SWAT: A Programmable, In-Memory, Distributed, High-Performance Computing Platform

Max Grossman
Rice University
6100 Main St, Houston, TX, USA
jmg3@rice.edu

Vivek Sarkar
Rice University
6100 Main St, Houston, TX, USA

ABSTRACT

The field of data analytics is currently going through a renaissance as a result of ever-increasing dataset sizes, the value of the models that can be trained from those datasets, and a surge in flexible, distributed programming models. In particular, the Apache Hadoop [1] and Spark [5] programming systems, as well as their supporting projects (e.g. HDFS, SparkSQL), have greatly simplified the analysis and transformation of datasets whose size exceeds the capacity of a single machine. While these programming models facilitate the use of distributed systems to analyze large datasets, they have been plagued by performance issues. The I/O performance bottlenecks of Hadoop are partially responsible for the creation of Spark. Performance bottlenecks in Spark due to the JVM object model, garbage collection, interpreted/-managed execution, and other abstraction layers are responsible for the creation of additional optimization layers, such as Project Tungsten [4]. Indeed, the Project Tungsten issue tracker states that the "majority of Spark workloads are not bottlenecked by I/O or network, but rather CPU and memory" [20].

In this work, we address the CPU and memory performance bottlenecks that exist in Apache Spark by accelerating user-written computational kernels using accelerators. We refer to our approach as Spark With Accelerated Tasks (SWAT). SWAT is an accelerated data analytics (ADA) framework that enables programmers to natively execute Spark applications on high performance hardware platforms with co-processors, while continuing to write their applications in a JVM-based language like Java or Scala. Runtime code generation creates OpenCL kernels from JVM bytecode, which are then executed on OpenCL accelerators. In our work we emphasize 1) full compatibility with a modern, existing, and accepted data analytics platform, 2) an asynchronous, event-driven, and resource-aware runtime, 3) multi-GPU memory management and caching, and 4) ease-of-use and programmability. Our performance evaluation demonstrates up to 3.24× overall application speedup relative to Spark across six machine learning benchmarks, with a detailed investigation of these performance improvements.

HPDC'16, May 31-June 04, 2016, Kyoto, Japan

ISBN 978-1-4503-4314-5/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2907294.2907307

Keywords

distributed, heterogeneous, Spark, GPU, OpenCL, data analytics

1. INTRODUCTION

The introduction of the Apache Hadoop [1] and Spark [5] programming systems to the data analytics community has greatly simplified the task of processing large datasets. While users celebrate the programmability of Hadoop, Spark, and the frameworks built on top of them, performance tuning of these systems is non-trivial. Their construction on an interpreted, managed runtime improves the flexibility of Spark and Hadoop, but this added layer of abstraction makes it difficult to tune locality, instruction scheduling, memory allocation, disk and network access, and other low-level performance knobs. Indeed, Spark's creation was partially motivated by criticisms of Hadoop's disk bottlenecks, while Project Tungsten [4] is motivated by CPU and memory bottlenecks in Spark.

Therefore, while the problem of building distributed applications to operate on large datasets is being addressed effectively, the challenge of achieving computational efficiency in large-scale systems without sacrificing programmability remains open. In particular, much of the existing work in the open source community has focused on fixing the I/O bottlenecks of these distributed programming systems, which has led to the computational inefficiencies becoming more pronounced. One recent approach to improving the computational performance of these big data analytics platforms has been the exploration of the use of accelerators, such as GPUs. We call these accelerator-based platforms *accelerated data analytics platforms* (ADAs).

Both of the leading data analytics platforms listed above (Hadoop and Spark) are JVM-based, but supporting full JVM execution on accelerators is infeasible. To bridge this gap, past work has taken a number of approaches. We characterize these approaches along four axes: the generality of the API exposed to the user, the degree to which low-level accelerator details are exposed to the user, compatibility with existing frameworks, and the flexibility/efficiency of the framework's runtime.

To one extreme, fixed-function frameworks offer high-level, domain-specific APIs which are highly programmable but also inherently inflexible. Some well-known examples are

BIDMach [8], Caffe [14], and HeteroSpark [17]. The specialization of fixed-function frameworks allows them to optimize both their performance and programmability for the domain they target, but limit their applicability to any others.

On the other hand, fully programmable ADA frameworks such as HeteroDoop [21] or HadoopCL [11] enable the implementation of novel algorithms and are more generally useful, but often pay the price of added overhead. For example, HadoopCL uses runtime code generation of accelerator kernels from JVM bytecode to support accelerated execution. This code generation step adds overhead to each job's execution, but augments the framework's overall flexibility.

While most ADAs choose to expose high-level accelerator-agnostic APIs, SparkCL [22], GPMR [23], and Glasswing [10] expose low-level details of accelerator hardware to the user, such as thread IDs and accelerator-specific address spaces. In our work, we follow the trends we observe in the community and hide hardware details from the user. This approach limits some user optimizations, but improves programmability.

While compatibility with existing frameworks is partly a software engineering concern, it is also an important constraint on the research problem. In past work, HadoopCL and SparkCL required modifications to the core code of Hadoop or Spark. While this approach enables applications to run unmodified and leverage optimizations that are not possible from outside of these frameworks, there are major barriers to broad acceptance in the long term if these extensions are not incorporated in the main development trunk of the underlying open source projects. Frameworks like HeteroDoop introduce entirely new APIs that sit on stop of other frameworks but which require a complete rewrite of any existing applications. Frameworks like MapCG [13] construct entirely new APIs and runtimes, limiting both the portability of legacy applications and the ability to co-locate them with existing frameworks like Spark and Hadoop. In this work, we build our solution as a third-party library which maintains the look and feel of Apache Spark and runs on top of Spark without requiring modifications to the internals of Spark.

Finally, the runtime system of an ADA is an important factor in its performance and stability. Related work like HadoopCL and HeteroDoop place a heavy emphasis on the efficient management of resources on behalf of the user, addressing out-of-memory errors and efficient scheduling without bothering the user with such low-level concerns. HadoopCL even includes a component to automatically determine at runtime which kernels to run in the JVM, on an accelerator, or on the CPU natively. Other work minimizes the safety for the user 1) by allowing them to allocate arbitrary amounts of accelerator memory, 2) by using blocking operations rather than asynchronous tasks, and 3) by generally requiring the user to do more manual resource management without protecting them from mistakes. We focus on building a flexible, asynchronous, and event-driven runtime that transparently handles resource management for the user.

In this paper we describe the open-source Spark With Accelerated Tasks (SWAT) framework, available at https://github.com/agrippa/spark-swat. Table 1 compares the relevant characteristics of SWAT with those of several related ADAs. SWAT accelerates the user-written computational kernels of Apache Spark jobs using OpenCL-supporting GPUs. SWAT is not fixed function: it allows Spark programmers to write custom kernels in high level JVM languages like Scala. SWAT is high-level: it does not expose accelerator hardware details, maintaining the same APIs as Spark. It fully integrates with the Spark framework as a third-party JAR, requiring only minimal code re-write for existing Spark applications. SWAT uses an asynchronous, event-driven, and resource-aware runtime to accurately and efficiently manage intra-node resources for the programmer.

