possible paths that are taken through the code that cannot be examined by a single execution of the code. Improvements to GPUMap can improve the flexibility of GPUMap by addressing these issues. In addition, there is some unused code that is generated by GPUMap and it may be possible to avoid generating this code altogether.

8.2.1 Bytecode Inspection

Existing projects that attempt to provide easier GPU acceleration in higher-level languages, such as Aparapi [3], Rootbeer [12], and Numba [7] inspect bytecode to aid with code translation. Incorporating bytecode inspection into GPUMap's code generation may be useful because bytecode inspection can help examine all possible paths through the code. In addition, examining the compiled Python bytecode may be effective because Python bytecode contains simpler operations that may map better to CUDA C++ constructs.

8.2.2 Decoupled Types

Removing type information from Class Representations and Function Representations, and attempting to enforce these at runtime rather than at CUDA compile time may be useful in improving the flexibility of GPUMap. In order to enforce types at runtime, converting the program into some sort of intermediate representation and implementing a CUDA interpreter for this intermediate representation may be necessary. In addition, incorporating bytecode inspection in creating this intermediate representation may be helpful. Enforcing types at runtime may cause a decrease in performance but if doing so is possible, the flexibility of GPUMap will drastically be increased.

One possible method to enforce type information at runtime is to create a variety of CUDA-friendly data structures that describe the available fields and methods of each object. Determining the type of an object at runtime by using a reference container which specifies the type of the object referred to by the reference may be possible. Every time the reference is accessed, the reference can be treated as the appropriate type, allowing only the correct methods and fields to be accessed. This sort of improvement will require a complete restructure of the way that GPUMap works.

8.2.3 Reducing Unnecessary Code

Currently during GPUMap's code generation, there is some unused code that is generated because GPUMap does not check whether the code will actually be used. For each function, if the function has n non-primitive parameters, 2^n different function definitions are created to accommodate for whether rvalue expressions or lvalue expressions are passed to the function. However, it is likely that not all of these different definitions of the function will be used. By examining the spots that rvalues are passed in the code when inspecting the ASTs, the code generator can determine which parameters of which functions need to be rvalue references.

Reducing the number of declarations of the same function can slightly improve performance during code generation and compilation by decreasing the time taken to generate and compile the code.

8.3 Language Features

There are many language features that are currently unsupported while using GPUMap, which drastically reduces the usability of GPUMap. One factor that limits the usability of GPUMap is that many built-in functions and fully-featured data structures are not available for use because they depend on dynamic allocation. In addition, exceptions are not supported in the code run with GPUMap, making identification of errors or special cases and handling them more difficult. Furthermore, NumPy is very commonly used in scientific applications, but usage of NumPy is currently not supported, preventing these applications from running with GPUMap. Adding these features will greatly improve the flexibility of GPUMap because GPUMap will be able to run a wider range of programs.

8.3.1 Dynamic Allocation

Currently, dynamic allocation is not used by GPUMap due to poor performance when dynamically allocating memory from many threads simultaneously [5]. If the dynamic allocation scheme used by CUDA threads is improved or if an alternative parallel allocation scheme is used, dynamic allocation may become viable to be extensively used throughout GPUMap. With the use of dynamic allocation, implementation of some sort of simple garbage collection, such as reference counting, is necessary.

In addition to mimicking Python's reference-passing behavior better, dynamic allocation would allow for easier implementation of dynamically-sized data structures. These data structures can then be used by code that is supplied by the programmer, but they can also be used to implement a considerable portion of Python's built-in functions that depend on the use of such data structures. These built-in functions and data structures can be implemented directly in CUDA C++ and included at compile time of the CUDA code. However, implementing some of the built-in functions in code that can be translated by GPUMap's language translation component may also be possible.

The addition of dynamic allocation and garbage collection may slightly decrease the performance of GPUMap, but it will greatly improve the flexibility of GPUMap by allowing the usage of a much wider variety of data structures.

8.3.2 Exceptions

GPUMap unfortunately does not support exception handling as CUDA does not natively support exceptions. However, it may be possible to implement exceptions in a manner similar to Rootbeer [12], which passes around a long where each bit is a flag representing an exception. This long is passed in and out of every function, allowing callees to set the flags and callers to consume them.

Using such a method, a data structure that can be used to keep track of exceptions must be allocated. The data structure provides space for code to mark that an exception happened. In addition, the data structure allows other code to check whether an exception happened so that the exception can be handled appropriately. If the exception is not handled, the exception should propagate back to the host. This means that the data structure should be able to be deserialized and processed by the host so that any uncaught exceptions can ultimately be raised in Python. In order to determine which exception was thrown by the GPU code on the host, a mapping between the exception's representation in the data structure and the exception itself must be created.

Because exception usage will likely decrease the performance of GPUMap, exception handling should be optionally disabled. However, exception handling should be available as a feature in order to increase GPUMap's flexibility because exceptions may sometimes be necessary to express certain programs.

8.3.3 NumPy Support

Scientific applications commonly make heavy usage of NumPy, as NumPy is useful for quickly processing large batches of numbers in the form of arrays and matrices. Adding NumPy support will greatly improve the flexibility of GPUMap because

GPUMap will be able to run many scientific applications. In addition, incorporating NumPy support into GPUMap may not be particularly difficult due to the fact that NumPy stores its data in C-style arrays, which are directly compatible with CUDA C++. However, re-implementing the features of NumPy in CUDA would be necessary, which may be very time consuming as NumPy is a large software package. Fortunately, re-using the existing NumPy source code may be possible, as a considerable portion of NumPy is written in C.

In addition to improving flexibility by allowing scientific applications to run, usage of NumPy with GPUMap should be able to slightly improve GPUMap's performance because no extra work is necessary prior to copying the NumPy arrays to the GPU during serialization.

Chapter 9

CONCLUSION

This thesis presents GPUMap, which is a proof-of-concept Python map function that aims to abstract away all the GPU programming and allows programmers to GPU-accelerate certain types of programs with no extra effort or knowledge. GPUMap works with normal, object-oriented Python code and does not require users to do any serialization or write kernels. This thesis 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 order to attempt to provide the programmer with a GPU-accelerated map function that does not require any extra effort to use.

When the programmer calls `gpumap(f, L)` on a function f and a list L, GPUMap performs runtime inspection by calling f on the first element of the list L. This runtime inspection allows GPUMap to determine the structure of the classes of objects in the input list, objects in the output list, objects included in f's closure variables, and any other objects that are necessary. In addition, the runtime inspection performs function tracing to determine which functions and methods are called and the argument types and return types of each function or method.

The information collected about each necessary class is used to generate corresponding CUDA C++ class definitions. Next, CUDA C++ function and method definitions for each of the necessary methods are generated using the information collected during runtime inspection as well as information collected from examining the abstract syntax trees (ASTs) of each function or method. Finally, a CUDA kernel template is filled in using information regarding the structure of the input and output lists.

Serialization of the first object is not necessary as f was already applied to it to produce the first item of the output list. Objects in the remainder of the input list, as well as f's closure variables, are serialized to match their corresponding CUDA C++ class definitions by extracting all their primitive data and arranging their primitive data in C-style structs, which also correspond to C++ objects. Recursive examination is performed on each object to obtain a list of the primitive fields contained in each object. Once a list of primitives for each object is created, it is serialized into a C-style struct using Python's `struct` module. Then, each serialized object from the input list is concatenated to produce a C++ array of these objects. In addition, if f includes any lists in its closure variables, those are concatenated in a similar manner. Each serialized list or object is then copied to the GPU.

Then, the kernel, along with any generated code is executed to transform the serialized portion of the input list into the output list. Once kernel execution has finished, the input list, any closure variables of f, and the output list are copied back to the host in their serialized form. These serialized objects are deserialized by using the `struct` module to unpack the serialized objects into lists of primitives. The primitives belonging to the input list items as well as any closure variables of f are then re-inserted back into their originating objects. The primitives belonging to the items in the output list are inserted into newly created objects that are copies of the first item from the output list, which was created during runtime inspection.

This approach unfortunately imposes several limitations on the flexibility of GPUMap. Due to the way information about classes is stored and used to generate C++ class definitions, all Python objects of the same class must have the same types in their corresponding fields. This reduces the flexibilty of GPUMap because sometimes Python objects of the same class may actually differ greatly in structure. Furthermore, due to the way information about functions and methods is stored and used to generate C++ function and method definitions, functions and methods must have consistent argument types and return types. Further research is necessary in determining how to relax these restrictions, as doing a straightforward CUDA C++ translation requires all of the type information in advance.

There are also many 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.

In order to see whether the simplified GPU-acceleration provided by GPUMap can be effectively integrated into Apache Spark, GPURDD was created. In the case of GPURDD's `map` and `foreach` methods, GPUMap is simply used to apply a given function to each item in a partition. However, 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. Because GPURDD incorporates GPUMap, all of the limitations of GPUMap carry over to GPURDD.

Although GPUMap is not as flexible as originally intended, GPUMap manages to achieve a considerable performance improvement for certain types of programs. For compatible algorithms that have considerably larger time complexity than $\mathcal{O}(n)$ and a large enough data set, GPUMap may be able to provide performance improvements. During benchmarking of GPUMap using the $\mathcal{O}(n^2)$ n-body algorithm, GPUMap was able to produce a speed up of 249 times on the largest data set. However, for algorithms with $\mathcal{O}(n)$ time complexity or better, GPUMap will likely not yield any considerable speed-ups due to $\mathcal{O}(n)$ serialization complexity. In the best case scenario of the pi estimation Spark benchmark, an $\mathcal{O}(n)$ algorithm, GPUMap was still not able to perform nearly as well as a normal CPU implementation, and caused about a 2.5 times slowdown. Fortunately, 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.

Due to GPUMap's lacking flexibility, GPUMap will need considerably more work in order to be truly useful. However, as a proof-of-concept, GPUMap demonstrates that it may be useful in speeding up Python code in particular situations. In addition, GPUMap demonstrates that it is possible to subtly incorporate GPU acceleration into existing batch operations that programmers already know how to use. Eventually, if GPU-acceleration is subtly incorporated into such batch operations, a time may come where programming languages offer a seamless hetereogeneous computing experience.

BIBLIOGRAPHY

[1] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th international conference on Machine learning*, pages 104–111. ACM, 2008.

[2] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[3] G. Frost and M. Ibrahim. Aparapi documentation. *AMD Open Source Zone*, Mar 2015.

[4] M. Grossman, M. Breternitz, and V. Sarkar. HadoopCL: MapReduce on distributed heterogeneous platforms through seamless integration of Hadoop and OpenCL. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1918–1927, May 2013.

[5] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W.-m. Hwu. Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1134–1139. IEEE, 2010.

[6] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157–174, 2012.

[7] S. K. Lam, A. Pitrou, and S. Seibert. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler*

Infrastructure in HPC, LLVM '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM.

