Scaling MapReduce Vertically and Horizontally

Ismail El-Helw
Department of Computer Science
Vrije Universiteit Amsterdam
The Netherlands
ielhelw@cs.vu.nl

Rutger Hofman
Department of Computer Science
Vrije Universiteit Amsterdam
The Netherlands
rutger@cs.vu.nl

Henri E. Bal
Department of Computer Science
Vrije Universiteit Amsterdam
The Netherlands
bal@cs.vu.nl

***Abstract*—Glasswing is a MapReduce framework that uses OpenCL to exploit multi-core CPUs and accelerators. However, compute device capabilities may vary significantly and require targeted optimization. Similarly, the availability of resources such as memory, storage and interconnects can severely impact overall job performance. In this paper, we present and analyze how MapReduce applications can improve their horizontal and vertical scalability by using a well controlled mixture of coarse- and fine-grained parallelism. Specifically, we discuss the Glasswing pipeline and its ability to overlap computation, communication, memory transfers and disk access. Additionally, we show how Glasswing can adapt to the distinct capabilities of a variety of compute devices by employing fine-grained parallelism. We experimentally evaluated the performance of five MapReduce applications and show that Glasswing outperforms Hadoop on a 64-node multi-core CPU cluster by factors between 1.2 and 4, and factors from 20 to 30 on a 23-node GPU cluster. Similarly, we show that Glasswing is at least 1.5 times faster than GPMR on the GPU cluster.**

Keywords*—*MapReduce, Heterogeneous, OpenCL, Scalability

I. Introduction

The successful emergence of programmable GPUs and other forms of accelerators has radically changed the scientific computation domain by providing hundreds of gigaFLOPs performance at low cost. Additionally, the number of cores in general purpose processors is constantly increasing. Efficient use of this computational power remains a daunting task. For example, it is often difficult to approach the theoretical peak performance of the available compute resources for real applications. This situation is amplified further with the rising trend of heterogeneous cluster configurations that host high-end multi-core CPUs with dedicated GPUs within the same box. Such advancements in processor technology pose new challenges in the domain of high performance computing and reinforce the significance of parallel programming techniques [1].

In this paper, we investigate and analyze the feasibility of exploiting the computing capabilities of modern hardware to accelerate existing Big Data systems. To this end, we study MapReduce, a popular and successful programming model that is considered to be one of the cornerstones of the Big Data ecosystem. In particular, we argue that existing MapReduce systems were designed primarily for coarse-grained parallelism and therefore fail to exploit current multi-core and many-core technologies.

We present and analyze the performance of Glasswing [2], a novel MapReduce framework that aims to improve job performance through efficient utilization of resources in heterogeneous cluster environments. Glasswing sets itself apart from existing MapReduce implementations by using a mixture of coarse-grained and fine-grained parallelism. The use of coarse granularity allows it to scale horizontally by distributing workload across cluster nodes. Leveraging fine-grained parallelism control adds to Glasswing's ability to scale vertically. Therefore, Glasswing attains higher performance out of the same processors and hardware than existing MapReduce systems.

The core of Glasswing is a 5-stage pipeline that overlaps computation, communication between cluster nodes, memory transfers to compute devices, and disk access in a coarse-grained manner. The pipeline is a key contributor to Glasswing's superior performance as it enables Glasswing to manage multiple data sets in different processing stages concurrently. Therefore, a job's execution time is improved and is only limited by the most dominant stage of the pipeline.

In addition to employing coarse-grained parallelism, Glasswing's design stresses the significance of vertical scalability. As a separate architectural contribution, Glasswing uses fine-grained parallelism within each node to target modern multi-core and many-core processors. It exploits OpenCL to execute tasks on different types of compute devices without sacrificing the MapReduce abstraction [2]. Additionally, it is capable of controlling task granularity to adapt to the diverse needs of each distinct compute device.

Thus, Glasswing enables applications to run on accelerators like GPUs and Intel's Xeon Phi to exploit their unique computation capabilities. Moreover, it does so using the same software abstraction and API. This is particularly beneficial to prevalent use cases such as machine learning applications which not only process large data sets but also are computationally expensive [3], [4]. For instance, common machine learning algorithms such as classification, clustering and generating user recommendations are often formulated as linear algebra problems composed of matrix and vector operations. These types of numerical calculations are excellent candidates for acceleration with GPUs [5], [6].

To use Glasswing, a programmer has to write map/reduce functions in OpenCL and possibly optimize them per compute device since OpenCL does not guarantee performance portability. However, our experience with Glasswing indicates that device-specific optimizations tend to be simple within the context of the MapReduce model. For instance, the predominant performance variable is the number of threads and the allocation of work over the threads. These variables are often the only parameters necessary to tune the application.

SC14, November 16-21, 2014, New Orleans, Louisiana, USA
978-1-4799-5500-8/14/$31.00

As opposed to existing MapReduce systems, Glasswing is structured in the form of a light-weight software library. Therefore, it does not require dedicated deployments, which facilitates its use in clusters and cloud computing environments. To verify our contributions, we implemented a number of applications using Glasswing and compared its performance to Hadoop [7] and GPMR [8] on a 64-node cluster. Experimental results show that Glasswing attains higher performance and scalability.

The contributions of this work are:

- showing that MapReduce can exploit overlap of computation with communication, memory transfers and disk access using a 5-stage pipeline;
- showing how performance is boosted by fine-grain parallelism using a variety of compute devices, in an evaluation of Glasswing's vertical and horizontal scalability using various cluster configurations;
- performance improvement over Hadoop on a 64-node cluster up to factors of 30 and 4.1 with and without GPU acceleration respectively; performance improvements ranging from 1.5x to 10x compared to GPMR.

The remainder of this paper is organized as follows. Section 2 presents more background and reviews related work. A presentation of the system design is given in Section 3. Section 4 evaluates the performance of Glasswing and discusses its key contributions. Finally, Section 5 concludes.

II. Background and Related Work

MapReduce is a programming model designed to cater for large data center computations and data mining applications [9]. It relieves developers from managing the complexity of concurrency, communication and fault tolerance. The map function processes an input dataset and produces intermediate key/value pairs. The reduce function processes each key and its associated list of values to produce a final dataset of key/value pairs. There are no restrictions on the order in which the map input dataset is processed therefore input splits can be processed concurrently by multiple mappers. Reduction parallelism is limited by the number of intermediate keys.