The remainder of this paper is structured as follows. Section 2 will briefly describe Apache Spark, as well as explain the characteristics of GPU accelerators that make them relevant to data analytics workloads. Section 3 will describe the techniques and contributions of this work in detail. Section 4 will evaluate the performance benefits of those methods experimentally. Section 5 will conclude with a discussion of our approach and a summary of future work.

2. BACKGROUND

In this work, we combine Apache Spark and GPU accelerators to produce a programmable and high-performance data analytics system. This section briefly provides background on Spark and GPUs, and puts this work in the context of our past work on combining Hadoop MapReduce and GPUs.

2.1 Apache Spark

Apache Spark is a distributed, multi-threaded, in-memory programming system. The core abstraction of Apache Spark is that of a resilient distributed dataset (RDD). An RDD represents a distributed vector of elements. Elements in an RDD can be of any serializable type. RDD creation is lazy: creating an RDD object in a Spark program does not necessarily evaluate and populate that RDD. Only certain operations in Spark programs force evaluation, resulting in long chains of lazily evaluated RDDs as one RDD is transformed into another. RDD resiliency derives from their ancestry tracking. By persisting information on how RDDs were created rather than their actual contents, Spark guarantees that lost data can be recovered through recomputation without storing large amounts of intermediate data on disk.

A single RDD is split into multiple partitions. All elements in the same partition are stored on the same machine, but different partitions may be stored on different machines. Hence, partitions are the granularity of distribution in Spark.

One of Spark's strengths is its API, i.e. the transformations that it supports on RDDs. Spark transformations run in parallel across the machines that an RDD is stored on. Transformations are functional: they are applied to one RDD and produce another. This generally leads to long chains of lazily evaluated RDDs, linked by functional transformations. The transformations that Spark supports include `map`, `reduce`, `filter`, `reduceByKey`, `groupByKey`, `join`, and `distinct`. This variety of transformations greatly expands the flexibility of Spark relative to its predecessor, Hadoop MapReduce.

Transformations generally take some Scala lambda `f`, apply it to the input RDD using the semantics of the transformation, and produce some output RDD. For example, the `map` transformation applies `f` to each element of the input RDD, producing the corresponding element in the output RDD. `filter`, on the other hand, applies `f` to each element

Framework	Fixed Function	Low-Level	Compatibility	Runtime Flexibility
SWAT	No	No	No code change, No data transorms	Asynchronous runtime, Full platform management
HadoopCL	No	No	Small code change, Major data transforms	Pipelined runtime, Auto-scheduling
HeteroDoop	No	No	No code compatibility, shared runtime with Spark	Transparent resource management
MapCG	No	No	None	Assumes fixed memory usage, work partitioned across CPU + GPU
GPMR	No	Yes	None	Efficient support of many-GPU execution through augmentation of the MapReduce pipeline
Glasswing	No	Yes	None	Low-level OpenCL-based APIs
SparkCL	No	Yes	No code compatibility, shared runtime with Spark	None
HeteroSpark	Yes	No	Callable from Spark jobs	Unspecified
BIDMach	Yes	No	None	Minimal
Caffe	Yes	No	None	Minimal

Table 1: A comparison of relevant characteristics of various ADAs, including whether they are fixed-function, whether they expose low-level hardware details to the user, how much code rewrite is necessary for an existing Hadoop/Spark application, and the flexibility of the underlying runtime system.

and stores that element in the output RDD if `f` returns true.

In this paper, we refer to the processing of a single RDD partition by a single transformation as a Spark "task".

In-memory caching is also an important feature of Spark. Spark allows users to explicitly mark certain RDDs as "cached", indicating that the programmer would like Spark's runtime to make a best-effort at keeping the partitions of this RDD in memory. This may benefit performance when the output of one transformation is immediately fed into another as input, or when a single RDD is accessed repeatedly.

Spark also supports broadcast variables. Spark broadcast variables are read-only data structures accessible on every node of a Spark cluster. Broadcast variables are an efficient way to share read-only data among all tasks in a Spark job.

2.2 GPUs

It should be apparent that Spark is a highly parallel framework for processing large amounts of data. This computational pattern fits well with GPUs.

GPUs are throughput-optimized devices which are generally attached to a host processor (i.e., CPU) but physically separate from it. They perform best relative to multi-core CPUs on highly parallel workloads with simple control flow (i.e. minimal conditional divergence between threads) and large memory bandwidth requirements.

GPU memory hierarchies also contain special-purpose memory that allow programmers to manually place data with certain characteristics in appropriate parts of the hierarchy. For example, GPU constant memory is useful for constant data structures read by all threads on the GPU. GPU shared (or scratchpad) memory is on-chip and useful for storing frequently accessed, small data structures.

Programming GPUs requires the use of low-level, data parallel programming models such as CUDA or OpenCL. These models allow programmers to manage GPU memory, GPU communication, and massively parallel kernel creation. They are generally considered by be verbose, especially when compared to high-level languages like Scala and Java.

The data parallelism, memory bandwidth, and low-level programming models of GPUs make them complementary to data analytics platforms like Spark. Both exhibit wide parallelism, either across many cores in a chip or across a whole distributed system. Spark processes large datasets in a streaming model, which matches well with GPUs' high memory bandwidth. Spark offers a high-level programming model but suffers from performance bottlenecks, while GPUs suffer from low-level programming models but have a high peak computational performance.

2.3 HadoopCL

In past work called HadoopCL, we explored accelerating Hadoop MapReduce using GPUs. While the motivations of HadoopCL and SWAT are similar, the constraints and resulting systems are not. Hadoop and Spark are different programming models that expose different abstractions and are entirely disjoint in their backend implementations. As a result, the accelerated frameworks HadoopCL and SWAT are also entirely disjoint. HadoopCL simplified the problem of building an efficient ADA by modifying core Hadoop code, introducing custom data structures, requiring the programmer to use HadoopCL-specific APIs, and significantly limiting the logic that could be written in accelerated kernels. On the other hand, SWAT is constructed entirely as a third-party library, does not introduce custom data structures, has an API that consists of a single method, and uses new techniques in code generation to broaden the JVM features that can be used in accelerated kernels. SWAT's runtime system is also novel, doing a better job of seamlessly combining multi-threaded JVM and OpenCL execution in an event-driven model than HadoopCL's runtime was able to.

2.4 Contributions

In this work, we build a novel accelerated data analytics platform, called SWAT, by accelerating Apache Spark with GPUs. Our main contributions beyond past work include:

1. Full compatibility with a modern data analytics platform, Apache Spark, by minimizing user-visible changes

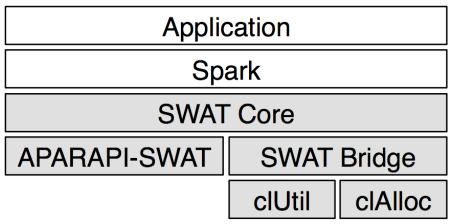

Figure 1: The SWAT software stack. Colored boxes indicate components that are novel contributions of this work.

to the API.

2. Extensions to the bytecode-to-OpenCL code generation work from [12] to support Spark-specific data structures, such as dense and sparse vectors from MLlib [24].
3. An asynchronous, event-driven, resource-aware runtime for managing JVM and accelerator resources.
4. A multi-GPU memory management and caching layer, with experiments in device locality-aware scheduling.
5. An evaluation of SWAT on six diverse machine learning benchmarks with detailed analysis to explain performance improvements and losses.