[8] P. Li, Y. Luo, N. Zhang, and Y. Cao. HeteroSpark: A heterogeneous CPU/GPU Spark platform for machine learning algorithms. In *Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on*, pages 347–348, Aug 2015.

[9] Y. Lin, S. Okur, and C. Radoi. Hadoop+Aparapi: Making heterogenous MapReduce programming easier. 2012.

[10] NVIDIA Corporation. CUDA C programming guide. 2017.

[11] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[12] P. C. Pratt-Szeliga, J. W. Fawcett, and R. D. Welch. 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*, pages 375–380. IEEE, 2012.

[13] Python Software Foundation. Data model. *The Python Language Reference*, Mar 2017.

[14] Python Software Foundation. Numeric types int, float, complex. *The Python Language Reference*, Mar 2017.

[15] R. Sedgewick. Analysis of shellsort and related algorithms. In *European Symposium on Algorithms*, pages 1–11. Springer, 1996.

[16] O. Segal, P. Colangelo, N. Nasiri, Z. Qian, and M. Margala. SparkCL: A

unified programming framework for accelerators on heterogeneous clusters. *CoRR*, abs/1505.01120, 2015.

[17] A. Šinkarovs, S.-B. Scholz, R. Bernecky, R. Douma, and C. Grelck. 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):952–971, 2014.

[18] J. A. Stuart and J. D. Owens. Multi-GPU MapReduce on GPU clusters. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1068–1079. IEEE, 2011.

[19] Y. Yan, M. Grossman, and V. Sarkar. JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA. In *European Conference on Parallel Processing*, pages 887–899. Springer, 2009.

[20] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[21] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