Below, we classify the most relevant existing work into three categories: optimizations and performance improvements applied to Hadoop; implementations that specifically target multi-core CPUs or GPUs; and systems exploiting other accelerators.

Hadoop Optimizations: Hadoop, Apache's open-source implementation of MapReduce [7], has been gaining massive momentum and widespread adoption since its creation. Naturally, a multitude of studies were performed to analyze and improve its performance. Some focus was given to the MapReduce data placement policy [10], [11] which builds on the principle of data locality. Hadoop-A [12] improved Hadoop's intermediate data shuffle algorithm with faster data transfers using native RDMA. DynMR [13] focuses on improving the performance of the TeraSort benchmark by interleaving reduce tasks and backfilling mappers in the Hadoop-compatible IBM Platform Symphony. Other projects looked into enhancing the Hadoop task scheduler algorithms in heterogeneous cloud environments [14] and improving fairness in multi-user clusters [15]. Shirahata et al. [16] improved the scheduling of compute-intensive tasks in clusters where some, but not all, nodes have GPUs.

TABLE I. Comparison between Glasswing and related projects.

	Out of Core	Compute Device	Cluster
Phoenix	×	CPU-only	×
Tiled-MapReduce	×	NUMA CPU	×
Mars	×	GPU-only	×
Ji et al. [17]	×	GPU-only	×
MapCG	×	CPU/GPU	×
Chen et al. [18]	×	GPU-only	×
GPMR	×	GPU-only	✓
Chen et al. [19]	×	AMD Fusion	×
Merge	×	Any	×
HadoopCL	✓	APARAPI	✓
Glasswing	✓	OpenCL enabled	✓

Runtime Systems For Multi-cores and GPUs: Multiple studies looked into using multi-core CPUs or GPUs to improve performance of MapReduce jobs. Table I displays the key features and differences between Glasswing and these efforts. Below, we discuss them in more detail.

Multiple studies investigated the benefits of running MapReduce on multi-core processors. Phoenix [20], [21] is an implementation of MapReduce for symmetric multi-core systems. It manages task scheduling across cores within a single machine. Tiled-MapReduce [22] improves on Phoenix by adding a task scheduler that is aware of Non-Uniform Memory Access (NUMA) architectures. Both systems use only a single node and do not exploit GPUs.

The adoption of MapReduce by compute-intensive scientific applications introduced several studies that utilize GPUs. Mars [23] is a MapReduce framework that runs exclusively on an NVidia GT80 GPU. Due to the lack of dynamic memory management and atomic operation support in the GT80, Mars performs all map and reduce executions twice. Ji et al. [17] presented an implementation that improves over Mars by enhancing its use of the memory hierarchy. MapCG [24] is a MapReduce framework that uses CUDA and pthreads to target GPUs and CPUs. In contrast to Mars, MapCG focuses on code portability whereby the same application can run on the CPU or GPU. Chen et al. [18] presented a framework that focuses on efficiently utilizing the GPU memory hierarchy for applications that perform aggregation operations in their reduce function. These studies share several limitations. Most important, they require the input, intermediate and output data to coexist in GPU memory which severely limits the problem sizes they can tackle, as testified by their reported dataset sizes. Further, they run on a single node only. In contrast, Glasswing was designed to be scalable and tackle massive out-of-core dataset sizes, and it runs over a cluster of machines. Additionally, it provides compute device flexibility whereby map and reduce tasks can be executed on CPUs or GPUs.

Mithra [25] presents the feasibility of utilizing GPUs for compute-intensive Monte Carlo simulations, using Hadoop for its job management. It implements the Black Scholes option pricing model in CUDA as a sample benchmark.

GPMR [8] is a cluster implementation of MapReduce that uses CUDA to execute tasks on NVidia GPUs. GPMR runs on GPUs only which makes it unsuitable for many applications.

Moreover, it is limited to processing data sets where intermediate data fits in host memory, whereas Glasswing handles out-of-core intermediate data (as well as out-of-core input and final data). HadoopCL [26] explores the potential of running OpenCL compute kernels within the Hadoop framework. It uses APARAPI [27] to translate Java-implemented map and reduce functions to OpenCL kernels. However, HadoopCL inherits data representation limitations from its use of APARAPI. For instance, the map/reduce functions cannot use Java objects and are limited to arrays of primitive types. Furthermore, breaking out of a simple loop or condition statement is prohibited.

Heterogeneous Systems and Accelerators: MapReduce implementations exist for FPGAs [28] and Cell Broadband Engine [29], [30]. Chen et al. [19] developed an in-core MapReduce framework targeting the AMD Fusion architecture which integrates a CPU and a GPU on the same die. Merge [31] is a parallel programming framework designed for heterogeneous multi-core systems. It provides a runtime similar to MapReduce which schedules tasks over a variety of processors. Merge relies on the developer to provide several implementations of performance-critical functions utilizing processor-specific intrinsics. This approach increases the complexity of managing code for different architectures. Merge runs on a single heterogeneous node which limits its scalability.

OpenCL: OpenCL is an open parallel programming standard that provides a unified model for programming modern processors. Applications employing OpenCL are split into two distinct code bases: compute kernels and host code. Compute kernels are usually the computationally intensive code segments and are highly optimized. The host code is responsible for performing all tasks that lead to the execution of the kernels such as managing synchronization and memory transfers to and from compute devices. Glasswing requires map and reduce functions to be implemented in OpenCL.

III. System Design

Glasswing, being a MapReduce system, has three main phases: the map phase, the shuffle phase whose equivalent is called merge phase in Glasswing, and the reduce phase. The map phase and the reduce phase are both implemented with an instantiation of the *Glasswing pipeline*. The Glasswing pipeline is the principal component of the framework. Its instantiations run independently on each node participating in a computation. One of the key contributions of the Glasswing pipeline is its ability to overlap communication, memory transfers and computation. To that end, it is capable of interacting with diverse compute device architectures using OpenCL to manage their memory and state. Moreover, stages of the Glasswing pipeline use fine-grain parallelism wherever feasible, either using OpenCL or host-level threads. The merge phase exchanges and sorts data and synchronizes between nodes, again speeded up by latency hiding and host thread parallelism.