3. METHODS

The software stack of SWAT is illustrated in Figure 1. In this section, we start by describing the SWAT API and then discuss each of these layers, starting at the bottom with the clAlloc and clUtil modules and moving progressively higher in the stack.

3.1 SWAT API

In a vanilla Spark program, RDDs are created by applying transformations or operations to other RDDS. In the example code snippet below, an RDD of integers is created from a file stored in HDFS, and a new RDD is created where element `i` contains the value of element `i` in the first RDD, multipled by two:

```
val input = sc.objectFile[Int](hdfsPath)
val doubled = input.map(i => 2 * i)
```

To run this kernel on an accelerator, SWAT simply requires that the `input` RDD be wrapped by a custom SWAT RDD object using a `cl` API call (shown below). No other code change is required, and this is the only method exposed by SWAT.

```
val input = cl(sc.objectFile[Int](hdfsPath))
val doubled = input.map(i => 2 * i)
```

SWAT currently supports intercepting and accelerating calls to Spark `map` and `mapValues` transformations. Other transformations could be supported, but we have not found motivating application kernels that use other transformations and would benefit from acceleration. For example, `filter` kernels tend to be short-lived and the GPU offload time would be dominated by overheads. Future work could investigate the use of kernel fusion across chained transformations to produce larger GPU kernels. These fused kernels might offset the offload overheads, making offload of lightweight transformations like `filter` profitable.

3.2 clUtil and clAlloc

clUtil is a simple utility library written by the authors that simplifies management of OpenCL devices. It makes tasks like device selection, device setup, querying device information, and kernel launches less verbose than the lower-level OpenCL APIs.

clAlloc is a thread-safe, single-accelerator memory management library built on top of the OpenCL APIs. It exposes two data structures: 1) an `allocator` object for each OpenCL device in a platform that serves as a context/handle for clAlloc operations on that device, and 2) `region` objects which represent a contiguous block of allocated memory on a single OpenCL device. Its API is as follows:

1. `clalloc_init(device)`: Initialize an `allocator` instance for the selected device.
2. `cl_allocate(nbytes, allocator)`: Allocate `nbytes` bytes on the device associated with `allocator`, returning a `cl_region` handle for the allocated memory or `NULL` if the allocation failed.
3. `cl_free(region, try_to_keep)`: Release the device memory represented by `region` for future allocations. If `try_to_keep` is true, clAlloc will make a best-effort to not use that memory to satisfy future allocations as it may be re-used soon.
4. `cl_reallocate(region)`: Using the provided `region`, attempt to re-allocate the same device memory. Successfully re-allocating memory guarantees that it has not been used to satisfy another allocation since `region` was originally allocated, and so its state is consistent with previous operations performed on the same `region`. If the `region` is not already free this simply increments a reference counter, allowing multiple kernels to share the same `region`.
5. `get_pinned(region)`/`release_pinned(buf)`: Fetch or release a page-locked buffer `buf` in host memory that matches the size of `region`. Page-locked buffers are necessary for performing asynchronous communication to or from accelerators.
6. `set_region(region, buf, nbytes)`/`get_region(buf, region, nbytes)`: Fill or fetch the contents of a region on an OpenCL device using a corresponding host buffer.

clAlloc adds a higher-level API on top of the standard OpenCL APIs, including features that enable higher layers of the software stack to perform inter-kernel data sharing and efficient data communication. clAlloc pre-allocates all device memory when it is initialized and partitions memory up for allocation requests on-demand using the OpenCL `clCreateSubBuffer` API.

Free device memory is represented by a free list, sorted by offset into the device memory address space. When a clAlloc `region` is freed with `cl_free` and with `try_to_keep` set to false, it is merged into any neighboring regions to reduce fragmentation. If `try_to_keep` is set to true, it is not merged.

clAlloc also stores free regions in buckets for efficient allocation. For a single device, `B` buckets are created. Each bucket `b` from 0 to `B`-1 stores all free regions on that device

with a size between 2^b (inclusive) and 2^{b+1} (exclusive). A special-purpose bucket is used to store any regions larger than $2^B - 1$. The free regions in each bucket are kept sorted by size, from smallest to largest. Regions freed with `try_to_keep` set to true are not kept in these buckets, only in the global free list for each device.

When allocating `nbytes`, clAlloc starts with the smallest bucket that may have a region of size `nbytes`, searching larger buckets until a free region that is large enough to satisfy this allocation is found. The free region is then trimmed to `nbytes`, any leftover space is re-inserted in the free list and free buckets, and the allocated region is returned.

If no free region is found in the buckets list, the allocator reverts to a linear search of the device-global free list for adjacent free regions that can be merged to produce a sufficiently large free region to satisfy this allocation. This step will only succeed where the previous one failed if some regions freed with `try_to_keep` set to true can be merged with neighboring free regions to de-fragment device memory. If this step fails, a `NULL` region is returned.

Note that clUtil and clAlloc are only exposed to the internals of SWAT, and not used directly by a SWAT programmer.

3.3 APARAPI-SWAT

APARAPI is an open source, general-purpose framework that enables transparent execution of Java programs on OpenCL devices through an API similar to Java's `Runnable`. APARAPI includes a runtime code generator from JVM bytecode to OpenCL kernels and handles all OpenCL memory allocation, data transfer, and kernel execution for the programmer.

In this work, we use APARAPI's code generator to translate the Scala lamdbas passed to Spark transformations (illustrated in Section 3.1) into OpenCL kernels that can be executed on an accelerator. This code generation is performed at runtime, converting dynamically loaded JVM bytecode into the OpenCL kernel language.

In past work, we have extended APARAPI's code generator to support 1) dynamic memory allocation inside OpenCL kernels through a retry-on-failure approach, and 2) JVM object references in bytecode translated to OpenCL kernels through generation of equivalent native structs and automatic serialization of those objects. For more details, see our previous work in [12].

In this work, we build on [12] to also support references to Spark-specific data structures in kernels. In particular, we support the `SparseVector` and `DenseVector` classes from Spark's MLlib, and the Scala `Tuple2` class used to store key-value pairs in Spark. An example of the OpenCL kernel code generated to store and manipulate a `DenseVector` object is shown below:

```
typedef struct __attribute__ ((packed)) dv {
  __global double*  values;
  int  size;
  int  stride;
} DenseVector;

static int DenseVector__size(
    __global DenseVector *this) {
  return (this->size);
}

static double DenseVector__apply(
    __global DenseVector *this, int index) {
  return (this->values)[this->stride * index];
}
```

One important item to note in the definition of the `DenseVector` struct is the addition of a `stride` field. During serialization of `DenseVector` JVM objects to native structs that can be accessed on the GPU, we tile and stride `DenseVector` objects to improve memory access coalescing on the GPU. This transformation places the `i`th element of neighboring `DenseVector` objects adjacent to each other. In our implementation, we tile 32 `DenseVector` objects together before striding them because NVIDIA GPUs schedule threads in "warps" of 32 threads. However, the implementation is structured so that this can be easily tuned when porting to new architectures. These same optimizations are also performed on `SparseVector` objects.

The automatic optimization of the user-written kernels during code generation is beyond the scope of this work, but ideas from related work [25][16][7] could be integrated into this code generator in the future.