APPENDICES

Appendix A

LANGUAGE FEATURE TESTS

This appendix contains the tests used to validate the language translation component of GPUMap, as discussed in section 5.1.

```
from unittest import TestCase

from func_def import FunctionConverter, MethodConverter
from data_model import FunctionRepresentation, MethodRepresentation, ExtractedClasses

import ast

class TestFunctionConverter(TestCase):
    def setUp(self):
        fr = FunctionRepresentation("", [], [], None, None)
        self.fc = FunctionConverter(fr)

    def test_literal_num(self):
        node = ast.parse("1234")
        self.assertEqual("1234", self.fc.visit(node))

        node = ast.parse("12.34")
        self.assertEqual("12.34", self.fc.visit(node))

        node = ast.parse("0x1ABCDEF")
        self.assertEqual("28036591", self.fc.visit(node))

        # G is not a valid hex digit and should not parse
        self.assertRaises(SyntaxError, ast.parse, "0x1ABCDEFG")

        node = ast.parse("0b11010101")
        self.assertEqual("213", self.fc.visit(node))

        # 8 is not a valid binary digit and should not parse
        self.assertRaises(SyntaxError, ast.parse, "0b11010181")

    def test_literal_string(self):
        node = ast.parse("\"1234\"")
        self.assertEqual("\"1234\"", self.fc.visit(node))

        node = ast.parse("\"abcde\"")
        self.assertEqual("\"abcde\"", self.fc.visit(node))

        node = ast.parse("\'12345\'")
        self.assertEqual("\"12345\"", self.fc.visit(node))

        node = ast.parse("\'abcdef\'")
        self.assertEqual("\"abcdef\"", self.fc.visit(node))

    def test_literal_bytes(self):
        node = ast.parse("b\"some_bytes\"")
        self.assertRaises(SyntaxError, self.fc.visit, node)
        node = ast.parse("B\"some_bytes\"")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_list(self):
        node = ast.parse("[]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("[a, b, c]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("[1, 3, 5, 7]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("[\"a\", \"b\", \"c\"]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_tuple(self):
        node = ast.parse("()")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("(a, b, c)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("(1, 3, 5, 7)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("(\"a\", \"b\", \"c\")")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_set(self):
        node = ast.parse("{a, b, c}")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("{1, 3, 5, 7}")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("{\"a\", \"b\", \"c\"}")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_dict(self):
        node = ast.parse("{}")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("{a: 1, b: 3, c: 5, d: 7}")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("{a: \"a\", b: \"b\", c: \"c\"}")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("{\"a\": 1, \"b\": 2, \"c\": 3}")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_ellipsis(self):
        node = ast.parse("...")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_name_constant(self):
        node = ast.parse("True")
        self.assertEqual("true", self.fc.visit(node))

        node = ast.parse("False")
        self.assertEqual("false", self.fc.visit(node))

        node = ast.parse("None")
        self.assertEqual("null", self.fc.visit(node))

    def test_name(self):
        node = ast.parse("x")
        self.assertEqual("x", self.fc.visit(node))

        node = ast.parse("val")
        self.assertEqual("val", self.fc.visit(node))

        node = ast.parse("_val")
        self.assertEqual("_val", self.fc.visit(node))

        node = ast.parse("a_val")
        self.assertEqual("a_val", self.fc.visit(node))

        node = ast.parse("Val")
        self.assertEqual("Val", self.fc.visit(node))

    def test_starred(self):
        node = ast.parse("*args")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("*obj.args")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("*obj.gen_args()")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_expr(self):
        """
        This test is covered by:
        test_unary_ops
        test_bin_ops
        test_bool_ops
        test_compare
        test_call
        test_if_exp
        test_attribute
        test_subscript
        test_index
        test_slice
        test_ext_slice
        test_list_comp
        test_set_comp
        test_gen_exp
        test_dict_comp
        """
        pass

    def test_unary_ops(self):
        """
        This test is covered by:
        test_unary_add
        test_unary_sub
        test_not
        test_invert
        """
        pass

    def test_unary_add(self):
        node = ast.parse("+1234")
        self.assertEqual("(+1234)", self.fc.visit(node))

        node = ast.parse("+1234.56")
        self.assertEqual("(+1234.56)", self.fc.visit(node))

        node = ast.parse("+x.a_field")
        self.assertEqual("(+x.a_field)", self.fc.visit(node))

        node = ast.parse("+x.a_method(y)")
        self.assertEqual("(+x.a_method(y))", self.fc.visit(node))
```

```
def test_unary_sub(self):
    node = ast.parse("-1234")
    self.assertEqual("(-1234)", self.fc.visit(node))

    node = ast.parse("-1234.56")
    self.assertEqual("(-1234.56)", self.fc.visit(node))

    node = ast.parse("-x.a_field")
    self.assertEqual("(-x.a_field)", self.fc.visit(node))

    node = ast.parse("-x.a_method(y)")
    self.assertEqual("(-x.a_method(y))", self.fc.visit(node))

def test_not(self):
    node = ast.parse("not True")
    self.assertEqual("(!true)", self.fc.visit(node))

    node = ast.parse("not b")
    self.assertEqual("(!b)", self.fc.visit(node))

    node = ast.parse("not x.a_field")
    self.assertEqual("(!x.a_field)", self.fc.visit(node))

    node = ast.parse("not x.a_method(y)")
    self.assertEqual("(!x.a_method(y))", self.fc.visit(node))

def test_invert(self):
    node = ast.parse("~num")
    self.assertEqual("(~num)", self.fc.visit(node))

    node = ast.parse("~0b1010011")
    self.assertEqual("(~83)", self.fc.visit(node))

    node = ast.parse("~x.a_field")
    self.assertEqual("(~x.a_field)", self.fc.visit(node))

    node = ast.parse("~x.a_method(y)")
    self.assertEqual("(~x.a_method(y))", self.fc.visit(node))

def test_bin_ops(self):
    """
    This test is covered by:
    test_add
    test_sub
    test_mult
    test_div
    test_floordiv
    test_modulo
    test_mat_mult
    test_pow
    test_lshift
    test_rshift
    test_bit_or
    test_bit_xor
    test_bit_and
    """
    pass

def test_add(self):
    node = ast.parse("31 + 57")
    self.assertEqual("(31 + 57)", self.fc.visit(node))

    node = ast.parse("31 + (-57)")
    self.assertEqual("(31 + (-57))", self.fc.visit(node))

    node = ast.parse("31 + x")
    self.assertEqual("(31 + x)", self.fc.visit(node))
```

```
        node = ast.parse("y + 57")
        self.assertEqual("(y + 57)", self.fc.visit(node))

        node = ast.parse("x + y")
        self.assertEqual("(x + y)", self.fc.visit(node))

        node = ast.parse("x + y + z")
        self.assertEqual("((x + y) + z)", self.fc.visit(node))

        node = ast.parse("x.i + y.i")
        self.assertEqual("(x.i + y.i)", self.fc.visit(node))

        node = ast.parse("x.a_method(a) + y.a_method(b)")
        self.assertEqual("(x.a_method(a) + y.a_method(b))", self.fc.visit(node))

    def test_sub(self):
        node = ast.parse("31 - 57")
        self.assertEqual("(31 - 57)", self.fc.visit(node))

        node = ast.parse("31 - (+57)")
        self.assertEqual("(31 - (+57))", self.fc.visit(node))

        node = ast.parse("31 - x")
        self.assertEqual("(31 - x)", self.fc.visit(node))

        node = ast.parse("y - 57")
        self.assertEqual("(y - 57)", self.fc.visit(node))

        node = ast.parse("x - y")
        self.assertEqual("(x - y)", self.fc.visit(node))

        node = ast.parse("x - y - z")
        self.assertEqual("((x - y) - z)", self.fc.visit(node))

        node = ast.parse("x.i - y.i")
        self.assertEqual("(x.i - y.i)", self.fc.visit(node))

        node = ast.parse("x.a_method(a) - y.a_method(b)")
        self.assertEqual("(x.a_method(a) - y.a_method(b))", self.fc.visit(node))

    def test_div(self):
        node = ast.parse("31 / 57")
        self.assertEqual("(31 / 57)", self.fc.visit(node))

        node = ast.parse("31 / (-57)")
        self.assertEqual("(31 / (-57))", self.fc.visit(node))

        node = ast.parse("31 / x")
        self.assertEqual("(31 / x)", self.fc.visit(node))

        node = ast.parse("y / 57")
        self.assertEqual("(y / 57)", self.fc.visit(node))

        node = ast.parse("x / y")
        self.assertEqual("(x / y)", self.fc.visit(node))

        node = ast.parse("x / y / z")
        self.assertEqual("((x / y) / z)", self.fc.visit(node))

        node = ast.parse("x.i / y.i")
        self.assertEqual("(x.i / y.i)", self.fc.visit(node))

        node = ast.parse("x.a_method(a) / y.a_method(b)")
        self.assertEqual("(x.a_method(a) / y.a_method(b))", self.fc.visit(node))

    def test_floordiv(self):
        node = ast.parse("31 // 57")
        self.assertEqual("int(31 / 57)", self.fc.visit(node))
        node = ast.parse("31 // (-57)")
        self.assertEqual("int(31 / (-57))", self.fc.visit(node))

        node = ast.parse("31 // x")
        self.assertEqual("int(31 / x)", self.fc.visit(node))

        node = ast.parse("y // 57")
        self.assertEqual("int(y / 57)", self.fc.visit(node))

        node = ast.parse("x // y")
        self.assertEqual("int(x / y)", self.fc.visit(node))

        node = ast.parse("x.i // y.i")
        self.assertEqual("int(x.i / y.i)", self.fc.visit(node))

        node = ast.parse("x // y // z")
        self.assertEqual("int(int(x / y) / z)", self.fc.visit(node))

        node = ast.parse("x.a_method(a) // y.a_method(b)")
        self.assertEqual("int(x.a_method(a) / y.a_method(b))", self.fc.visit(node))

    def test_modulo(self):
        node = ast.parse("31 % 57")
        self.assertEqual("(31 % 57)", self.fc.visit(node))

        node = ast.parse("31 % (-57)")
        self.assertEqual("(31 % (-57))", self.fc.visit(node))

        node = ast.parse("31 % x")
        self.assertEqual("(31 % x)", self.fc.visit(node))

        node = ast.parse("y % 57")
        self.assertEqual("(y % 57)", self.fc.visit(node))

        node = ast.parse("x % y")
        self.assertEqual("(x % y)", self.fc.visit(node))

        node = ast.parse("x % y % z")
        self.assertEqual("((x % y) % z)", self.fc.visit(node))

        node = ast.parse("x.i % y.i")
        self.assertEqual("(x.i % y.i)", self.fc.visit(node))

        node = ast.parse("x.a_method(a) % y.a_method(b)")
        self.assertEqual("(x.a_method(a) % y.a_method(b))", self.fc.visit(node))

    def test_mat_mult(self):
        node = ast.parse("x @ y")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("x @ y @ z")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("x.i @ y.i")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("x.a_method(a) @ y.a_method(b)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_pow(self):
        node = ast.parse("31 ** 57")
        self.assertEqual("pow(31, 57)", self.fc.visit(node))

        node = ast.parse("31 ** (-57)")
        self.assertEqual("pow(31, (-57))", self.fc.visit(node))

        node = ast.parse("31 ** x")
        self.assertEqual("pow(31, x)", self.fc.visit(node))

        node = ast.parse("y ** 57")
        self.assertEqual("pow(y, 57)", self.fc.visit(node))

        node = ast.parse("x ** y")
        self.assertEqual("pow(x, y)", self.fc.visit(node))

        node = ast.parse("x ** y ** z")
        self.assertEqual("pow(x, pow(y, z))", self.fc.visit(node))

        node = ast.parse("x.i ** y.i")
        self.assertEqual("pow(x.i, y.i)", self.fc.visit(node))

        node = ast.parse("x.a_method(a)** y.a_method(b)")
        self.assertEqual("pow(x.a_method(a), y.a_method(b))", self.fc.visit(node))

    def test_lshift(self):
        node = ast.parse("0b11010101 << 7")
        self.assertEqual("(213 << 7)", self.fc.visit(node))

        node = ast.parse("0b11010101 << (-7)")
        self.assertEqual("(213 << (-7))", self.fc.visit(node))

        node = ast.parse("0b11010101 << x")
        self.assertEqual("(213 << x)", self.fc.visit(node))

        node = ast.parse("y << 7")
        self.assertEqual("(y << 7)", self.fc.visit(node))

        node = ast.parse("x << y")
        self.assertEqual("(x << y)", self.fc.visit(node))

        node = ast.parse("x << y << z")
        self.assertEqual("((x << y) << z)", self.fc.visit(node))

        node = ast.parse("x.i << y.i")
        self.assertEqual("(x.i << y.i)", self.fc.visit(node))

        node = ast.parse("x.a_method(a) << y.a_method(b)")
        self.assertEqual("(x.a_method(a) << y.a_method(b))", self.fc.visit(node))

    def test_rshift(self):
        node = ast.parse("0b11010101 >> 7")
        self.assertEqual("(213 >> 7)", self.fc.visit(node))

        node = ast.parse("0b11010101 >> (-7)")
        self.assertEqual("(213 >> (-7))", self.fc.visit(node))

        node = ast.parse("0b11010101 >> x")
        self.assertEqual("(213 >> x)", self.fc.visit(node))

        node = ast.parse("y >> 7")
        self.assertEqual("(y >> 7)", self.fc.visit(node))

        node = ast.parse("x >> y")
        self.assertEqual("(x >> y)", self.fc.visit(node))

        node = ast.parse("x >> y >> z")
        self.assertEqual("((x >> y) >> z)", self.fc.visit(node))

        node = ast.parse("x.i >> y.i")
        self.assertEqual("(x.i >> y.i)", self.fc.visit(node))

        node = ast.parse("x.a_method(a) >> y.a_method(b)")
        self.assertEqual("(x.a_method(a) >> y.a_method(b))", self.fc.visit(node))

    def test_bit_or(self):
        node = ast.parse("0b11010101 | 0b0111")
        self.assertEqual("(213 | 7)", self.fc.visit(node))

        node = ast.parse("0b11010101 | x")
        self.assertEqual("(213 | x)", self.fc.visit(node))

        node = ast.parse("y | 0b0111")
        self.assertEqual("(y | 7)", self.fc.visit(node))

        node = ast.parse("x | y")
        self.assertEqual("(x | y)", self.fc.visit(node))

        node = ast.parse("x | y | z")
        self.assertEqual("((x | y) | z)", self.fc.visit(node))

        node = ast.parse("x.i | y.i")
        self.assertEqual("(x.i | y.i)", self.fc.visit(node))

        node = ast.parse("x.a_method(a) | y.a_method(b)")