Execution starts with launching the map phase and, concurrently, the merge phase at each node. After the map phase completes, the merge phase continues until it has received all data sent to it by map pipeline instantiations at other nodes. After the merge phase completes, the reduce phase is started.

Below, the key design concepts of the framework are discussed in more detail: the map and reduce pipelines, intermediate data management, and multiple buffering. We also discuss the interaction between the framework and the user application.

A. Map Pipeline

The high-level purpose of the map pipeline is to process input data and transform it into intermediate key/value pairs. To achieve this task efficiently, the map pipeline divides its work into five distinct operations that can be carried out concurrently. The top row of Figure 1 depicts the task division of the map pipeline.

The **first stage** of the pipeline reads input files, splits them into records, and loads them into main memory. Upon completion of this task, it notifies the next stage of the pipeline of pending work.

The **second stage** of the pipeline guarantees the delivery of the loaded input to the selected compute device. The process of staging input varies depending on the architecture of the selected compute device. For example, GPUs have memory that may not be accessible, or accessible efficiently, from the host, so the data must be copied to GPU memory. On the other hand, CPUs and GPUs with unified memory capabilities can access the loaded input directly from host RAM. In this case, the input stager is disabled. Once staging is complete, the next pipeline item is notified of pending work.

Execution of the map functions over the staged input split is performed by the **third stage**. This pipeline item is responsible for scheduling a number of OpenCL threads to process the staged input split. The Glasswing OpenCL middleware exports kernel entry points for the OpenCL threads. These compute kernels divide the available number of records between them and invoke the application-specific map function on each record. In contrast to existing systems that process each split sequentially, Glasswing processes each split in parallel, exploiting the abundance of cores in modern compute devices. This design decision places less stress on the file system that provides the input data since the pipeline reads one input split at a time. Once the compute kernel completes its execution, it notifies the following pipeline stage.

The **fourth stage** collects the produced key/value pairs and delivers them to host memory. Like the second stage, this stage is disabled if the output buffer uses unified memory.

The **final stage** of the map pipeline is responsible for sorting the gathered intermediate key/value pairs and partitioning them across the cluster nodes. Glasswing partitions intermediate data based on a hash function which can be overloaded by the user. Thus, intermediate data generated from each input split is further divided into a number of *Partitions*. All generated *Partitions* are written to local disk to prevent data loss. Subsequently, each *Partition* is pushed to a cluster node based on a mapping of *Partition* identifiers to nodes. If a *Partition* is destined for the local node, it is simply added to the node's intermediate data in-memory cache. Otherwise, the *Partition* is sent over the network to the appropriate cluster node. Due to the large intermediate data volume of some MapReduce jobs, the partitioning stage of the pipeline may

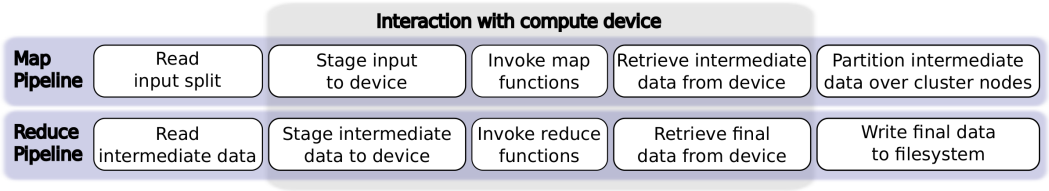

Fig. 1. The 5-stage map and reduce pipelines.

account for a significant portion of the run time. Therefore, the partitioning stage can be configured to use multiple host threads to speedup the processing of intermediate data.

B. Intermediate Data Management

In parallel to the map pipeline, each cluster node runs an independent group of threads to manage intermediate data. The management of intermediate data is divided into three key components. First, each node maintains an in-memory cache of *Partitions* which are merged and flushed to disk when their aggregate size exceeds a configurable threshold. Second, intermediate data *Partitions* produced by other cluster nodes are received and added to the in-memory cache. Finally, *Partitions* residing on disk are continuously merged using multi-way merging so the number of intermediate data files is limited to a configurable count. Glasswing can be configured to use multiple threads to speed-up both the merge and flush operations. All intermediate data *Partitions* residing in the cache or disk are stored in a serialized and compressed form.

The process of merging intermediate data directly affects the performance of a job since the reduce phase of the computation awaits its completion. Hence, we identified the **merge delay** as a performance metric related to intermediate data management. We define the **merge delay** as the time dedicated to merging intermediate data after the completion of the map phase and before reduction starts.

C. Reduce Pipeline

Similar to the map pipeline, the reduce pipeline is composed of 5 distinct stages as depicted in the bottom row of Figure 1. The first pipeline element is responsible for loading intermediate data to main memory. However, as opposed to the map input, intermediate data may reside in multiple *Partitions*. Therefore, the reduce input reader performs one last merge operation and supplies the pipeline with a consistent view of the intermediate data.

The three middle phases of the reduce pipeline are synonymous to those of the map pipeline. Hence, they are responsible for delivering the reduce input to the compute device, applying the application-specifc reduction operation, and obtaining the final output. Finally, the last stage of the reduce pipeline writes the output data to persistent storage.

The classical MapReduce model differs in the nature of parallelism in the map and reduce phases. Parallel execution of mappers is straightforward: they operate on independent data. However, reduce parallelism is limited by the number of keys since reduce functions process the values of a key sequentially. Glasswing provides two different forms of fine-grain parallelism to accelerate reduction. First, applications can choose to process each single key with multiple threads. This is advantageous to compute-intensive applications that can benefit from parallel reduction. Second, the reduce pipeline of a single node is capable of processing multiple keys concurrently. This feature enables the use of all cores of the compute device. For applications with little value data per key, this may lead to little work per kernel thread. To alleviate thread creation overhead, Glasswing provides the possibility to have each reduce kernel thread process multiple keys sequentially.

If the number of values to be reduced for one key is too large for one kernel invocation, some state must be saved across kernel calls. Glasswing provides scratch buffers for each key to store such state.