3.4 SWAT Bridge

Sitting on top of clAlloc and clUtil is the SWAT Bridge, a bridge between the components of SWAT running in the JVM and those sitting on top of OpenCL. As such, SWAT Bridge's upward exposed APIs are expressed in terms of JVM or Spark objects, which it then translates into commands to the OpenCL-centric layers below.

The Bridge's primary responsibilities are the caching of data on OpenCL devices across kernels and tasks, the creation and management of native SWAT contexts, the setting of arguments to OpenCL kernels, and the management of asynchronously executing OpenCL operations (including kernel executions and data communication). It exposes APIs to the JVM that enqueue work for accelerators and block or poll on their completion. At a high level, the Bridge accepts input buffers from the JVM, places them on the accelerator, launches computation on those buffers at the JVM's request, retrieves outputs, and signals the JVM on completion of various stages and as resources are released.

Described in Section 2.1, Spark uses in-memory caching to address I/O bottlenecks in past frameworks without losing resiliency guarantees. Logic in the Bridge supports similar caching of data on OpenCL devices. In particular, we investigated caching RDD partitions and broadcast variables on the GPU.

The Bridge stores two mappings for caching data on OpenCL devices: one mapping from unique RDD partition IDs to their clAlloc regions, and another from unique broadcast variable IDs to their clAlloc regions. When layers higher in the software stack indicate that a partition or broadcast variable should be allocated and populated on a device, the Bridge first checks if an entry already exists for it in one of the cached mappings. If it does and a call to `cl_reallocate` succeeds on it, the Bridge can skip creating a separate allocation. This saves space on the device through deduplication, and reduces data communication to the device. When freeing regions associated with cacheable data, the `try_to_keep` flag is set to true.

In practice, we found that caching partitions of RDDs on the device was not useful. The limited size of device memory and the scale of Spark datasets meant that RDD partitions

were never re-used before being evicted from device memory: the memory used to store them had to be allocated to store other data before the application came back around to a re-use of that partition. In fact, the increased fragmentation of device memory and added overhead of managing the Bridge's RDD cache mapping generally hurt performance. Broadcast variables, on the other hand, are frequently used, may be shared across multiple stages of a Spark application, and are usually smaller than an RDD partition. We generally see benefits from caching them. For the evaluation conducted in Section 4, we do not cache RDD partitions in device memory but do cache broadcast variables.

SWAT does not make use of GPU constant or scratchpad memory for a number of reasons. The OpenCL APIs for accessing GPU constant memory severely restrict the ability of our runtime to store general, dynamically initialized data in constant memory. We could not design a general technique to make use of constant memory for arbitrary data and so, rather than impinging on the generality of our framework by asking programmers to explicitly handle constant memory, we classify it as future work.

Preliminary experiments with scratchpad memory did show performance benefits from using it to store certain thread-local, SWAT internal data structures or broadcast variables. However, the OpenCL kernel language requires annotating every pointer with the address space it points to. The small size of GPU scratchpad memory meant that storing even moderately sized broadcast variables would require some sort of tiling or partitioning, where part of the broadcasted variable was stored in scratchpad memory with the remainder stored in global device memory. This would have meant generating multiple versions of any OpenCL kernel function that used broadcast variables with different signatures depending on the address space of the data being referenced. This approach would have also caused thread divergence as different threads went down the scratchpad memory code path or the device memory code path. Further experiments showed that this added complexity negated the performance benefits of scratchpad memory for the cases where we could automatically identify good candidates for storage in scratchpad. Again, rather than ask the programmer to explicitly manage scratchpad memory or reduce the generality of our framework, we classify its use as future work.

Besides caching, the bridge's other main responsibility is exposing an API that allows higher layers to enqueue asynchronous OpenCL operations and check for their completion. In support of this, the bridge implements an asynchronous runtime that coordinates asynchronous OpenCL operations with the host application using pthreads condition variables, OpenCL events, and OpenCL event callbacks. This runtime is illustrated in Figure 2. Below the dotted line in Figure 2 are OpenCL operations managed by the OpenCL runtime including data communication, kernel execution, and SWAT-specific callbacks. These callbacks are identified by the light gray boxes. All of these operations are asynchronous with respect to the host JVM, with OpenCL events being used to maintain inter-operation dependencies.

The functions that the SWAT Bridge exposes to higher layers in SWAT are listed below:

1. `setPinnedArrayArg`: This function initiates the transfer of the contents of a host page-locked buffer to a buffer on an OpenCL device. It also sets the appropriate arguments of an OpenCL kernel to point to the same OpenCL buffers. This call is non-blocking, and creates OpenCL events for later operations to depend on.

2. `launchKernel`: This function launches a new kernel processing data initialized using `setPinnedArrayArg`. This kernel is made dependent on the writes started by `setPinnedArrayArg` using OpenCL events. Each kernel launch is uniquely identified by a 64-bit sequence number, which is incremented by one on each kernel launch.

3. `waitForKernel`: This function forces the current JVM thread to wait for a specific kernel launch to complete, identified by its sequence number. This involves a wait on a condition variable which is set by the box labelled "Release Device Buffers" in Figure 2.

4. `waitForInputBuffersRelease`: This function forces the current JVM thread to wait for a set of transfers to the device to complete, signalling that the host buffers they originate from are now available for re-use by the host.

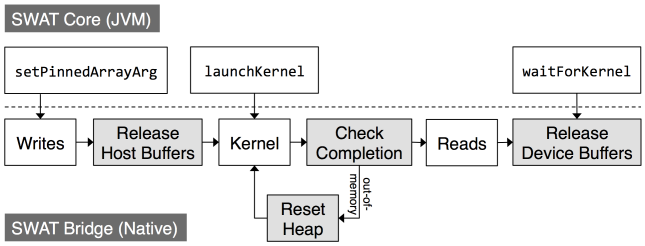

Figure 2: The flow of event-driven actions in the SWAT Bridge at runtime.

All of these APIs are thread-safe as they may be concurrently called by multiple JVM threads.

3.5 SWAT Core

SWAT Core refers to the components of SWAT that sit inside the JVM. SWAT Core runs inside a Spark Worker JVM. The interface between Spark and SWAT is a custom SWAT RDD class (illustrated in Figure 3). When Spark has a partition to process it calls a `compute` method on the custom RDD, passing it an iterator over an input partition. The `compute` method returns an iterator over output items. SWAT currently adds two custom RDD classes: one each for the Spark `map` and `mapValues` transformations. Both of these classes hand off processing of the input partitions to a shared code base in `CLProcessor`.

`CLProcessor` has four main responsibilities: 1) setup and configuration of the SWAT environment, 2) input buffering and serialization, 3) launching a GPU batched kernel, and 4) output deserialization and writing. At a high level, `CLProcessor` accumulates many input items from the input iterator, launches a batched OpenCL kernel on the accumulated inputs, and returns the accumulated outputs to Spark through the iterator which was returned by the RDD `compute` method.

There are five categories of objects that the `CLProcessor` is responsible for initializing:

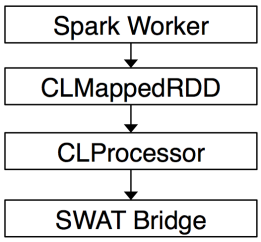

Figure 3: Example stack trace of entry point to SWAT Core.

1. OpenCL objects: This includes SWAT-specific items such as clAlloc allocators, as well as OpenCL-specific items like compiled kernels and OpenCL contexts.