        self.assertEqual("(x.a_method(a) | y.a_method(b))", self.fc.visit(node))

    def test_bit_xor(self):
        node = ast.parse("0b11010101 ^ 0b0111")
        self.assertEqual("(213 ^ 7)", self.fc.visit(node))

        node = ast.parse("0b11010101 ^ x")
        self.assertEqual("(213 ^ x)", self.fc.visit(node))

        node = ast.parse("y ^ 0b0111")
        self.assertEqual("(y ^ 7)", self.fc.visit(node))

        node = ast.parse("x ^ y")
        self.assertEqual("(x ^ y)", self.fc.visit(node))

        node = ast.parse("x ^ y ^ z")
        self.assertEqual("((x ^ y) ^ z)", self.fc.visit(node))

        node = ast.parse("x.i ^ y.i")
        self.assertEqual("(x.i ^ y.i)", self.fc.visit(node))

        node = ast.parse("x.a_method(a) ^ y.a_method(b)")
        self.assertEqual("(x.a_method(a) ^ y.a_method(b))", self.fc.visit(node))

    def test_bit_and(self):
        node = ast.parse("0b11010101 & 0b0111")
        self.assertEqual("(213 & 7)", self.fc.visit(node))

        node = ast.parse("0b11010101 & x")
        self.assertEqual("(213 & x)", self.fc.visit(node))

        node = ast.parse("y & 0b0111")
        self.assertEqual("(y & 7)", self.fc.visit(node))

        node = ast.parse("x & y")
        self.assertEqual("(x & y)", self.fc.visit(node))

        node = ast.parse("x & y & z")
        self.assertEqual("((x & y) & z)", self.fc.visit(node))

        node = ast.parse("x.i & y.i")
        self.assertEqual("(x.i & y.i)", self.fc.visit(node))

        node = ast.parse("x.a_method(a) & y.a_method(b)")
        self.assertEqual("(x.a_method(a) & y.a_method(b))", self.fc.visit(node))

    def test_bool_op(self):
        """
        This test is covered by:
```

```
        test_and
        test_or
        """
        pass

    def test_and(self):
        node = ast.parse("a and b")
        self.assertEqual("(a && b)", self.fc.visit(node))

        node = ast.parse("not a and b")
        self.assertEqual("((!a) && b)", self.fc.visit(node))

        node = ast.parse("a and not b")
        self.assertEqual("(a && (!b))", self.fc.visit(node))

        node = ast.parse("a and b and c")
        self.assertEqual("(a && b && c)", self.fc.visit(node))

        node = ast.parse("a and b and c and d")
        self.assertEqual("(a && b && c && d)", self.fc.visit(node))

        node = ast.parse("a and (b and c)")
        self.assertEqual("(a && (b && c))", self.fc.visit(node))

        node = ast.parse("a and b or c")
        self.assertEqual("((a && b) || c)", self.fc.visit(node))

        node = ast.parse("a and (b or c)")
        self.assertEqual("(a && (b || c))", self.fc.visit(node))

    def test_or(self):
        node = ast.parse("a or b")
        self.assertEqual("(a || b)", self.fc.visit(node))

        node = ast.parse("not a or b")
        self.assertEqual("((!a) || b)", self.fc.visit(node))

        node = ast.parse("a or not b")
        self.assertEqual("(a || (!b))", self.fc.visit(node))

        node = ast.parse("a or b or c")
        self.assertEqual("(a || b || c)", self.fc.visit(node))

        node = ast.parse("a or b or c or d")
        self.assertEqual("(a || b || c || d)", self.fc.visit(node))

        node = ast.parse("a or (b or c)")
        self.assertEqual("(a || (b || c))", self.fc.visit(node))

        node = ast.parse("a or b and c")
        self.assertEqual("(a || (b && c))", self.fc.visit(node))

        node = ast.parse("(a or b) and c")
        self.assertEqual("((a || b) && c)", self.fc.visit(node))

    def test_compare(self):
        """
        Tested by
         test_eq
         test_noteq
         test_lessthan
         test_lessthanequal
         test_greaterthan
         test_greatherthanequal
         test_is
         test_is_not
         test_in
         test_not_in
        """

    def test_eq(self):
        # object comparison not yet supported
        node = ast.parse("1 == 3")
        self.assertEqual("(1 == 3)", self.fc.visit(node))

        node = ast.parse("1 == 3 == 4")
        self.assertEqual("(1 == 3 && 3 == 4)", self.fc.visit(node))

        node = ast.parse("a == 1")
        self.assertEqual("(a == 1)", self.fc.visit(node))

        node = ast.parse("a == b == c")
        self.assertEqual("(a == b && b == c)", self.fc.visit(node))

        node = ast.parse("a == x.some_method(b)")
        self.assertEqual("(a == x.some_method(b))", self.fc.visit(node))

        node = ast.parse("x.some_method(a) == x.some_method(b) " +
                         "== x.some_method(c) == x.some_method(d)")
        self.assertEqual("(x.some_method(a) == x.some_method(b) " +
                         "&& x.some_method(b) == x.some_method(c) " +
                         "&& x.some_method(c) == x.some_method(d))",
                         self.fc.visit(node))

    def test_noteq(self):
        # object comparison not yet supported
        node = ast.parse("1 != 3")
        self.assertEqual("(1 != 3)", self.fc.visit(node))

        node = ast.parse("1 != 3 != 4")
        self.assertEqual("(1 != 3 && 3 != 4)", self.fc.visit(node))

        node = ast.parse("a != 1")
        self.assertEqual("(a != 1)", self.fc.visit(node))

        node = ast.parse("a != b != c")
        self.assertEqual("(a != b && b != c)", self.fc.visit(node))

        node = ast.parse("a != x.some_method(b)")
        self.assertEqual("(a != x.some_method(b))", self.fc.visit(node))

        node = ast.parse("x.some_method(a) != x.some_method(b) " +
                         "!= x.some_method(c) != x.some_method(d)")
        self.assertEqual("(x.some_method(a) != x.some_method(b) " +
                         "&& x.some_method(b) != x.some_method(c) " +
                         "&& x.some_method(c) != x.some_method(d))",
                         self.fc.visit(node))

    def test_lessthan(self):
        node = ast.parse("1 < 3")
        self.assertEqual("(1 < 3)", self.fc.visit(node))

        node = ast.parse("1 < 3 < 4")
        self.assertEqual("(1 < 3 && 3 < 4)", self.fc.visit(node))

        node = ast.parse("a < 1")
        self.assertEqual("(a < 1)", self.fc.visit(node))

        node = ast.parse("a < b < c")
        self.assertEqual("(a < b && b < c)", self.fc.visit(node))

        node = ast.parse("a < x.some_method(b)")
        self.assertEqual("(a < x.some_method(b))", self.fc.visit(node))

        node = ast.parse("x.some_method(a) < x.some_method(b) " +
                         "< x.some_method(c) < x.some_method(d)")
        self.assertEqual("(x.some_method(a) < x.some_method(b) " +
                         "&& x.some_method(b) < x.some_method(c) " +
                         "&& x.some_method(c) < x.some_method(d))",
                         self.fc.visit(node))

    def test_lessthaneq(self):
        node = ast.parse("1 <= 3")
        self.assertEqual("(1 <= 3)", self.fc.visit(node))

        node = ast.parse("1 <= 3 <= 4")
        self.assertEqual("(1 <= 3 && 3 <= 4)", self.fc.visit(node))

        node = ast.parse("a <= 1")
        self.assertEqual("(a <= 1)", self.fc.visit(node))

        node = ast.parse("a <= b <= c")
        self.assertEqual("(a <= b && b <= c)", self.fc.visit(node))

        node = ast.parse("a <= x.some_method(b)")
        self.assertEqual("(a <= x.some_method(b))", self.fc.visit(node))

        node = ast.parse("x.some_method(a) <= x.some_method(b) " +
                         "<= x.some_method(c) <= x.some_method(d)")
        self.assertEqual("(x.some_method(a) <= x.some_method(b) " +
                         "&& x.some_method(b) <= x.some_method(c) " +
                         "&& x.some_method(c) <= x.some_method(d))",
                         self.fc.visit(node))

    def test_greaterthan(self):
        node = ast.parse("1 > 3")
        self.assertEqual("(1 > 3)", self.fc.visit(node))

        node = ast.parse("1 > 3 > 4")
        self.assertEqual("(1 > 3 && 3 > 4)", self.fc.visit(node))

        node = ast.parse("a > 1")
        self.assertEqual("(a > 1)", self.fc.visit(node))

        node = ast.parse("a > b > c")
        self.assertEqual("(a > b && b > c)", self.fc.visit(node))

        node = ast.parse("a > x.some_method(b)")
        self.assertEqual("(a > x.some_method(b))", self.fc.visit(node))

        node = ast.parse("x.some_method(a) > x.some_method(b) " +
                         "> x.some_method(c) > x.some_method(d)")
        self.assertEqual("(x.some_method(a) > x.some_method(b) " +
                         "&& x.some_method(b) > x.some_method(c) " +
                         "&& x.some_method(c) > x.some_method(d))",
                         self.fc.visit(node))

    def test_greaterthaneq(self):
        node = ast.parse("1 >= 3")
        self.assertEqual("(1 >= 3)", self.fc.visit(node))

        node = ast.parse("1 >= 3 >= 4")
        self.assertEqual("(1 >= 3 && 3 >= 4)", self.fc.visit(node))

        node = ast.parse("a >= 1")
        self.assertEqual("(a >= 1)", self.fc.visit(node))

        node = ast.parse("a >= b >= c")
        self.assertEqual("(a >= b && b >= c)", self.fc.visit(node))

        node = ast.parse("a >= x.some_method(b)")
        self.assertEqual("(a >= x.some_method(b))", self.fc.visit(node))

        node = ast.parse("x.some_method(a) >= x.some_method(b) " +
                         ">= x.some_method(c) >= x.some_method(d)")
        self.assertEqual("(x.some_method(a) >= x.some_method(b) " +
                         "&& x.some_method(b) >= x.some_method(c) " +
                         "&& x.some_method(c) >= x.some_method(d))",
                         self.fc.visit(node))

    def test_lessthan(self):
        node = ast.parse("1 < 3")
        self.assertEqual("(1 < 3)", self.fc.visit(node))

        node = ast.parse("1 < 3 < 4")
        self.assertEqual("(1 < 3 && 3 < 4)", self.fc.visit(node))

        node = ast.parse("a < 1")
        self.assertEqual("(a < 1)", self.fc.visit(node))

        node = ast.parse("a < b < c")
        self.assertEqual("(a < b && b < c)", self.fc.visit(node))

        node = ast.parse("a < x.some_method(b)")
        self.assertEqual("(a < x.some_method(b))", self.fc.visit(node))

        node = ast.parse("x.some_method(a) < x.some_method(b) " +
                         "< x.some_method(c) < x.some_method(d)")
        self.assertEqual(
            "(x.some_method(a) < x.some_method(b) " +
            "&& x.some_method(b) < x.some_method(c)" +
            " && x.some_method(c) < x.some_method(d))",
            self.fc.visit(node))

    def test_is(self):
        # object comparison not yet supported
        node = ast.parse("1 is 3")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("1 is 3 is 4")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("a is 1")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("a is b is c")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("a is x.some_method(b)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("x.some_method(a) is x.some_method(b)" +
                         " is x.some_method(c) is x.some_method(d)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_is_not(self):
        # object comparison not yet supported
        node = ast.parse("1 is not 3")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("1 is not 3 is not 4")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("a is not 1")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("a is not b is not c")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("a is not x.some_method(b)")
        self.assertRaises(SyntaxError, self.fc.visit, node)
```

```
        node = ast.parse("x.some_method(a) is not x.some_method(b)" +
                         " is not x.some_method(c) is not x.some_method(d)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_in(self):
        node = ast.parse("1 in [1, 2, 3]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("1 in (1, 2, 3)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("a in (1, 2, 3)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("a in some_list")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("(a in some_list) in (True)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("x.some_method(a) in some_list")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("x.some_method(a) in get_list()")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("x.some_method(a) in x.get_list()")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_not_in(self):
        node = ast.parse("1 in [1, 2, 3]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("1 in (1, 2, 3)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("a in (1, 2, 3)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("a in some_list")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("(a in some_list) in (True)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("x.some_method(a) in some_list")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("x.some_method(a) in get_list()")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("x.some_method(a) in x.get_list()")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_call(self):
        node = ast.parse("some_function()")
        self.assertEquals("some_function()", self.fc.visit(node))

        node = ast.parse("some_function(1)")
        self.assertEquals("some_function(1)", self.fc.visit(node))

        node = ast.parse("some_function(1, 2, 3)")
        self.assertEquals("some_function(1, 2, 3)", self.fc.visit(node))

        node = ast.parse("some_function(a)")
        self.assertEquals("some_function(a)", self.fc.visit(node))

        node = ast.parse("some_function(a, b, c)")
        self.assertEquals("some_function(a, b, c)", self.fc.visit(node))

        node = ast.parse("some_function(*args)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("some_function(a, b, *args)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("some_function(a, b, param1=c)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("some_function(a, b, param1=c, param2=d)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("x.some_method()")
        self.assertEquals("x.some_method()", self.fc.visit(node))

        node = ast.parse("x.some_method(1)")
        self.assertEquals("x.some_method(1)", self.fc.visit(node))

        node = ast.parse("x.some_method(1, 2, 3)")
        self.assertEquals("x.some_method(1, 2, 3)", self.fc.visit(node))

        node = ast.parse("x.some_method(a)")
        self.assertEquals("x.some_method(a)", self.fc.visit(node))

        node = ast.parse("x.some_method(a, b, c)")
        self.assertEquals("x.some_method(a, b, c)", self.fc.visit(node))