D. Pipeline Buffering Levels

As data buffers are concerned, the Glasswing pipeline falls into two groups: the input group, with the stages that process input buffers (Input, Stage, Kernel), and the output group, with the stages that process output buffers (Kernel, Retrieve and Output). This allows the pipeline to operate in single, double, or triple buffering mode. For example, with single buffering, the three stages in the input group execute sequentially as they are interlocked, as do the stages in the output group. However, the input group still operates in parallel with the output group, as they do not share data buffers. When the pipeline is configured with higher buffering, the interlocking window within the input and output groups is relaxed, allowing the stages within each group to run concurrently. The limit is reached for triple buffering, which allows the pipeline to run its five stages in full concurrency. Double or triple buffering comes at the cost of more buffers, which may be a limited resource for GPUs, or of more contention for CPU cores. Where the balance in this trade-off lies depends on the application, which must be tuned to find the best fit.

E. Fault Tolerance

Fault tolerance of the MapReduce model is a well-studied problem. Since Glasswing is a single-tenant system, it is relieved from managing the complexity of monitoring and scheduling multiple concurrent jobs. Glasswing provides data durability guarantees for its applications. For instance, the output of all map operations produced by a cluster node is stored persistently on disk in addition to a second copy that goes into the processing of intermediate data. However, Glasswing currently does not handle task failure. The standard approach of managing MapReduce task failure is re-execution: if a task fails, its partial output is discarded and its input is rescheduled for processing. Addition of this functionality would consist of bookkeeping only which would involve negligible overhead.

F. User Application APIs

Glasswing provides two sets of APIs for developers to implement their applications and integrate them with the framework. The **Configuration API** allows developers to specify key job parameters. For instance, an application can specify the input files to be processed as well as their formats. Furthermore, applications can specify which compute devices are to be used and configure the pipeline buffering levels. The Glasswing **OpenCL API** provides utilities for the user's OpenCL map/reduce functions that process the data. This API strictly follows the MapReduce model: the user functions consume input and emit output in the form of key/value pairs. Glasswing implements two mechanisms for collecting and storing such output. The first mechanism uses a shared buffer pool to store all output data. The second mechanism provides a hash table implementation to store the key/value pairs. Glasswing provides support for an application-specific combiner stage (a local reduce over the results of one map chunk) only for the second mechanism. Applications can select the output collection mechanism using the Configuration API. The Glasswing APIs are thoroughly discussed in [2].

IV. Evaluation

The following subsections describe the performance evaluation of Glasswing. We experimentally evaluate a number of aspects: 1) horizontal scalability, where overall system performance on clusters is compared and analysed for a number of open-source MapReduce systems, and where the potential benefits of GPUs are considered; 2) in-depth analysis of Glasswing's optimizations for the three main phases: map, merge and reduce; 3) vertical scalability, where Glasswing performance with different accelerators is considered.

The experiments were conducted on the DAS4 cluster at VU Amsterdam. The cluster nodes run CentOS Linux and are connected via Gigabit Ethernet and QDR InfiniBand. There are two types of nodes in the cluster. The 68 *Type-1* nodes have a dual quad-core Intel Xeon 2.4GHz CPU, 24GB of memory and two 1TB disks configured with software RAID0. The *Type-2* nodes have a dual 6-core Xeon 2GHz CPU and 64GB memory. The nodes have hyperthreading enabled, so in many cases the nodes can run 16 (for *Type-1*) processes or 24 (for *Type-2*) fully in parallel. 23 of the *Type-1* nodes have an NVidia GTX480 GPU. 8 *Type-2* nodes have an NVidia K20m GPU. Two more *Type-2* node are equipped with an Intel Xeon Phi and one more node has an NVidia GTX680 GPU.

For the Glasswing experiments, AMD's OpenCL SDK 2.7 was used for the CPUs, and NVidia's OpenCL SDK that comes bundled with CUDA 5.0 for the GPUs. The Xeon Phi is used with Intel's OpenCL SDK 3.0 and device driver with Xeon Phi(MIC) support. We used Hadoop stable version 1.0.0.

To fairly represent the wide spectrum of MapReduce applications we implemented and analyzed five applications with diverse properties. Each application represents a different combination of compute intensity, input/output patterns, intermediate data volume and key space.

A. Horizontal Scalability

To evaluate the scalability and performance of Glasswing, we conducted measurements on the cluster of *Type-1* nodes with a master and 1 up to 64 slaves. For comparison, we also implemented our applications in Hadoop using the same algorithms, and for the compute-intensive applications we did a GPU comparison against GPMR [8][1].

Hadoop was selected for comparison because it is a de-facto standard and capable of managing large data sets, in contrast to the other systems in Section II. Wherever possible, the Hadoop applications use Hadoop's SequenceFile API to efficiently serialize input and output without the need for text formatting. Hadoop was configured to disable redundant speculative computation, since the DAS4 cluster is extremely stable. We performed a parameter sweep on the cluster to determine the optimal number of mappers and reducers for each Hadoop application; consequently all cores of all nodes are occupied maximally. We ensured that the Hadoop executions are well load-balanced and suffer no harmful job restarts. For the comparison against Hadoop, we instrumented Glasswing to use HDFS so it has no file access time advantage over Hadoop. We used Hadoop's bundled library libhdfs that employs JNI to access HDFS's Java API. HDFS was deployed on all nodes participating in a computation and connected using IP over InfiniBand. Unless stated otherwise, HDFS files use a replication factor of 3 which is common practice. Glasswing's job coordinator is like Hadoop's: both use a dedicated master node; Glasswing's scheduler considers file affinity in its job allocation. We verified the output of Glasswing and Hadoop applications to be identical and correct. Before each test, the filesystem cache was purged to guarantee test consistency.

GPMR was selected for comparison because it supports clusters, it is capable of using GPUs, and its source code is publicly available. The data layout for this comparison follows the experimental setup reported by GPMR [8]: all files are fully replicated on the local file system of each node. The network is again IP over Infiniband. Those of our applications that are fit for GPU acceleration, the compute-intensive ones, also came bundled with the GPMR release. We will highlight the relevant differences in the performance discussions below.

In this section, *speedup* is used in the sense of execution time of one slave node over the execution time of *n* slave nodes *of the same framework*. In the graphics, execution time and speedup are presented in one figure. The falling lines (upper left to lower right) are execution times, with time at the left-hand Y-axis; the rising lines (lower left to upper right) are speedups, with figures at the right-hand Y-axis. The number of nodes and speedup (right-hand Y-axis) are presented in logscale to improve readability.