2. Native Input Buffers: These are JVM handles on native, page-locked buffers allocated from clAlloc. Multiple page-locked buffers may be grouped into a single Native Input Buffer handle if they are required to serialize a given input type. For example, accumulating vectors from a `DenseVector` input iterator requires three native input buffers: one buffer to store the values of each vector, one buffer to store the length of each vector, and one buffer to store the offset of each vector in the values buffer.

3. JVM Input Buffers: These are small JVM objects that contain the logic to serialize items from an input iterator into backing Native Input Buffers. Sometimes it is necessary to store small temporary buffers in JVM Input Buffers. An actively buffering JVM Input Buffer is always backed by a Native Input Buffer in which it stores the accumulated and serialized input items.

4. Native Output Buffers: Page-locked host buffers that SWAT Bridge transfers the outputs of an OpenCL kernel into, asynchronously.

5. JVM Output Buffers: JVM objects backed by Native Output Buffers that expose an iterator interface which `CLProcessor` can use to deserialize and fetch output elements from Native Output Buffers.

Only one JVM input buffer and JVM output buffer are created by `CLProcessor`. Multiple native input and output buffers are created, but are limited to a fixed number to prevent out-of-memory errors caused by excessive native buffer allocation. Only a single JVM input or output buffer is necessary as different native buffers can be swapped in and out as the storage backing the JVM buffers' interfaces.

The CLProcessor uses two JVM threads: a dedicated reader thread and the main Spark thread. A reader thread is spawned by the main Spark thread when the SWAT RDD `compute` method is called. The reader thread retrieves an iterator for a given input partition. The reader thread is responsible for accumulating items from the input iterator, through the JVM Input Buffer, and into a Native Input Buffer. It then initiates the asynchronous input copies to the accelerator from the Native Input Buffer and launches an asynchronous kernel to process them, using the SWAT Bridge. Illustrative pseudocode for the reader thread is listed in Algorithm 1.

```
1  currentInputSeqNo = 0
2  lastSequenceNo = -1
3  done = false
4  jvmInputBuf = createJVMInputBuffer()
5  jvmInputBuf.setNativeInputBuf(fetchNativeInputBuffer())
6
7  while not done do
8      jvmInputBuf.accumulate(inputIter)
9
10     nextNativeInputBuf = fetchNativeInputBuffer()
11     jvmInputBuf.transferOverflowTo(nextNativeInputBuf)
12     jvmInputBuf.nativeInputBuf.copyToAccelerator()
13
14     currNativeOutputBuf = fetchNativeOutputBuffer()
15     bridge.setupOutputBuffers(currNativeOutputBuf)
16
17     kernelSequenceNo = currentInputSeqNo++
18     done = bridge.run(kernelSequenceNo)
19     if done then
20         lastSequenceNo = kernelSequenceNo
21     end
22 end
```

Algorithm 1: Pseudocode for SWAT Core reader thread.

The main Spark thread for the current partition first retrieves an output iterator from the SWAT RDD object's `compute` method and then repeatedly calls that iterator's `hasNext` and `next` methods to retrieve output items for the current partition, until `hasNext` returns false. `hasNext` checks that there are no remaining output items by verifying that the input iterator is finished, there are no pending OpenCL kernels, and that there are no pending output items left in any Native Output Buffers. `next`, on the other hand, either immediately returns an output item from a currently active Native Output Buffer or loads a new Native Output Buffer by waiting for the appropriate kernel launch to finish, based on a kernel sequence number. Illustrative pseudocode for the output iterator logic is listed in Algorithm 2.

```
1  currentOutputSeqNo = 0
2  jvmOutputBuffer = ∅
3
4  Procedure next
5      if jvmOutputBuffer == ∅ then
6          currNativeOutputBuffer =
7              bridge.waitForFinishedKernel(
8                  currentOutputSeqNo)
9          currentOutputSeqNo++
10         jvmOutputBuffer.fillFrom(currNativeOutputBuffer)
11     end
12     return jvmOutputBuffer.next
13
14 Procedure hasNext
15     return lastSequenceNo == -1 or
16         currentOutputSeqNo <= lastSequenceNo or
17         jvmOutputBuffer.hasNext;
```

Algorithm 2: Pseudocode for the SWAT Output Iterator.

Dataset	Size	# Items	Scala Type
Hyperlink	16 GB	1,289,970K	(Int, Int)
Census	14 GB	49,166K	DenseVector
ImageNet	1.3 GB	40,646K	(Int, DenseVector)

Table 2: Characteristics of each dataset.

4. EXPERIMENTAL EVALUATION

In this section we evaluate the performance gains and losses made using the SWAT framework. We start by considering its overall performance and scalability relative to Spark. We then look at task-level acceleration, i.e. how much faster the processing of a single RDD partition is in SWAT relative to Spark. Afterwards, we start looking at the underlying characteristics of selected applications running on SWAT to explain the higher level characteristics.

4.1 Experimental Setup

All benchmarks and metrics are evaluated on a hardware platform containing a 12-core 2.80GHz Intel X5660 CPU with 48GB of system RAM and two NVIDIA M2050 GPUs each with 2.5GB of device memory in each node. Nodes in this platform are connected by QDR Infiniband. Our experiments are limited to a maximum of nine nodes (one master, eight workers) by hardware availability. All experiments were run with 12 Spark executor threads in each node. The softwate platform consists of JDK 1.7.0_80, Scala 2.11.5, Spark 1.2.0, HDFS 2.5.2, and ICC 15.0.2. For the overall and task-level performance results, each benchmark was tested ten times at each node count. For the more detailed performance analysis, median runs were selected for study.

We use six benchmarks to evaluate SWAT:

1. PageRank: A graph algorithm that ranks nodes in the graph based on the nodes that link to them.
2. Connected: Connected components graph algorithm.
3. NN: A simple neural net implementation.
4. Fuzzy: A probabilistic clustering algorithm.
5. KMeans: An iterative clustering algorithm.
6. Genetic: A genetic, evolutionary algorithm. In this case, we use a genetic algorithm to find cluster centroids.

When writing these benchmarks for Spark, we chose to cache in-memory any RDD that was read more than once. We did this to keep the performance comparison fair, and believe it to be a reasonable choice. For example, the input points for the KMeans clustering algorithm are read on each iteration of the algorithm and so we mark them as cached. All applications were implemented using Spark's Scala API.

For these six benchmarks, we evaluate on three datasets. For PageRank and Connected we use the Hyperlink Graph available from the Web Data Commons [18]. For Fuzzy, KMeans, and Genetic we evaluate on the Census dataset available from the UCI Machine Learning Repository [6]. For NN we evaluate on a subset of the images in the ImageNet dataset [9]. The size of each dataset is listed in Table 2.

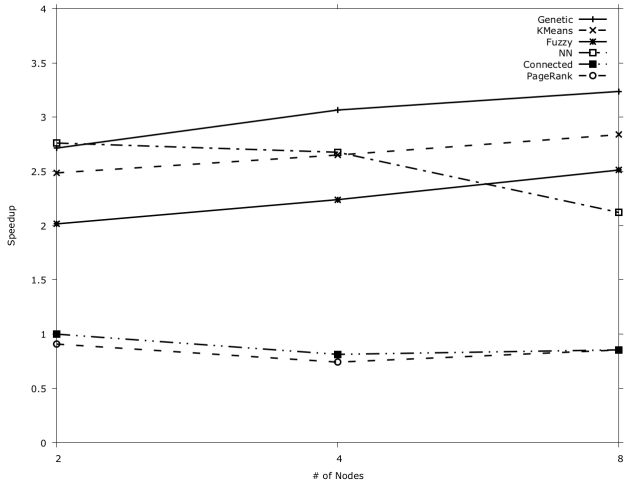

Figure 4: Overall speedup of each benchmark running on 1 master node and N worker nodes, for N = 2, 4, and 8.