        node = ast.parse("x.some_method(*args)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("x.some_method(a, b, *args)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("x.some_method(a, b, param1=c)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("x.some_method(a, b, param1=c, param2=d)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_ifexp(self):
        node = ast.parse("a if test else b")
        self.assertEquals("(test ? a : b)", self.fc.visit(node))

        node = ast.parse("1 if test else 0")
        self.assertEquals("(test ? 1 : 0)", self.fc.visit(node))

        node = ast.parse("1 if a or b else 0")
        self.assertEquals("((a || b) ? 1 : 0)", self.fc.visit(node))

        node = ast.parse("1 if a or b and c else 0")
        self.assertEquals("((a || (b && c)) ? 1 : 0)", self.fc.visit(node))

        node = ast.parse("a if test1 and test2 else b")
        self.assertEquals("((test1 && test2) ? a : b)", self.fc.visit(node))

        node = ast.parse("a_func() if a or b and c else other_func()")
        self.assertEquals("((a || (b && c)) ? a_func() : other_func())", self.fc.visit(node))

        node = ast.parse("a if test_func() else b")
        self.assertEquals("(test_func() ? a : b)", self.fc.visit(node))

    def test_attribute(self):
        node = ast.parse("x.field1")
        self.assertEqual("x.field1", self.fc.visit(node))

        node = ast.parse("x._field2")
        self.assertEqual("x._field2", self.fc.visit(node))

        node = ast.parse("x.some_method")
        self.assertEqual("x.some_method", self.fc.visit(node))

        # x.123 is not valid and should not parse
        self.assertRaises(SyntaxError, ast.parse, "x.123")

    def test_subscript(self):
        """
        This test is covered by:
        test_index
        test_slice
        test_ext_slice
        """

    def test_index(self):
        node = ast.parse("some_list[0]")
        self.assertEqual("some_list[0]", self.fc.visit(node))

        node = ast.parse("some_list[i]")
        self.assertEqual("some_list[i]", self.fc.visit(node))

        node = ast.parse("some_list[i + j]")
        self.assertEqual("some_list[(i + j)]", self.fc.visit(node))

        node = ast.parse("some_list[func()]")
        self.assertEqual("some_list[func()]", self.fc.visit(node))

        node = ast.parse("some_list[func1() + func2()]")
        self.assertEqual("some_list[(func1() + func2())]", self.fc.visit(node))

        node = ast.parse("some_list[func(i) + func(j)]")
        self.assertEqual("some_list[(func(i) + func(j))]", self.fc.visit(node))

        node = ast.parse("some_list[x.some_method()]")
        self.assertEqual("some_list[x.some_method()]", self.fc.visit(node))

        node = ast.parse("some_list[x.some_method(i)]")
        self.assertEqual("some_list[x.some_method(i)]", self.fc.visit(node))

        node = ast.parse("get_list()[x.some_method(i)]")
        self.assertEqual("get_list()[x.some_method(i)]", self.fc.visit(node))

        node = ast.parse("x.get_list()[x.some_method(i)]")
        self.assertEqual("x.get_list()[x.some_method(i)]", self.fc.visit(node))

    def test_slice(self):
        node = ast.parse("some_list[0:5]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("some_list[i:i+5]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("some_list[i + j : i + j + 5]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("some_list[func1() : func2()]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("some_list[func(i) : func(j)]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("some_list[x.some_method(i) : x.some_method(j)]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("get_list()[1:5]")
        self.assertRaises(SyntaxError, self.fc.visit, node)
        node = ast.parse("x.get_list()[x.some_method(i): x.some_method(j)]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_ext_slice(self):
        node = ast.parse("some_list[0:5, 3]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("some_list[i:i+5, i+7]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("some_list[i + j : i + j + 5, i + j + 8]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("some_list[func1() : func2(), func2() + 2]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("some_list[func(i) : func(j), func(j) + func(i)]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("some_list[x.some_method(i) : x.some_method(j), x.some_method(k)]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("get_list()[1:5, 7]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("x.get_list()[x.some_method(i): x.some_method(j), s.some_method(k)]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_list_comp(self):
        node = ast.parse("[i * 2 for i in range(1, 1000)]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("[item for item in a_list]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("[(item1, item2) for item in some_tuples]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("[func(item) for item in a_list]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("[item for item in get_list()]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("[item for item in x.get_list()]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_set_comp(self):
        node = ast.parse("{i * 2 for i in range(1, 1000)}")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("{item for item in a_list}")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("{(item1, item2) for item in some_tuples}")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("{func(item) for item in a_list}")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("{item for item in get_list()}")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("{item for item in x.get_list()}")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_gen_exp(self):
        node = ast.parse("(i * 2 for i in range(1, 1000))")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("(item for item in a_list)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("((item1, item2) for item in some_tuples)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("(func(item) for item in a_list)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("(item for item in get_list())")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("(item for item in x.get_list())")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_dict_comp(self):
        node = ast.parse("{key: value for key, value in some_tuples}")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("{key: func(key) for key in a_list}")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("{x: 3*x for x in a_list}")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("{key: value for key, value in get_pairs()}")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("{key: value for key, value in x.get_pairs()}")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_assign(self):
        class TestClass:
            pass

        # initial use of a1
        node = ast.parse("a1 = 1234")
        self.assertEquals("auto a1 = 1234", self.fc.visit(node))

        # reassignment of a1
        node = ast.parse("a1 = func()")
        self.assertEquals("a1 = func()", self.fc.visit(node))

        #initial use of a2
        node = ast.parse("a2 = func(x, y)")
        self.assertEquals("auto&& a2 = func(x, y)", self.fc.visit(node))

        node = ast.parse("a2.some_field = 1234")
        self.assertEquals("a2.some_field = 1234", self.fc.visit(node))

        # reassignment of a2
        node = ast.parse("a2 = b.some_method()")
        self.assertEquals("a2 = b.some_method()", self.fc.visit(node))

        node = ast.parse("a2 = b + c")
        self.assertEquals("a2 = (b + c)", self.fc.visit(node))

        node = ast.parse("a2 = b + func(c)")
        self.assertEquals("a2 = (b + func(c))", self.fc.visit(node))

        node = ast.parse("a2 = c.some_method() + b")
        self.assertEquals("a2 = (c.some_method() + b)", self.fc.visit(node))

        node = ast.parse("a2 = b + c.some_method(x)")
        self.assertEquals("a2 = (b + c.some_method(x))", self.fc.visit(node))
        #initial declaration of a3
        # assign a non-primitive
        self.fc.func_repr.args.append("var")
        self.fc.func_repr.arg_types.append(TestClass)

        node = ast.parse("a3 = var")
        self.assertEquals("auto&& a3 = var", self.fc.visit(node))

        # initial declaration of a4
        # assign a list
        self.fc.func_repr.args.append("var_list")
        self.fc.func_repr.arg_types.append("List<TestClass>")

        node = ast.parse("a4 = var_list")
        self.assertEquals("auto&& a4 = var_list", self.fc.visit(node))

        # initial declaration of a5
        # assign a non primitive list item
        node = ast.parse("a5 = var_list[3]")
        self.assertEquals("auto&& a5 = var_list[3]", self.fc.visit(node))

        # initial declaration of a6
        # assign a primitive list item
        self.fc.func_repr.args.append("var_list2")
        self.fc.func_repr.arg_types.append("List<int>")

        node = ast.parse("a6 = var_list2[3]")
        self.assertEquals("auto a6 = var_list2[3]", self.fc.visit(node))

        # not an initial declaration
        node = ast.parse("a_list[5] = func(x)")
        self.assertEquals("a_list[5] = func(x)", self.fc.visit(node))

        # this is only valid if get_list() returns a reference!
        node = ast.parse("get_list()[0] = func(y)")
        self.assertEquals("get_list()[0] = func(y)", self.fc.visit(node))

        # unpacking not allowed
        node = ast.parse("a, b = get_tuple()")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("a, b = (1, 2)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("a, b = get_tuple(x)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        # multiple assignment not allowed
        node = ast.parse("a = b = c = 1")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("a = b = c = func(d)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_aug_assign(self):
        # a1 already declared
        self.fc.local_vars["a1"] = None

        node = ast.parse("a1 += 1234")
        self.assertEquals("a1 = a1 + 1234", self.fc.visit(node))

        # reassignment of a1
        node = ast.parse("a1 /= func()")
        self.assertEquals("a1 = a1 / func()", self.fc.visit(node))

        #initial use of a2
        node = ast.parse("a2 &= func(x, y)")
        self.assertEquals("a2 = a2 & func(x, y)", self.fc.visit(node))

        node = ast.parse("a2.some_field %= 1234")
        self.assertEquals("a2.some_field = a2.some_field % 1234", self.fc.visit(node))

        # reassignment of a2
        node = ast.parse("a2 *= b.some_method()")
        self.assertEquals("a2 = a2 * b.some_method()", self.fc.visit(node))

        node = ast.parse("a2 -= b + c")
        self.assertEquals("a2 = a2 - (b + c)", self.fc.visit(node))

        node = ast.parse("a2 += c.some_method() + b")
        self.assertEquals("a2 = a2 + (c.some_method() + b)", self.fc.visit(node))

        node = ast.parse("a2 /= b + c.some_method(x)")
        self.assertEquals("a2 = a2 / (b + c.some_method(x))", self.fc.visit(node))

        # not an initial declaration
        node = ast.parse("a_list[5] >>= func(x)")
        self.assertEquals("a_list[5] = a_list[5] >> func(x)", self.fc.visit(node))

        # this is only valid if get_list() returns a reference!
        node = ast.parse("get_list()[0] <<= func(y)")
        self.assertEquals("get_list()[0] = get_list()[0] << func(y)", self.fc.visit(node))

    def test_raise(self):
        # no exceptions allowed
        node = ast.parse("raise SyntaxError(\"message\")")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_assert(self):
        node = ast.parse("assert True")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("assert a + b > 0")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("assert a and b")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("assert check_condition()")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("assert x.check_condition(b)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_delete(self):
        # no del allowed
        node = ast.parse("del a_list[3]")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("del a")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_pass(self):
        node = ast.parse("pass")
        self.assertEquals("", self.fc.visit(node))

    def test_import(self):
        node = ast.parse("import a_module")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("import a_module.a_class")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("import a_module as m")
        self.assertRaises(SyntaxError, self.fc.visit, node)
        node = ast.parse("import a_module.a_class as c")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_import_from(self):
        node = ast.parse("from a_module import a_class")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("from a_module import a_class as c")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("from . import a_module")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("from . import a_module as m")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_if(self):
        input_code = """
if a:
    b = func()
    c.some_method(d)
    if a and b or c:
        b.some_method(e)
    elif d < c.some_method2():
        c.some_method(e)
    else:
        a = func2(a)
else:
    a = func3(a)
"""
        output_code = """\
if (a) {
    auto&& b = func();
    c.some_method(d);
    if (((a && b) || c)) {
        b.some_method(e);
    } else {
        if ((d < c.some_method2())) {
            c.some_method(e);
        } else {
            a = func2(a);
        }
    }
} else {
    a = func3(a);
}\
"""
        self.fc.local_vars["a"] = None
        node = ast.parse(input_code)
        self.assertEquals(output_code, self.fc.visit(node))

    def test_for(self):
        input_code = """
for i in range(5, 1000, 5):
    b = func(i)
    for a in a_list:
        a.some_method(b)
        a.some_other_method(b)
        for j in range(len(another_list)):
            another_list[j].some_method()
"""
        output_code = """\
auto __iterator_1 = RangeIterator(5, 1000, 5);
while(__iterator_1.has_next()) {
    int i = __iterator_1.next();
    auto&& b = func(i);
    auto __iterator_2 = ListIterator<SomeClass>(a_list);
    while (__iterator_2.has_next()) {
        SomeClass &a = __iterator_2.next();
        a.some_method(b);
        a.some_other_method(b);
        auto __iterator_3 = RangeIterator(0, len(another_list), 1);