1) I/O-bound applications: Figure 2 presents the experimental results of three I/O-bound applications, Pageview Count (*PVC*), Wordcount (*WC*), and TeraSort (*TS*) in Hadoop and Glasswing. Since these applications are I/O-bound, they do not run efficiently on GPUs so we do not report results for Glasswing on the GPU or GPMR.

PVC processes the logs of web servers and counts the frequency of URL occurrences. It is an I/O-bound application as its kernels perform little work per input record. Its input data are web server logs of wikipedia that are made available

[1]We would have liked to include HadoopCL [26] in our evaluation as it is highly relevant work, but its authors indicated that it is not yet open-sourced.

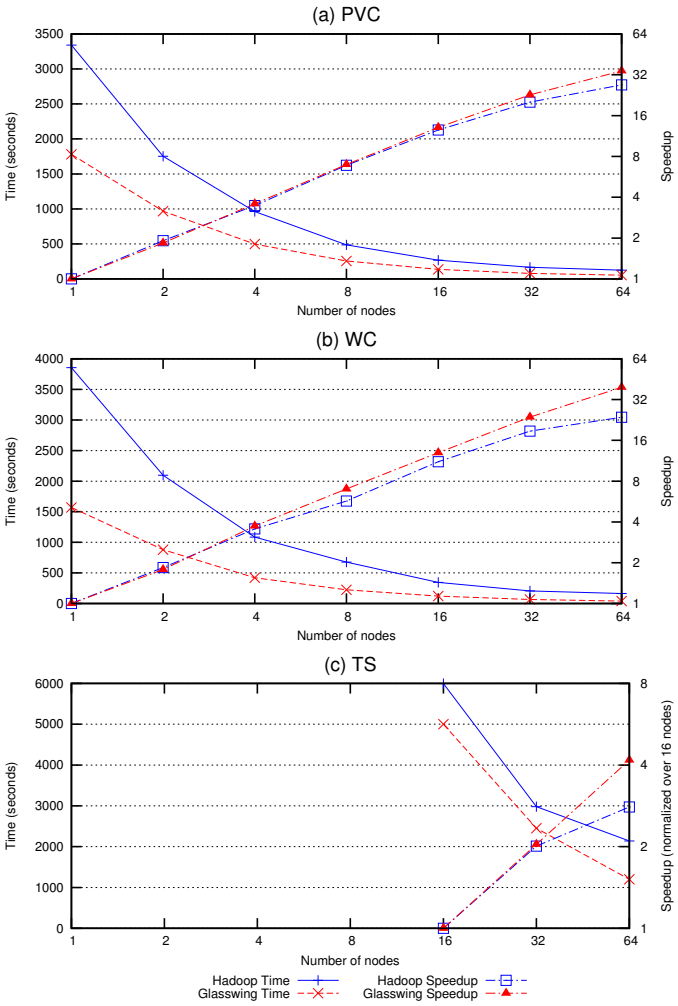

Fig. 2. Performance of I/O-bound applications.

by the WikiBench project [32]. We used 143GB of log traces from the 2007-19 data set. The logs are highly sparse in that duplicate URLs are rare, so the volume of intermediate data is large, with a massive number of keys. Figure 2(a) shows that the speedup of Glasswing and Hadoop is very comparable, with Glasswing scaling only slightly better for a large number of nodes. However, in execution time Glasswing is nearly twice as fast as Hadoop. We attribute this to a combination of factors. First, Glasswing uses pipeline parallelism to overlap I/O and computation. Second, Glasswing uses fine-grained parallelism in a number of its stages, e.g. intermediate data partitioning and key handling in the reduce phase. Third, Glasswing has a latency advantage because it *pushes* its intermediate data to the reducer node, whereas Hadoop *pulls* its intermediate data.

WC counts the frequency of word occurrences in a group of input files. *WC* is commonly used in data mining. As our input set for *WC*, we used the publicly available English wikipedia dump [33], which is composed of wiki markup pages and embedded XML metadata. We used 87GB of version 20130102. This data set is irregular, in that it exhibits high repetition of a smaller number of words beside a large number of sparse words. As is shown in Figure 2(b), Glasswing performs 2.46 times faster sequentially than Hadoop, and its scaling is better: 62% parallel efficiency on 64 nodes compared to 37%. *WC* shows better parallel scalability than *PVC*, because the *WC* kernel performs somewhat more computation than the *PVC* kernel. This enlarges the benefits of Glasswing's pipeline overlap between computation and communication.

The **TeraSort** (***TS***) benchmark is one of the most data-intensive MapReduce applications and aims to sort a set of randomly generated 10-byte keys accompanied with 90-byte values. However, TS requires the output of the job to be totally ordered across all partitions. Hence, all keys occurring in output partition *N-1* must be smaller than or equal to keys in partition *N* for all *N* partitions. In order to guarantee total order of the job's output, the input data set is sampled in an attempt to estimate the spread of keys. Consequently, the job's map function uses the sampled data to place each key in the appropriate output partition. Furthermore, each partition of keys is sorted independently by the framework resulting in the final output. TS does not require a reduce function since its output is fully processed by the end of the intermediate data shuffle. We tested TS with an input data set size of 2TB. This input configuration is particularly interesting as each of its input, intermediate and output data sizes cannot fit in the aggregate memory of the 64 cluster nodes. Hence, TS is an effective benchmark to validate Glasswing's contributions. As opposed to our other applications, the replication factor of the output of TS is set to 1. Due to the large input dataset, runs on smaller numbers of machines were infeasible because of lack of free disk space. The standard Hadoop implementation was used both to generate the input data and to run the Hadoop experiments. Figure 2(c) shows that Glasswing outperforms Hadoop on 64 nodes by a factor of 1.78, due to the same reasons stated above.

2) Compute-bound applications: Figure 3 presents performance of two compute-bound applications, K-Means Clustering (*KM*) and Matrix Multiply (*MM*). It shows the performance numbers of Hadoop and Glasswing using the CPU as the compute device. Additionally, it compares between GPMR and Glasswing with GPU acceleration.