4.2 Overall Speedup and Scalability

Figure 4 shows the overall speedup SWAT achieved relative to Spark on 2, 4, and 8 worker nodes. There are two clear categories of benchmarks: while Genetic, KMeans, Fuzzy, and NN all show speedups between 2× and 3.5×, PageRank and Connected either show no change, or slight slowdowns. We explain this through the characteristics of the applications: speedups are achieved when non-trivial computation is present in the application logic being accelerated by GPUs. For applications that are I/O bound on disk or network bandwidth and have small computational kernels, SWAT demonstrates no improvement. We support these claims in Sections 4.4, 4.5, and 4.6.

Figure 5 shows the scalability of each benchmark running on both Spark and SWAT when moving from two to eight executor nodes. Linear scalability would be denoted by a 4× speedup on the y-axis. At the scale of only eight executor nodes, it is difficult to make conclusions about the scalability of either framework. In general, neither consistently achieves linear scalability. However, these applications are not perfectly parallel and have collect or reduction stages which would make perfect scalability unlikely. Evaluating the scalability of the Spark framework is beyond the scope of this paper, but we observe no trends indicating a loss of scalability with SWAT.

We note that in Figure 5, the two applications with the worst scalability (PageRank and Connected) also demonstrated the lowest speedups in Figure 4. This poor scalability is caused by the same network and I/O bottlenecks that caused poor speedups when comparing SWAT to Spark.

4.3 Task-Level Speedup

Stepping down in the granularity of work being measured, recall that Section 2.1 defined a Spark task as the processing of a single RDD partition by a single transformation. When accelerated by SWAT, this includes all accelerator communication and computation. Figure 6 shows the average speedup for all different types of tasks in each benchmark. For most benchmarks, the task-level speedups are similar to the overall speedups. The primary outlier is NN:

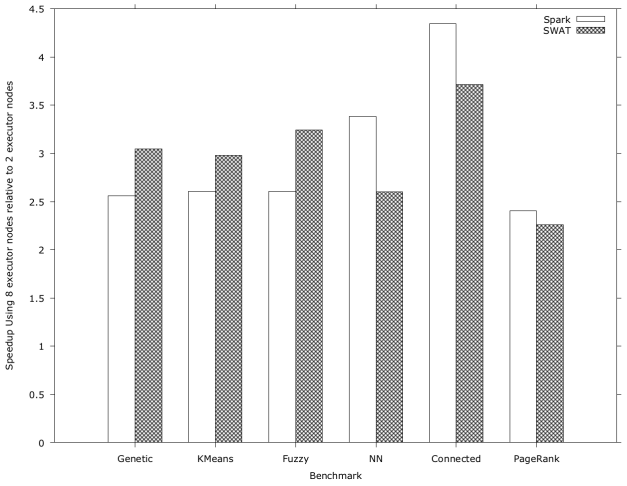

Figure 5: Speedup for each benchmark from two to eight nodes, using Spark or SWAT. This illustrates the scalability of each platform.

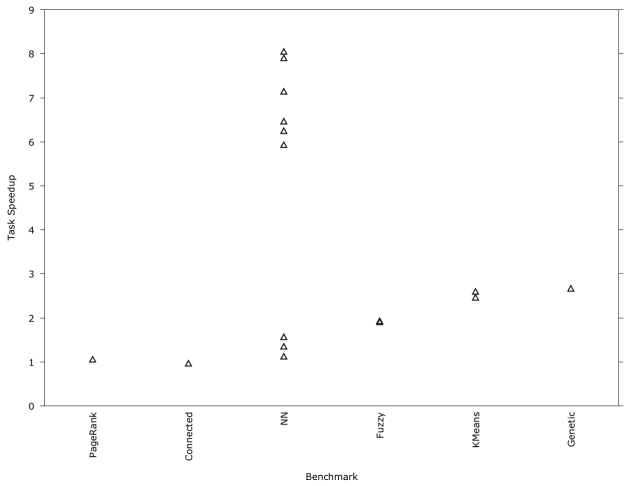

Figure 6: The task-level speedup averaged within each stage of each benchmark that is accelerated by SWAT. For example, KMeans runs two stages on the GPU and so we show two data points, each of which is the average speedup for all tasks in those stages.

Some of its tasks show between 5 and 9× speedup, despite only achieving ∼2× speedup overall. Further investigation showed that the longest transformation of the NN benchmark consumes 26% of overall job time. Tasks in this stage only achieved a speedup of 1.11× when using the GPU, limiting the speedup achievable by Amdahl's Law. While each task consumed on average 48 seconds, only ∼1 second of each task was spent using the SWAT runtime. The remainder is spent fetching shuffle outputs at the start of each task.

4.4 SWAT Execution Timelines

Stepping down another level of granularity, we select five threads from a node running the PageRank and Genetic benchmarks and visualize their behavior using a custom SWAT profiling tool. We use these two benchmarks to study more fine-grain characteristics of the SWAT framework, as they are representative of the workloads that perform well or poorly on SWAT. We categorize the work performed in SWAT into three categories:

1. Input I/O, which includes deserialization and disk I/O.
2. OpenCL operations, which includes both data communication with and execution on the OpenCL device. Section 4.5 will dive deeper into the time spent in specific OpenCL operations, such as communication and execution.
3. Output I/O, which includes serialization and disk I/O.

Figure 7 shows the execution timeline for the PageRank benchmark, and Figure 8 shows it for the Genetic benchmark. Clearly, PageRank is dominated by input and output I/O while Genetic is dominated by computation. Combining these observations with Amdahl's Law explains the higher overall speedups achieved for the Genetic benchmark compared to the PageRank benchmark.

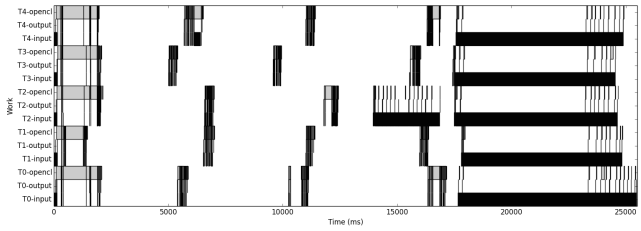

Figure 7: PageRank execution timeline. Light gray indicates input I/O, dark gray indicates OpenCL operations, and black indicates output I/O. No dark gray is visible at this time scale as little computation is performed in PageRank.

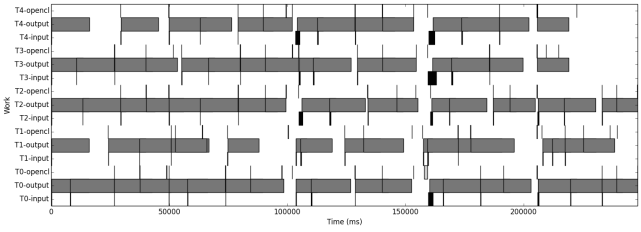

Figure 8: Genetic execution timeline. Light gray indicates input I/O, dark gray indicates OpenCL operations, and black indicates output I/O. Note that this figure is dominated by dark gray, indicating a large amount of time in OpenCL operations.