        while(__iterator_3.has_next()) {
            int j = __iterator_3.next();
            another_list[j].some_method();
        }
    }
}\
"""
        self.fc.local_vars["a_list"] = "List<SomeClass>" # a_list is a list
        node = ast.parse(input_code)
        self.assertEquals(output_code, self.fc.visit(node))

    def test_while(self):
        input_code = """\
while True:
    a = func1()
    b = func2()
    while a and b and a < b:
        a += 1
        b -= 1
    if a > b:
        break
    elif a == b:
        continue
"""
        output_code = """\
while (true) {
    auto&& a = func1();
    auto&& b = func2();
    while ((a && b && (a < b))) {
        a = a + 1;
        b = b - 1;
    }
    if ((a > b)) {
        break;
    } else {
        if ((a == b)) {
            continue;
        }
    }
}\
"""
        node = ast.parse(input_code)
        self.assertEquals(output_code, self.fc.visit(node))

    def test_break(self):
        node = ast.parse("break")
        self.assertEquals("break", self.fc.visit(node))

    def test_continue(self):
        node = ast.parse("continue")
        self.assertEquals("continue", self.fc.visit(node))

    def test_try(self):
        input_code = """
try:
    a = b
    a.some_method(x)
    if a.check_condition():
        a.some_other_method(y)
except Error1:
    a.reset()
except Error2:
    pass
```

```
"""
        node = ast.parse(input_code)
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_with(self):
        input_code = """
with some_func(x) as val:
    val.do_something()
    if val.check_something():
        val.do_something_else()
"""
        node = ast.parse(input_code)
        self.assertRaises(SyntaxError, self.fc.visit, node)

        input_code = """
with v as val:
    val.do_something()
    if val.check_something():
        val.do_something_else()
"""
        node = ast.parse(input_code)
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_func_def(self):
        func_repr = FunctionRepresentation("get_length", ["x", "y", "z"],
                                           [float, float, float], float, None)
        self.fc = FunctionConverter(func_repr)

        input_code = """
def get_length(x, y, z):
    x_sq = x * x
    y_sq = y * y
    z_sq = z * z
    return sqrt(x_sq + y_sq + z_sq)
"""
        output_code = """\
__device__ float get_length(float x, float y, float z) {
    auto x_sq = (x * x);
    auto y_sq = (y * y);
    auto z_sq = (z * z);
    return sqrt(((x_sq + y_sq) + z_sq));
}
"""
        node = ast.parse(input_code)
        self.assertEquals(output_code, self.fc.visit(node))

        func_repr = FunctionRepresentation("get_remainder", ["numerator", "denominator"],
                                           [int, int], int, None)
        self.fc = FunctionConverter(func_repr)

        input_code = """
def get_remainder(numerator, denominator):
    temp = numerator
    while temp - denominator > 0:
        temp -= denominator
    return temp
"""
        output_code = """\
__device__ int get_remainder(int numerator, int denominator) {
    auto temp = numerator;
    while (((temp - denominator) > 0)) {
        temp = temp - denominator;
    }
    return temp;
}
"""
        node = ast.parse(input_code)
        self.assertEquals(output_code, self.fc.visit(node))
        class Vector3D:
            pass

        func_repr = FunctionRepresentation("get_length", ["v"], [Vector3D], float, None)
        self.fc = FunctionConverter(func_repr)

        input_code = """
def get_length(v):
    x_sq = v.x * v.x
    y_sq = v.y * v.y
    z_sq = v.z * v.z
    return sqrt(x_sq + y_sq + z_sq)
"""

        output_code = """\
__device__ float get_length(Vector3D&& v) {
    auto x_sq = (v.x * v.x);
    auto y_sq = (v.y * v.y);
    auto z_sq = (v.z * v.z);
    return sqrt(((x_sq + y_sq) + z_sq));
}
"""
        node = ast.parse(input_code)
        self.assertEquals(output_code, self.fc.visit(node))

        func_repr = FunctionRepresentation("get_length", ["v"], [Vector3D], float, None)
        self.fc = FunctionConverter(func_repr, refs=[True])

        input_code = """
def get_length(v):
    x_sq = v.x * v.x
    y_sq = v.y * v.y
    z_sq = v.z * v.z
    return sqrt(x_sq + y_sq + z_sq)
"""

        output_code = """\
__device__ float get_length(Vector3D& v) {
    auto x_sq = (v.x * v.x);
    auto y_sq = (v.y * v.y);
    auto z_sq = (v.z * v.z);
    return sqrt(((x_sq + y_sq) + z_sq));
}
"""
        node = ast.parse(input_code)
        self.assertEquals(output_code, self.fc.visit(node))

        func_repr = FunctionRepresentation("get_length", ["v"], [Vector3D], float, None)
        self.fc = FunctionConverter(func_repr)

        input_code = """
def get_length(v):
    def calc_length(x, y, z):
        x_sq = x * x
        y_sq = y * y
        z_sq = z * z
        return sqrt(x_sq + y_sq + z_sq)
    return calc_length(v.x, v.y, v.z)
"""
        node = ast.parse(input_code)
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_lambda(self):
        node = ast.parse("lambda x: x + 1")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("lambda x: func(x)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("lambda x, y: x + y")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_return(self):
        node = ast.parse("return")
        self.assertEquals("return", self.fc.visit(node))

        node = ast.parse("return a")
        self.assertEquals("return a", self.fc.visit(node))

        node = ast.parse("return a + b")
        self.assertEquals("return (a + b)", self.fc.visit(node))

        node = ast.parse("return 1")
        self.assertEquals("return 1", self.fc.visit(node))

        node = ast.parse("return True")
        self.assertEquals("return true", self.fc.visit(node))

        node = ast.parse("return func(x)")
        self.assertEquals("return func(x)", self.fc.visit(node))

        node = ast.parse("return x.some_method(y)")
        self.assertEquals("return x.some_method(y)", self.fc.visit(node))

    def test_yield(self):
        node = ast.parse("yield a")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("yield a + b")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("yield 1")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("yield True")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("yield func(x)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("yield x.some_method(y)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_yield_from(self):
        node = ast.parse("yield from a_generator")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("yield from get_generator(x)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

        node = ast.parse("yield from x.get_generator(y)")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_global(self):
        node = ast.parse("global var")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_nonlocal(self):
        node = ast.parse("nonlocal var")
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_ClassDef(self):
        input_code = """
class TestClass:
    def __init__(a, b):
        self.a = a
        self.b = b

    def some_method(arg1, arg2):
        return arg1 + arg2
"""
        node = ast.parse(input_code)
        self.assertRaises(SyntaxError, self.fc.visit, node)

    def test_async_func_def(self):
        func_repr = FunctionRepresentation("get_length", ["x", "y", "z"],
                                           [float, float, float], float, None)
        self.fc = FunctionConverter(func_repr)

        input_code = """
async def get_length(x, y, z):
    x_sq = x * x
    y_sq = y * y
    z_sq = z * z
    return sqrt(x_sq + y_sq + z_sq)
"""
        node = ast.parse(input_code)
        self.assertRaises(SyntaxError, self.fc.visit, node)

        func_repr = FunctionRepresentation("get_remainder", ["numerator", "denominator"],
                                           [int, int], int, None)
        self.fc = FunctionConverter(func_repr)

        input_code = """
async def get_remainder(numerator, denominator):
    temp = numerator
    while temp - denominator > 0:
        temp -= denominator
    return temp
"""

        node = ast.parse(input_code)
        self.assertRaises(SyntaxError, self.fc.visit, node)

class TestMethodConverter(TestCase):
    class Vector1D:
        def __init__(self, v):
            self.v = v

    class Vector3D:
        def __init__(self, x, y, z):
            self.x = x
            self.y = y
            self.z = z

    def setUp(self):
        method_repr = MethodRepresentation(TestMethodConverter.Vector3D,
                                           "get_length", ["self"],
                                           [TestMethodConverter.Vector3D], float, None)
        v = TestMethodConverter.Vector3D(3,
                                         TestMethodConverter.Vector1D(4),
                                         TestMethodConverter.Vector1D(5))
        extracted = ExtractedClasses()
        class_repr = extracted.extract(v)
        self.mc = MethodConverter(class_repr, method_repr)

    def test_name(self):
        node = ast.parse("x")
        self.assertEqual("x", self.mc.visit(node))

        node = ast.parse("val")
        self.assertEqual("val", self.mc.visit(node))
        node = ast.parse("_val")
        self.assertEqual("_val", self.mc.visit(node))

        node = ast.parse("a_val")
        self.assertEqual("a_val", self.mc.visit(node))

        node = ast.parse("Val")
        self.assertEqual("Val", self.mc.visit(node))

        node = ast.parse("self")
        self.assertEqual("(*this)", self.mc.visit(node))

        node = ast.parse("selfxyz")
        self.assertEqual("selfxyz", self.mc.visit(node))

    def test_attribute(self):
        node = ast.parse("self.field1")
        self.assertEqual("(*this).field1", self.mc.visit(node))

        node = ast.parse("self._field2")
        self.assertEqual("(*this)._field2", self.mc.visit(node))

        node = ast.parse("self.some_method")
        self.assertEqual("(*this).some_method", self.mc.visit(node))

    def test_assign(self):
        class TestClass:
            pass

        # initial use of a1
        node = ast.parse("a1 = 1234")
        self.assertEquals("auto a1 = 1234", self.mc.visit(node))

        # reassignment of a1
        node = ast.parse("a1 = func()")
        self.assertEquals("a1 = func()", self.mc.visit(node))

        #initial use of a2
        node = ast.parse("a2 = func(x, y)")
        self.assertEquals("auto&& a2 = func(x, y)", self.mc.visit(node))

        node = ast.parse("a2.some_field = 1234")
        self.assertEquals("a2.some_field = 1234", self.mc.visit(node))

        # reassignment of a2
        node = ast.parse("a2 = b.some_method()")
        self.assertEquals("a2 = b.some_method()", self.mc.visit(node))

        node = ast.parse("a2 = b + c")
        self.assertEquals("a2 = (b + c)", self.mc.visit(node))

        node = ast.parse("a2 = b + func(c)")
        self.assertEquals("a2 = (b + func(c))", self.mc.visit(node))

        node = ast.parse("a2 = c.some_method() + b")
        self.assertEquals("a2 = (c.some_method() + b)", self.mc.visit(node))

        node = ast.parse("a2 = b + c.some_method(x)")
        self.assertEquals("a2 = (b + c.some_method(x))", self.mc.visit(node))

        #initial declaration of a3
        # assign a non-primitive
        self.mc.func_repr.args.append("var")
        self.mc.func_repr.arg_types.append(TestClass)

        node = ast.parse("a3 = var")
        self.assertEquals("auto&& a3 = var", self.mc.visit(node))
        # initial declaration of a4
        # assign a list
        self.mc.func_repr.args.append("var_list")
        self.mc.func_repr.arg_types.append("List<TestClass>")

        node = ast.parse("a4 = var_list")
        self.assertEquals("auto&& a4 = var_list", self.mc.visit(node))

        # initial declaration of a5
        # assign a non primitive list item
        node = ast.parse("a5 = var_list[3]")
        self.assertEquals("auto&& a5 = var_list[3]", self.mc.visit(node))

        # initial declaration of a6
        # assign a primitive list item
        self.mc.func_repr.args.append("var_list2")
        self.mc.func_repr.arg_types.append("List<int>")

        node = ast.parse("a6 = var_list2[3]")
        self.assertEquals("auto a6 = var_list2[3]", self.mc.visit(node))

        node = ast.parse("a7 = self.x")
        self.assertEquals("auto a7 = (*this).x", self.mc.visit(node))

        node = ast.parse("a8 = self.y")
        self.assertEquals("auto&& a8 = (*this).y", self.mc.visit(node))

        node = ast.parse("a9 = self")
        self.assertEquals("auto&& a9 = (*this)", self.mc.visit(node))

        # not an initial declaration
        node = ast.parse("a_list[5] = func(x)")
        self.assertEquals("a_list[5] = func(x)", self.mc.visit(node))

        # this is only valid if get_list() returns a reference!
        node = ast.parse("get_list()[0] = func(y)")
        self.assertEquals("get_list()[0] = func(y)", self.mc.visit(node))

        # unpacking not allowed
        node = ast.parse("a, b = get_tuple()")
        self.assertRaises(SyntaxError, self.mc.visit, node)

        node = ast.parse("a, b = (1, 2)")
        self.assertRaises(SyntaxError, self.mc.visit, node)

        node = ast.parse("a, b = get_tuple(x)")
        self.assertRaises(SyntaxError, self.mc.visit, node)

        # multiple assignment not allowed
        node = ast.parse("a = b = c = 1")
        self.assertRaises(SyntaxError, self.mc.visit, node)

        node = ast.parse("a = b = c = func(d)")
        self.assertRaises(SyntaxError, self.mc.visit, node)

    def test_func_def(self):
        input_code = """
def get_length(self):
    x_sq = self.x * self.x
    y_sq = self.y.val * self.y.val
    z_sq = self.z.val * self.z.val
    return sqrt(x_sq + y_sq + z_sq)
"""
        output_code = """\
__device__ float Vector3D::get_length() {
    auto x_sq = ((*this).x * (*this).x);
    auto y_sq = ((*this).y.val * (*this).y.val);
    auto z_sq = ((*this).z.val * (*this).z.val);
    return sqrt(((x_sq + y_sq) + z_sq));
}
"""
        node = ast.parse(input_code)
        self.assertEquals(output_code, self.mc.visit(node))


        input_code = """
def get_length(self):
    def calc_length(x, y, z):
        x_sq = x * x
        y_sq = y * y
        z_sq = z * z
        return sqrt(x_sq + y_sq + z_sq)
    return calc_length(self.x, self.y.val, self.z.val)
"""
        node = ast.parse(input_code)
        self.assertRaises(SyntaxError, self.mc.visit, node)
```