KM partitions observations (vector points) in a multi-dimensional vector space, by grouping close-by points together. *KM* is a compute-intensive application and its complexity is a function of the number of dimensions, centers and observations. *KM* is an iterative algorithm, but our implementations perform just one iteration since this shows the performance well for all frameworks. The Hadoop implementation of *KM* uses the DistributedCache API to distribute the centers onto all cluster nodes to avoid remote file reads. To evaluate *KM*, we randomly generated 8192 centers and 1024^3 points of single-precision floats in 4 dimensions. With 8192 centers, the I/O time for all platforms and file systems is negligible compared to the computation time. Figure 3(a) presents the performance on the CPU. Glasswing is superior to Hadoop, comparable to the performance gains of the I/O-bound applications.

Figure 3(c) shows that *KM* profits from GPU acceleration: the single-node Glasswing run time is 426s, a gain of 28x over Hadoop. This is in line with the greater compute power of the GPU and *KM*'s abundant data parallelism. The GPMR paper [8] states that its *KM* implementation is optimized for a small number of centers and is not expected to run efficiently for larger numbers of centers. Therefore, to ensure comparison fairness we present its performance for two different configu-

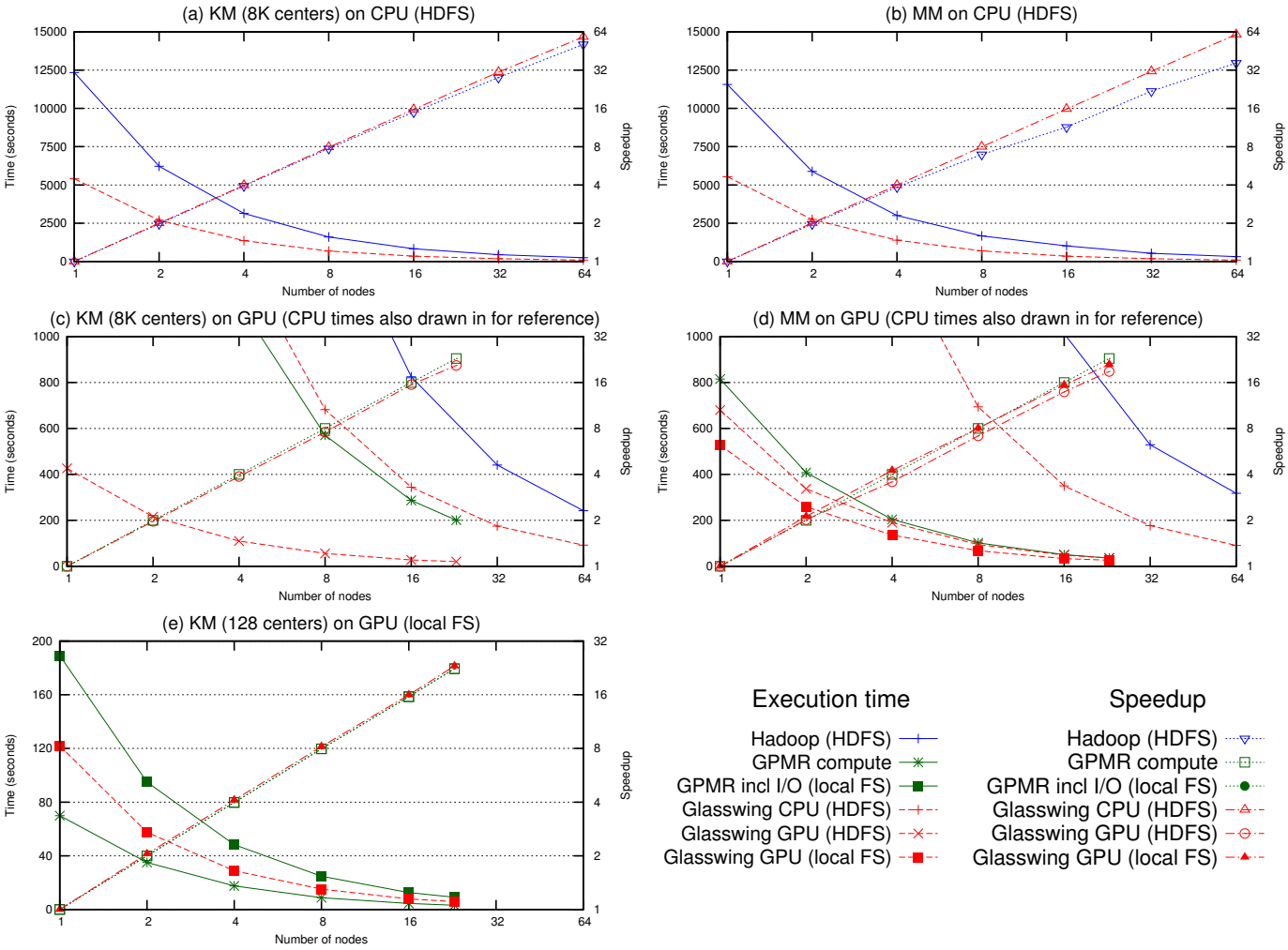

Fig. 3. Performance of compute-bound applications.

rations. First, we applied two small adaptations to the GPMR application code to be able to use more than 128 centers. Figure 3(c) shows that the GPMR implementation indeed is inefficient for 8192 centers. Second, Figure 3(e) presents a separate comparison using the unmodified GPMR code with 128 centers, with again 1024^3 points. With this few centers, *KM* becomes I/O-dominant: reading the data from the nodes' local disks takes twice as long as the computation. Instrumentation showed that the *computation* time of Glasswing's generic code equals GPMR's optimized code. Glasswing overlaps computation and I/O so its total time is approximately the maximum of computation and I/O. On the other hand, GPMR first reads all data, then starts its computation pipeline; its total time is the sum of computation and I/O. Figure 3(e) shows separate lines for GPMR's computation time (lower line) and total time including I/O (upper line). GPMR's total time is about 1.5x Glasswing's for all cluster sizes.