4.5 OpenCL Profiling

Finally, getting to the lowest work granularity, we use OpenCL event profiling [15] to analyze the time spent in

OpenCL operations (the dark gray bars in Figures 7 and 8). We subdivide OpenCL operations into data communication to the accelerator, computation on the accelerator, and data communication from the accelerator. OpenCL event profiling allows us to further split the time spent in each of these operations into:

1. Queued time: Time an operation spends in an OpenCL command queue on the host.

2. Submitted time: Time an operation spends submitted to the device driver, pending execution.

3. Run time: Time an operation spends running on the device.

Figure 9 plots the OpenCL profiling info for each benchmark. Each vertical bar represents the total time spent in a single node during a single job writing to the device, computing on the device, and reading from the device. Each of those operations is then subdivided into time those operations spend queued, submitted, and running. We can draw several conclusions from this plot.

PageRank spends much more time communicating to and from the device than actually running on it. The large amount of time the Write and Read portions spend in the Submitted state suggest high contention for the PCIe bus, forcing copies to wait before being able to execute. However, little time is spent in the Submitted stage for the Compute portion of PageRank, indicating that the device is poorly utilized and generally available.

Genetic, on the other hand, demonstrates long Submitted periods for all portions of the job: Write, Compute, and Read. This suggests high contention for both the GPUs and PCIe bus being shared by multiple JVM threads. While the different scales for the left and right y-axes make it difficult to see, we also observe that Genetic spends significantly more time in the Run stage of the Compute portion, relative to PageRank: 483,677.065 ns for Genetic versus 1,492.259 ns for PageRank.

4.6 Hardware Utilization

In addition to studying the performance of our application in terms of time-to-complete certain software operations, we also study how hardware utilization differs between Spark and SWAT. In particular, we look at CPU utilization and system memory utilization.

Figure 10 shows the change in CPU and memory utilization for the PageRank benchmark. Figure 11 shows the same information for the Genetic benchmark. Table 3 lists peak utilization information for all benchmarks.

We observe that the results in Figure 10 and Table 3 support the conclusion that PageRank is not a compute-bound benchmark, only achieving a peak CPU utilization of 55% when running on Spark. Similarly, Figure 11 and Table 3 show that Genetic is compute-bound, achieving a peak CPU utilization of 90% when running on Spark.

Performing a comparative study between Spark and SWAT, we note that CPU utilization drops by an average of 31% across all benchmarks when using SWAT, but system memory utilization increases by an average of 25%. Both of these results are expected. It is natural for the host utilization to drop if the main compute workload is now offloaded to an accelerator. We also expect system memory utilization to

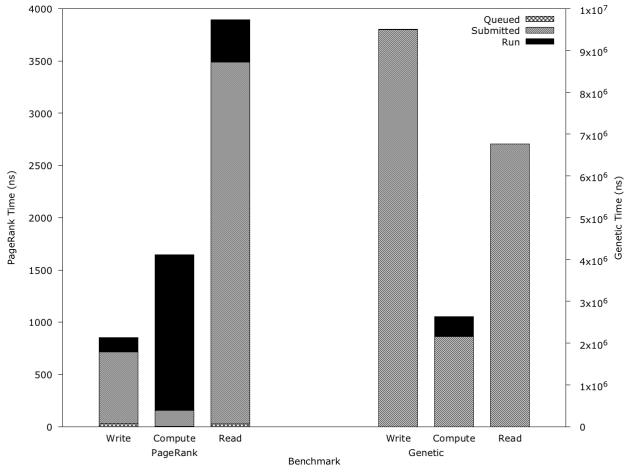

Figure 9: OpenCL event profiling information for PageRank and Genetic. Each vertical bar represents the total read, compute, or write time for a single benchmark. Each vertical bar is broken up into queued, submitted, and run times for that operation. PageRank is plotted against the left y-axis, Genetic is plotted against the right y-axis.

increase as SWAT allocates extra management data structures and host buffers for input and output accumulation.

Note that while there is an increase in system memory utilization with SWAT, the memory controls implemented as part of the SWAT Core are effective in keeping system memory utilization stable throughout the job: it does not oscillate or monotonically increase. In fact, it closely resembles the behavior of Spark's memory utilization, albeit with a constant factor added on top.

5. DISCUSSION AND CONCLUSIONS

In this work, we present work on an accelerated data analytics platform called SWAT that combines the distributed execution and high-level programming model of Apache Spark with the computational bandwidth of GPUs. Our approach uses code generation, an asynchronous & event driven runtime, dynamic GPU memory allocation inside & outside kernels, and a careful approach to resource management. Our performance evaluation in Section 4 shows that for compute-bound applications, our SWAT implementation achieves up to a 3.24× performance improvement, with no negative effects on scalability.

Some of the constraints we placed on this problem were important in shaping the solution. First, we wanted compatability with an existing and accepted framework out-of-the-box. That meant minimizing code change for existing applications and no access to framework-internal state. No existing work fulfills these constraints to our knowledge. Second, we wanted a minimalistic API that abstracted out all low-level details, leaving the runtime with more freedom to optimize. With a single method in our API and no custom data structures required, we satisfy this goal well. Third, we wanted to support the development of novel software applications and algorithms, requiring a code generation approach rather than the library-based approach that most related works have taken [8][14][17]. We see SWAT and

Benchmark	Peak CPU Utilization			Peak System Mem Utilization		
	Spark	SWAT	% Change	Spark	SWAT	% Change
PageRank	55%	50%	-9%	38%	49%	+27% (+12.96 GB)
Connected	28%	36%	-22%	81%	91%	+12% (+5.76 GB)
NN	89%	66%	-26%	77%	85%	+10% (+4.8 GB)
Fuzzy	92%	52%	-43%	30%	42%	+43% (+20.64 GB)
KMeans	85%	45%	-47%	37%	47%	+30% (+14.4 GB)
Genetic	90%	53%	-41%	37%	47%	+29% (+13.92 GB)
Average			**-31%**			**+25% (+12.00 GB)**

Table 3: Resource Utilization summary across all benchmarks

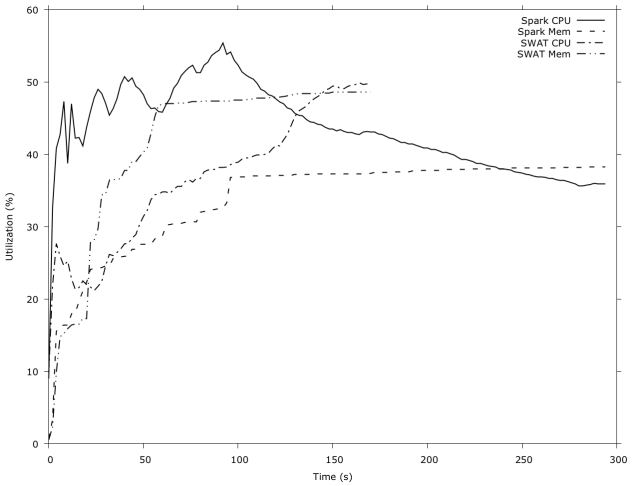

Figure 10: Host processor and memory utilization of the PageRank benchmark running on Spark and SWAT.