Appendix B

SERIALIZATION TEST

This appendix contains the implementation of the serialization test used to validate the serialization functionality of GPUMap, as discussed in section 5.2.

```python
from serialization import ListSerializer
from data_model import ExtractedClasses
from class_def import ClassDefGenerator

import random
from copy import deepcopy

from pycuda import autoinit
import pycuda.driver as cuda
from pycuda.compiler import SourceModule

class TestClassA:
    def __init__(self, a, b, c, o):
        self.a = a
        self.b = b
        self.c = c
        self.d = 0
        self.o = o

class TestClassB:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
        self.q = TestClassC(1)

class TestClassC:
    def __init__(self, i):
        self.i = i

class TestSerialization:
    def test_serialize_deserialize_object(self):
        class_reprs = ExtractedClasses()

        items = [TestClassA(random.randint(0, 100),
                            random.randint(100, 200),
                            random.randint(200, 300),
                            TestClassB(random.randint(0, 100),
                                       random.randint(100, 200),
                                       random.randint(200, 300)))
                 for _ in range(1000)]

        duplicate_items = deepcopy(items)

        candidate_item = items[0]
        class_repr = class_reprs.extract(candidate_item)
        serializer = ListSerializer(class_repr, items)

        bytes = serializer.to_bytes()

        new_items = serializer.from_bytes(bytes) # unpack back into same list
        assert new_items is items
        assert len(duplicate_items) == len(new_items)
        for i1, i2 in zip(duplicate_items, new_items):
            assert i1.a == i2.a
            assert i1.b == i2.b
            assert i1.c == i2.c
            assert i1.d == i2.d
            assert i1.o.x == i2.o.x
            assert i1.o.y == i2.o.y
            assert i1.o.z == i2.o.z
            assert i1.o.q.i == i2.o.q.i

        print("passed object to_bytes test")

        serializer2 = ListSerializer(class_repr, length=len(items))

        output_list = serializer2.create_output_list(bytes, candidate_item)
        assert output_list is not items

        assert len(duplicate_items) == len(output_list)
        for i1, i2 in zip(duplicate_items, output_list):
            assert i1.a == i2.a
            assert i1.b == i2.b
            assert i1.c == i2.c
            assert i1.d == i2.d
            assert i1.o.x == i2.o.x
            assert i1.o.y == i2.o.y
            assert i1.o.z == i2.o.z
            assert i1.o.q.i == i2.o.q.i

        print("passed object create_output_list test")

    def test_serialize_deserialize_primitives(self):
        class_reprs = ExtractedClasses()

        items = [random.randint(0, 1024) for _ in range(1000)]

        duplicate_items = deepcopy(items)

        candidate_item = items[0]

        class_repr = class_reprs.extract(candidate_item)
        serializer = ListSerializer(class_repr, items)

        bytes = serializer.to_bytes()

        new_items = serializer.from_bytes(bytes)

        assert len(duplicate_items) == len(new_items)
        for i1, i2 in zip(duplicate_items, new_items):
            assert i1 == i2

        print("passed primitive to_bytes test")

        serializer2 = ListSerializer(class_repr, length=len(items))
        output_list = serializer2.create_output_list(bytes, candidate_item)
        assert output_list is not items

        assert len(duplicate_items) == len(output_list)
        for i1, i2 in zip(duplicate_items, output_list):
            assert i1 == i2

        print("passed primitive create_output_list test")

    def test_gpu(self):
        class_reprs = ExtractedClasses()
        items = [TestClassA(random.randint(0, 100),
                            random.randint(100, 200),
                            random.randint(200, 300),
                            TestClassB(random.randint(0, 100),
                                       random.randint(100, 200),
                                       random.randint(200, 300)))
                 for _ in range(1000)]

        candidate_item = items[0]
        class_repr = class_reprs.extract(candidate_item)

        serializer = ListSerializer(class_repr, items)
        bytes = serializer.to_bytes()

        cls_def_gen = ClassDefGenerator()
        from builtin import builtin

        kernel = builtin + cls_def_gen.all_cpp_class_defs(class_reprs) + """
extern "C" {
__global__ void test(List<TestClassA> *things) {
  List<TestClassA> &things_ref = *things;
  int val = things_ref[threadIdx.x].a;
  int val2 = things_ref[threadIdx.x].b;
  int val3 = things_ref[threadIdx.x].o.x;
  things_ref[threadIdx.x].o.x = val * val2 * val3;
}
}
"""
        mod = SourceModule(kernel, options=["--std=c++11"], no_extern_c=True)
        stuff = mod.get_function("test")

        list_ptr = cuda.to_device(bytes)
        new_bytes = bytearray(len(bytes))
        stuff(list_ptr, block=(len(items), 1, 1))
        cuda.memcpy_dtoh(new_bytes, list_ptr)

        deserializer = ListSerializer(class_repr, length=len(items))
        new_list = deserializer.create_output_list(new_bytes, candidate_item)

        for item, new_item in zip(items, new_list):
            assert item.a == new_item.a
            assert item.b == new_item.b
            assert item.c == new_item.c
            assert item.o.x * item.a * item.b == new_item.o.x
            assert item.o.y == new_item.o.y
            assert item.o.z == new_item.o.z

        print("GPUTest done!")


if __name__ == "__main__":
    t = TestSerialization()
    t.test_serialize_deserialize_object()
    t.test_serialize_deserialize_primitives()
    t.test_gpu()
```

Appendix C

N-BODY TEST

This appendix contains the implementation of the n-body test used to validate the functionality of GPUMap, as discussed in section 5.3, and also used to benchmark GPUMap, as discussed in section 6.1.

```python
from mapper import gpumap
from util import get_time, Results

from random import uniform
import pickle
import math


class Body:
    def __init__(self, x, y, z, vx, vy, vz, mass):
        self.pos = Vector3(x, y, z)
        self.vel = Vector3(vx, vy, vz)
        self.mass = mass


class Vector3:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def add(self, other):
        return Vector3(
            self.x + other.x,
            self.y + other.y,
            self.z + other.z
        )

    def sub(self, other):
        return Vector3(
            self.x - other.x,
            self.y - other.y,
            self.z - other.z
        )

    def scale(self, scalar):
        return Vector3(
            scalar * self.x,
            scalar * self.y,
            scalar * self.z
        )

    def length(self):
        return math.sqrt(
            math.pow(self.x, 2) +
            math.pow(self.y, 2) +
            math.pow(self.z, 2)
        )
```

```
class BodyGenerator:
    def __init__(self, num_bodies):
        self.bodies = None
        self.num_bodies = num_bodies
        self.pickled_bodies = None

    def generate_bodies(self):
        rand_pos = lambda: uniform(-1000, 1000)
        rand_vel = lambda: uniform(-10, 10)
        rand_mass = lambda: uniform(-20, 20)
        self.bodies = [
            Body(rand_pos(), rand_pos(), rand_pos(),
                 rand_vel(), rand_vel(), rand_vel(),
                 rand_mass()) for _ in range(self.num_bodies)
            ]
        self.pickled_bodies = pickle.dumps(self.bodies)

    def get_copy(self):
        return pickle.loads(self.pickled_bodies)


class Simulation:
    def __init__(self, bodies, num_steps):
        self.bodies = bodies
        self.indices = list(range(len(bodies)))
        self.dt = 0.01
        self.padding = 0.0000001
        self.num_steps = num_steps

    def advance(self, dt, padding):
        pass

    def run(self):
        for _ in range(self.num_steps):
            self.advance(self.dt, self.padding)


class GPU_Simulation(Simulation):
    def advance(self, dt, padding):
        bodies = self.bodies

        def calc_vel(i):
            b1 = bodies[i]
            for b2 in bodies:
                d_pos = b1.pos.sub(b2.pos)
                distance = d_pos.length() + padding
                mag = dt / math.pow(distance, 3)
                b1.vel = b1.vel.sub(d_pos.scale(b2.mass).scale(mag))

        gpumap(calc_vel, self.indices)

        def update(body):
            body.pos = body.pos.add(body.vel.scale(dt))

        gpumap(update, bodies)


class CPU_Simulation(Simulation):
    def advance(self, dt, padding):
        bodies = self.bodies

        def calc_vel(i):
            b1 = bodies[i]
            for b2 in bodies:
                d_pos = b1.pos.sub(b2.pos)
                distance = d_pos.length() + padding
                mag = dt / math.pow(distance, 3)
                b1.vel = b1.vel.sub(d_pos.scale(b2.mass).scale(mag))
        list(map(calc_vel, self.indices))

        def update(body):
            body.pos = body.pos.add(body.vel.scale(dt))

        list(map(update, bodies))


def test():
    num_steps = 10
    with open("nbodies_out.csv", "w") as f:
        print("num_bodies,gpu_time,cpu_time", file=f)
        num_bodies = 2
        while num_bodies <= 8192:
            print("num bodies", num_bodies)
            body_gen = BodyGenerator(num_bodies)
            body_gen.generate_bodies()

            gpu_bodies = body_gen.get_copy()
            gpu_sim = GPU_Simulation(gpu_bodies, num_steps)
            gpu_time = get_time(gpu_sim.run)

            cpu_bodies = body_gen.get_copy()
            cpu_sim = CPU_Simulation(cpu_bodies, num_steps)
            cpu_time = get_time(cpu_sim.run)

            for gpu_body, cpu_body in zip(gpu_bodies, cpu_bodies):
                assert abs(gpu_body.pos.x - cpu_body.pos.x) < 0.001
                assert abs(gpu_body.pos.y - cpu_body.pos.y) < 0.001
                assert abs(gpu_body.pos.z - cpu_body.pos.z) < 0.001
                assert abs(gpu_body.vel.x - cpu_body.vel.x) < 0.001
                assert abs(gpu_body.vel.y - cpu_body.vel.y) < 0.001
                assert abs(gpu_body.vel.z - cpu_body.vel.z) < 0.001
                assert abs(gpu_body.vel.z - cpu_body.vel.z) < 0.001
                assert abs(gpu_body.mass - cpu_body.mass) < 0.001


            print("{},{},{}".format(num_bodies, gpu_time, cpu_time), file=f)
            f.flush()
            Results.output_results("nbody", "num_bodies{}.csv".format(num_bodies))
            Results.clear_results()
            num_bodies *= 2


def warmup():
    warmup_bodies = 256
    num_steps = 10

    body_gen = BodyGenerator(warmup_bodies)
    body_gen.generate_bodies()

    # warmup
    gpu_bodies = body_gen.get_copy()
    gpu_sim = GPU_Simulation(gpu_bodies, num_steps)
    gpu_sim.run()

    cpu_bodies = body_gen.get_copy()
    cpu_sim = CPU_Simulation(cpu_bodies, num_steps)
    cpu_sim.run()
    Results.clear_results()

if __name__ == "__main__":
    warmup()
    test()
```