Our implementation of ***MM*** multiplies two square matrices A and B by tiling them into multiple sub-matrices. Each sub-matrix is identified by the coordinate of its top left row and column. To test *MM*, we generated two square matrices of size 32768^2. We used OpenCL's work-item information to create two custom workload divisions. The first scheme targets GPU architectures by dividing the threads into multiple groups, each responsible for computing a tile in the result sub-matrix. The second scheme is more suitable for generic CPU architectures where each thread assumes the responsibility of computing a whole tile in the result sub-matrix. It is used by the Hadoop implementation and the Glasswing CPU implementation. Figure 3(b) shows *MM* on the CPU, where the performance gains over Hadoop are confirmed.

As Figure 3(d) shows, *MM* benefits from GPU acceleration. In contrast to *KM*, *MM* consumes a large volume of data which limits the performance acceleration provided by the GPU. Instrumentation shows that *MM* is I/O-bound on the GPU when combined with HDFS usage, unlike its compute-bound behavior on the CPU. The execution times on the local FS, represented by the lower line for Glasswing, illustrate how performance is influenced by HDFS, even though HDFS replication ensures that almost all file accesses are local. HDFS comes with considerable overhead, the most important source being Java/native switches and data transfers through JNI.

The GPMR *MM* implementation performs a map kernel to compute intermediate submatrices but does not aggregate the partial submatrices as it has no reduce implementation. As a

TABLE II. *WC* MAP PIPELINE TIME BREAKDOWN IN SECONDS.

	double buffering			single
Hash table	✓	✓	×	✓
Combiner	✓	×	×	✓
Input	127	126	105	107
Kernel	215	226	78	201
Partitioning	158	182	997	65
Map elapsed time	216	227	998	308
Merge delay	2.1	4.8	4.8	2.7
Reduce time	36	56	57	46

result, it does not store or transfer intermediate data between nodes. Moreover, GPMR does not read its input matrices from files, but generates them on the fly and excludes the generation time from its performance numbers. Nevertheless, GPMR's *MM* is outperformed by the Glasswing GPU implementation. We attribute that to the fact that the Glasswing GPU kernel is more carefully performance-engineered.

To verify whether the conclusions from the experiments on the *Type-1* cluster of GTX480 are also valid on other GPUs, we ran Glasswing *KM* and *MM* on 1 upto 8 *Type-2* nodes equipped with a K20m and obtained consistent scaling results.

This evaluation of the horizontal scalability allows to draw some important conclusions. First, Glasswing is capable of using CPU, memory and disk resources more efficiently than Hadoop, leading to a single-node performance improvement factor of at least 1.88x. Second, compute-bound applications benefit from GPU acceleration; the single-node improvement is a factor 28 for *KM*. Third, Glasswing exhibits better horizontal scalability than Hadoop. For instance, the factor by which Glasswing's *WC* outperforms Hadoop grows from 2.46x on a single node to 4.1x on 64 nodes, and for *TS* the performance gap grows from 1.2x on 16 nodes to 1.78x on 64 nodes. Finally, in addition to its full scalability support, Glasswing is faster than GPMR, again because it better exploits pipelining.

B. Evaluation of Pipeline Optimizations

We present a performance analysis of the three main constituents of Glasswing: 1) the map pipeline; 2) handling of intermediate data, including the merge phase that continues after the map phase has completed; and 3) the reduce pipeline. To investigate the capabilities of the Glasswing pipeline we instrumented it with timers for each pipeline stage. Below, we present detailed analyses of *WC* and *KM* as representatives of I/O-bound and compute-intensive applications respectively. The pipeline analysis was performed on one *Type-1* node without HDFS. Smaller data sets were used to emphasize the performance differences.

1) Map Pipeline: WordCount: In these two subsections, we investigate the performance effects of three optimizations in the map pipeline: the hash table, used to harvest map kernel output; the effect of a combiner kernel; and the benefits of multiple buffering. Table II presents the time breakdown of *WC*'s map pipeline stages. The first three columns present three different configurations:
(i) using the hash table and a combiner function,
(ii) using the hash table without a combiner, and
(iii) using the simple output collection without a combiner.
The rightmost column has configuration *(i)* but differs in that single buffering is used in stead of double buffering. For Glasswing CPU, the OpenCL buffers use unified host/device memory, so the Stage and Retrieve stages are disabled, and there is no triple buffering configuration here. Note that the Kernel time exceeds the Input time: in the experiments in this section, the data is read from the local file system, whereas in the horizontal scalability experiments in Section IV-A, the data was read through HDFS, which is more expensive. On HDFS, Input is the dominant stage by a small margin.

The use of the hash table in **configuration *(i)*** allows *WC* to store the contents of each key once. The combine function further reduces the size of intermediate data by aggregating values for each key. Since *WC* is data-intensive, it spends a considerable amount of time reading input and partitioning intermediate data. However, the total elapsed time is very close to the kernel execution time, which is the dominant pipeline stage; the pipeline overlaps its activity with the other pipeline stages. For example, the sum of the time spent in the stages is 498s while the elapsed time is 215s.

Configuration *(ii)* shows a slightly increased kernel execution time. Even though the combine kernel is not executed, Glasswing executes a compacting kernel after map() to place values of the same key in contiguous memory. This process relieves the pipeline from copying and decoding the whole hash table memory space. Moreover, there is a significant rise in the time spent partitioning intermediate data. This is caused by the increase in intermediate data due to the lack of a combiner. The increased number of key occurrences is also reflected in a higher time necessary for the merger threads to finish, and a higher reduce time.

Configuration *(iii)* presents completely different behavior: using the simple output collection instead of the hash table lowers the kernel execution time significantly. To understand the performance difference, the application nature and workload must be considered. *WC* exhibits a high repetition of a number of keys which increases the contention on the hash table, so threads must loop multiple times before they allocate space for their new value. However, when simple output collection is used, each thread allocates space via a single atomic operation. The improved kernel execution time comes at the cost of increased intermediate data volume and partitioning overhead. Since the partitioning stage has to decode each key/value occurrence individually, its time vastly exceeds the kernel execution and becomes the dominant stage of the pipeline, thus increasing elapsed time significantly. The number of keys and values in intermediate data are the same as in configuration *(ii)*, which is reflected in the times for merge delay and reduce. The partitioning stage can be accelerated by increasing its parallelism; with 32 threads, the partitioning stage still takes 851s. In that case, the merger threads are starved from CPU core usage during the map phase, which causes the merge delay to increase to 26s.