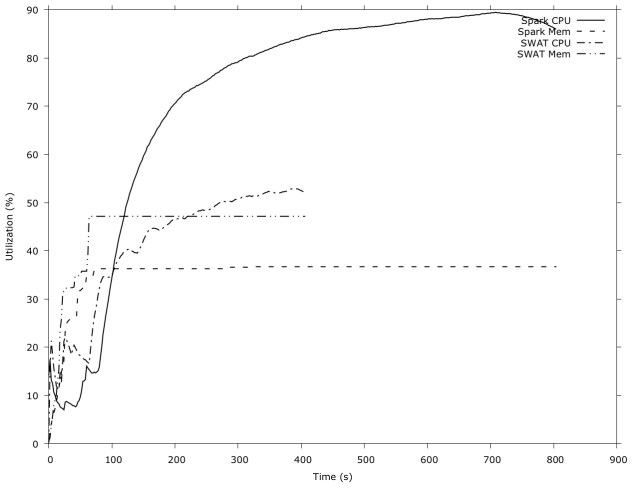

Figure 11: Host processor and memory utilization of the Genetic benchmark running on Spark and SWAT.

frameworks like it enabling domain experts to quickly develop novel and well-performing applications.

In our implementation, we emphasize tight resource management and minimize dynamic allocations by the framework on both the host and device to ensure that our framework is stable for long-running applications.

We also identify three hardware trends relevant to this work.

First, we see more accelerators packed into single nodes and the ratio of CPU cores to accelerators decreasing. With fewer threads sharing each accelerator, the device contention observed in Section 4.5 becomes less of a performance bottleneck.

Second, we also see higher performance interconnects between host and device, either in the form of higher bandwidth and lower latency buses (e.g. NVLink [19]) or system-on-a-chip solutions where the host and device physically share memory, like an AMD APU [3]. In more tightly coupled systems the data communication overheads for all heterogeneous applications are reduced, and we will see an increase in the domain of applications that can achieve performance improvement with SWAT. It is important to note that GPUs are not exclusively useful for compute-bound applications: in general, they also offer more memory bandwith than CPUs. The challenge today is that bandwidth-bound applications generally operate on large amounts of data, which must be transferred across a PCIe bus for the GPU to operate on. This leads to large communication overheads and nullifies any memory bandwidth benefits. Tightly coupled host-device systems reduce this problem.

Third, we see more researchers considering pairing power-efficient, lightweight host processors (e.g. Intel Atom, ARM) with GPUs to achieve high performance at low energy costs. In Section 4.6 we observed that using SWAT reduced CPU utilization by, on average, 31%. Most of our applications running on SWAT were only using ∼50% of the available CPU cycles. It is possible that running this framework on a platform with lightweight CPU cores instead would yield energy gains without any loss in performance.

We plan to extend this work in a number of ways. We will investigate load-balancing across the JVM and GPUs. As we mentioned above, many of our SWAT applications show ∼50% CPU utilization. It may be possible to do useful work in the JVM in parallel with GPU computation during those spare cycles.

We also will explore automatic device selection. In such a system, the framework would be responsible for determining that the JVM was the better execution platform for applications like PageRank and Connected, while the GPU should be used for NN, Fuzzy, KMeans, and Genetic. Past work

has explored using historical performance data combined with current device load to do online prediction of task performance in Hadoop [11]. More recent work has looked at estimating task performance offline based on static kernel features using an SVM [2]. We plan to combine the lessons learned from these works to perform more accurate performance prediction for Spark tasks.

As a stress test for the SWAT runtime and API, ongoing work also focuses on porting an existing large-scale application, CS-BWAMEM [26], to use SWAT. We will continue to improve the performance, stability, and flexiblity of the open source release.

SWAT combines the strengths of Spark and GPUs into a unified and programmable framework. There are many exciting future directions for this work, and we see its design and performance characteristics as being complemented by ongoing and future hardware trends. We also believe SWAT to be the first ADA constructed on Spark to be sufficiently flexible for real world application development. SWAT continues the recent work on accelerated data analytics platforms by emphasizing compatibility, careful resource management, runtime asynchrony, and programmability.

6. ACKNOWLEDGMENTS

This work was supported in part by the Data Analysis and Visualization Cyberinfrastructure funded by NSF under grant OCI-0959097 and Rice University.

7. REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org/.

[2] G. K. V. S. Akihiro Hayashi, Kazuaki Ishizaki. Machine-Learning-based Performance Heuristics for Runtime CPU/GPU Selection. In *12th International Conference on the Principles and Practice of Programming on the Java Platform, PPPJ*, 2015.

[3] AMD. AMD Accelerated Processing Units (APUs). http://www.amd.com/en-us/innovations/software-technologies/apu.

[4] Apache. Project Tungsten. https://issues.apache.org/jira/browse/SPARK-7075.

[5] Apache. Spark: Lightning-fast cluster computing.

[6] A. Asuncion and D. Newman. Uci machine learning repository, 2007.

[7] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Compiler Construction*, pages 244–263. Springer, 2010.

[8] J. Canny and H. Zhao. Bidmach: Large-scale learning with zero memory allocation. In *BigLearning, NIPS Workshop*, 2013.

[9] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[10] I. El-Helw, R. Hofman, and H. E. Bal. Glasswing: accelerating mapreduce on multi-core and many-core clusters. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 295–298. ACM, 2014.

[11] M. Grossman, M. Breternitz, and V. Sarkar. Hadoopcl2: Motivating the design of a distributed, heterogeneous programming system with machine-learning applications. In *IEEE Transactions on Parallel and Distributed Systems*.

[12] M. Grossman, S. Imam, and V. Sarkar. Hj-opencl: Reducing the gap between the jvm and accelerators. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, pages 2–15. ACM, 2015.

[13] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. Mapcg: writing parallel program portable between cpu and gpu. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 217–226. ACM, 2010.

[14] Y. Jia. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pages 675–678. ACM, 2014.

[15] Khronos Group. clGetEventProfilingInfo. https://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/clGetEventProfilingInfo.html.

[16] S. Lee, S.-J. Min, and R. Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, 44(4):101–110, 2009.

[17] 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. IEEE, 2015.

[18] H. Mühleisen and C. Bizer. Web data commons-extracting structured data from two large web corpora. *LDOW*, 937:133–145, 2012.

[19] NVIDIA. NVIDIA NVLINK HIGH-SPEED INTERCONNECT. http://www.nvidia.com/object/nvlink.html.

[20] Reynold Xin. Project Tungsten (Spark 1.5 Phase 1). https://issues.apache.org/jira/browse/SPARK-7075, 2015.

[21] A. Sabne. Heterodoop: A mapreduce programming system for accelerator clusters. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 235–246. ACM, 2015.

[22] O. Segal, P. Colangelo, N. Nasiri, Z. Qian, and M. Margala. Sparkcl: A unified programming framework for accelerators on heterogeneous clusters. *arXiv preprint arXiv:1505.01120*, 2015.

[23] 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.

[24] The Apache Software Foundation. Spark MLlib. http://spark.apache.org/mllib/.

[25] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for cuda. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54, 2013.

[26] Yu-Ting Chen. Cloud-Scale BWAMEM. https://github.com/ytchen0323/cloud-scale-bwamem.