Appendix D

BUBBLE SORT TEST

This appendix contains the implementation of the bubble sort test used to validate the functionality of GPUMap, as discussed in section 5.4, and also used to benchmark GPUMap, as discussed in section 6.2.

```python
from mapper import gpumap
from random import randint
from pickle import dumps, loads
from util import get_time, Results

class TestSort:
    def prepare(self, size):
        self.lists = [[randint(0, 1000000) for _ in range(size)] for _ in range(5000)]
        pickle_str = dumps(self.lists)
        self.lists2 = loads(pickle_str)
        self.sorted = [sorted(l) for l in self.lists]

    def warmup(self):
        self.prepare(256)
        gpumap(bubblesort, self.lists)
        list(map(bubblesort, self.lists2))
        Results.clear_results()

    def run(self):
        with open("bubble_out.csv", "a") as f:
            print("list_size,num_lists,gpu,cpu",file=f)
            size = 2
            while size <= 8192:
                self.prepare(size)
                gpu = get_time(gpumap, bubblesort, self.lists)
                cpu = get_time(list, map(bubblesort, self.lists2))

                for l1, l2, l3 in zip(self.lists, self.lists2, self.sorted):
                    for i1, i2, i3 in zip(l1, l2, l3):
                        assert i1 == i2 == i3

                print("{},{},{},{}".format(size, len(self.lists), gpu, cpu), file=f)
                f.flush()

                Results.output_results("bubblesort", "results-%d.csv" % size)
                Results.clear_results()
                size *= 2

def bubblesort(lst):
    for i in range(len(lst)):
        for j in range(i+1, len(lst)):
            if lst[j] < lst[i]:
                temp = lst[j]
                lst[j] = lst[i]
                lst[i] = temp

if __name__ == "__main__":
    t = TestSort()
    t.warmup()
    t.run()
```

Appendix E

SHELL SORT TEST

This appendix contains the implementation of the shell sort test used to validate the functionality of GPURDD, as discussed in section 5.5, and also used to benchmark GPURDD, as discussed in section 6.3.

```
from time import perf_counter
from random import randint
from pyspark.sql import SparkSession
from gpurdd import GPURDD
from util import get_time

def shellsort(items):
    gap = len(items) // 2
    while gap > 0:
        for i in range(gap, len(items)):
            val = items[i]
            j = i
            while j >= gap and items[j - gap] > val:
                items[j] = items[j - gap]
                j -= gap
            items[j] = val
        gap //= 2

def run_test(sort_func, file_name):
    with open(file_name, "w") as f:
        # warmup
        sort_func(100, 100, 10)

        num_lists = 100
        items_per_list = 10000

        print("num_lists,items_per_list,cpu,gpu", file=f)
        while num_lists <= 100000:
            cpu_time, gpu_time = sort_func(num_lists, items_per_list, 10)
            print("{num_lists},{items_per_list},{cpu},{gpu}".format(num_lists=num_lists,
                                                                        items_per_list=items_per_list,
                                                                        cpu=cpu_time, gpu=gpu_time),
                  file=f)
            f.flush()
            num_lists *= 10

if __name__ == "__main__":
    spark = SparkSession.builder.appName("GPUSort").getOrCreate()

    def sort(n, m, partitions):
        data = spark.sparkContext.parallelize(range(n), partitions)
        lists = data.map(lambda i: [randint(0, 1000000) for _ in range(m)]).cache()
        lists2 = lists.map(lambda i: i).cache()

        lists.count()
        lists2.count()

        def cpu_version():
            lists.foreach(shellsort)
        def gpu_version():
            GPURDD(lists2).foreach(shellsort)


        cpu_time = get_time(cpu_version)
        gpu_time = get_time(gpu_version)

        for l1, l2 in zip(lists.collect(), lists2.collect()):
           assert l1 == l2

        return cpu_time, gpu_time

    run_test(sort, "shellsort_results.csv")

    spark.stop()
```

Appendix F

PI ESTIMATION TEST

This appendix contains the implementation of the pi estimation test used to validate the functionality of GPURDD, as discussed in section 5.6, and also used to benchmark GPURDD, as discussed in section 6.4.

This appendix contains code that is used in all the variations of the pi estimation tests, as well as the implementation of each version of the pi estimation tests. Below is the code that is used across all the pi estimation tests:

```python
import math

class CoordinatePair:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def count(self):
        return 1 if math.pow(self.x * 2 - 1, 2) + math.pow(self.y * 2 - 1, 2) < 1 else 0

    def check(self):
        return True if math.pow(self.x * 2 - 1, 2) + math.pow(self.y * 2 - 1, 2) < 1 else False

    def __eq__(self, other):
        if isinstance(other, CoordinatePair):
            return self.x == other.x and self.y == other.y
        return False


def run_test(calc_pi, file_name):
    with open(file_name, "w") as f:
        # warmup
        calc_pi(10000, 10)

        darts = 10000
        print("darts,parts,cpu,gpu", file=f)
        while darts <= 1000000000:
            parts = 10
            cpu_time, gpu_time = calc_pi(darts, parts)

            print("{},{},{},{}".format(darts, parts, cpu_time, gpu_time), file=f)
            darts *= 10
            f.flush()
```

F.1 Map Test

This section contains the code for the first version of the pi estimation test, which uses `map` to check whether coordinate points are in the unit circle.

```
from random import random

from pyspark.sql import SparkSession

from pi import CoordinatePair, run_test
from gpurdd import GPURDD
from util import get_time


if __name__ == "__main__":
    spark = SparkSession.builder.appName("Pi_Map").getOrCreate()

    def calc_pi(n, partitions):
        data = spark.sparkContext.parallelize(range(n), partitions)
        randoms = data.map(lambda i: CoordinatePair(random(), random())).cache()
        randoms.count()

        def inside_unit_circle(cp):
            return cp.count()

        cpu_rdd = randoms.map(inside_unit_circle)
        gpu_rdd = GPURDD(randoms).map(inside_unit_circle)

        def cpu_version():
            cpu_rdd.count()

        def gpu_version():
            gpu_rdd.count()

        cpu_time = get_time(cpu_version)
        gpu_time = get_time(gpu_version)

        assert cpu_rdd.collect() == gpu_rdd.collect()

        return cpu_time, gpu_time

    run_test(calc_pi, "pi_map_results.csv")

    spark.stop()
```

F.2 Filter Test

This section contains the code for the second version of the pi estimation test, which uses `filter` to remove points outside the unit circle.

```
from random import random

from pyspark.sql import SparkSession

from pi import CoordinatePair, run_test
```

```python
from gpurdd import GPURDD
from util import get_time


if __name__ == "__main__":
    spark = SparkSession.builder.appName("Pi_Filter").getOrCreate()

    def calc_pi(n, partitions):
        data = spark.sparkContext.parallelize(range(n), partitions)
        randoms = data.map(lambda i: CoordinatePair(random(), random())).cache()
        randoms.count()

        def inside_unit_circle(cp):
            return cp.check()

        cpu_rdd = randoms.filter(inside_unit_circle)
        gpu_rdd = GPURDD(randoms).filter(inside_unit_circle)

        def cpu_version():
            cpu_rdd.count()

        def gpu_version():
            gpu_rdd.count()

        cpu_time = get_time(cpu_version)
        gpu_time = get_time(gpu_version)

        assert cpu_rdd.collect() == gpu_rdd.collect()

        return cpu_time, gpu_time

    run_test(calc_pi, "pi_filter_results.csv")

    spark.stop()
```

F.3 Combined Test

This section contains the code for the final version of the pi estimation test that combines checking whether points are in the unit circle and counting them in a single `map` call.

```python
from random import random
from operator import add

from pyspark.sql import SparkSession

from pi import CoordinatePair, run_test
from gpurdd import GPURDD
from util import get_time

if __name__ == "__main__":
    spark = SparkSession.builder.appName("Pi_Combined").getOrCreate()

    def calc_pi(n, partitions):
        data = spark.sparkContext.parallelize(range(n//100), partitions)
        lists = data.map(lambda i: [CoordinatePair(random(), random()) for _ in range(100)]).cache()
        lists.count()

        def count_and_reduce(cps):
```

```
        sum = cps[0].count()
        for i in range(1, len(cps)):
            sum += cps[i].count()
        return sum

    cpu_rdd = lists.map(count_and_reduce)
    gpu_rdd = GPURDD(lists).map(count_and_reduce)

    def cpu_version():
        cpu_rdd.count()

    def gpu_version():
        gpu_rdd.count()

    cpu_time = get_time(cpu_version)
    gpu_time = get_time(gpu_version)

    assert cpu_rdd.collect() == gpu_rdd.collect()

    return cpu_time, gpu_time

run_test(calc_pi, "pi_combined_results.csv")

spark.stop()
```