While the three lefthand columns of numbers in Table II show double buffering mode, the right-hand column shows the performance for **single buffering mode**. In this case, the pipeline stages of the input group are serialized, as are the pipeline stages of the output group. The map elapsed time equals the sum of the input stage and the kernel stage. Partitioning is faster because there is less contention for the CPU cores.

TABLE III. *KM* MAP PIPELINE TIME BREAKDOWN IN SECONDS.

	(a) CPU			(b) GPU		
Hash table	✓	✓	×	✓	✓	×
Combiner	✓	×	×	✓	×	×
Input	4.0	3.6	3.8	3.1	3.4	3.1
Stage	–	–	–	9.5	10.6	10.2
Kernel	168	171	144	13.2	14.8	13.6
Retrieve	–	–	–	10.5	11.2	2.6
Partitioning	3.59	41.2	75.3	0.1	3.0	11.6
Map elapsed time	168	172	145	13.3	14.9	14.4
Merge delay	0.2	13.9	13.9	0.08	2.6	2.5
Reduce time	0.036	1.3	1.3	0.6	2.3	2.3

2) Map Pipeline: K-Means: Table III(a) presents the map pipeline timers of *KM* using the CPU as the compute device with the same 3 configurations as for *WC* in table II. Unlike *WC*, *KM* is compute-intensive and is dominated by the map kernel execution as presented by the relative times of the different pipeline stages. In configuration *(i)*, *KM* uses the hash table to store its intermediate data and applies the combiner to reduce its volume. Configuration *(ii)* shows an increase in kernel execution and partitioning for the same reasons as *WC* in the absence of a combiner function. Configuration *(iii)* exhibits improved kernel execution time compared to its peers as it relies on the faster simple output collection method. However, as opposed to *WC*, *KM* produces a small data volume. Therefore, the intermediate data partitioning time does not exceed the dominating kernel execution time. Hence, the time elapsed in the map pipeline is less than in the previous two configurations. The time to finish merging the intermediate data after the map and the reduce time grow significantly in configuration *(ii)* and *(iii)*; the total execution time is still smallest for configuration *(iii)*.

Table III(b) assesses the effects in *KM* of the 3 configurations on the GTX480 GPU, using the same data set as for Table III(a). It is clear from the timers that the kernel execution and elapsed time of the map pipeline using the GPU outperforms the CPU executions. Configuration *(ii)* shows increased kernel time, for the same reason as on the CPU. However, the use of the simple output collection method in configuration *(iii)* does not improve the overall map elapsed time. This can be explained by several factors. First, the increased intermediate data volume places additional stress on the GPU's limited memory bandwidth, limiting the improvement in kernel execution time. Furthermore, the NVidia OpenCL driver adds some coupling between memory transfers and kernel executions, thus introducing artificially high times for nondominant stages; especially the Stage and Retrieve stages suffer from this. Like on the CPU, the increased intermediate volume causes the merge completion time and the reduce time to grow significantly compared to configuration *(i)*.

In conclusion, the use of the hash table in conjunction with the combiner serves as the optimal configuration to match the GPU architecture capabilities. Another notable difference in Table III(b) compared to Table III(a) is the significant drop in intermediate data partitioning time across all configurations even though the host code is using the same CPU. This is because there is no contention on CPU resources by the kernel threads when using the GPU as the compute device.

3) Optimizations on Intermediate Data Handling: Glasswing provides control of fine-grained parallelism to handle intermediate data in a number of ways. First, N, the number of partitioner threads that run concurrently with the map computation can be set. Second, P, the number of partitions per node, can be set. Having $P > 1$ may bring multiple benefits: keys do not have to be compared when they are allocated to different partitions; merging of different intermediate partitions can be done in parallel by separate threads; and intermediate data that grows to exceed its cache can be merged and flushed in parallel for each partition. In the experiments in this section, the number of threads allocated to merging and flushing are chosen equal to P.

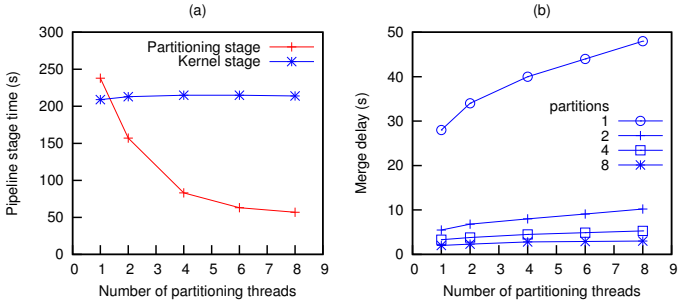

Fig. 4. Analysis of intermediate data handling.

Figure 4(a) shows how N influences the relevant map pipeline stages in *WC*. We do not present the Input stage as it is constant and nondominant, or the Stage and Retrieve stages because they are disabled on the CPU. The elapsed time for the map pipeline equals the maximum of the other two stages that are presented in Figure 4(a). With $N = 1$, the Partitioning stage is dominant; when that stage is parallelized, its time drops below the Kernel stage already from $N \geq 2$ threads onwards.

Variation of P has no noticeable influence on the timings of the map pipeline. However, it strongly influences another phase of Glasswing: the delay until the merge threads have finished after the map phase has completed. Figure 4(b) shows the influence of P and N on the merge delay. An increase in P leads to a sharp decrease in merge delay, even stronger than linear since the merger threads are already active during the map phase. An increase in N causes an increase of the merge delay. This effect is much smaller than that of P.

For *WC*, the conclusion from Figure 4 is that the number of partioning threads must be chosen as 2, and P must be chosen as 8 (we used these settings in the horizontal scalability analysis). These timings depend on the application nature, so for each application these settings must be determined separately.

4) Reduce Pipeline Efficiency: Glasswing provides applications with the capability to process multiple intermediate keys concurrently in the same reduce kernel, each key being processed by a separate thread. An optimization on top of that is to additionally save on kernel invocation overhead by having each kernel thread process multiple keys sequentially. Figure 5 evaluates the effectiveness of these optimizations. It presents the breakdown of *WC*'s reduce pipeline for a varying number of keys to be processed in parallel. The data set used contains millions of unique keys (words). Setting the number of concurrent keys to one causes (at least) one kernel invocation per key, with very little value data per reduce invocation.

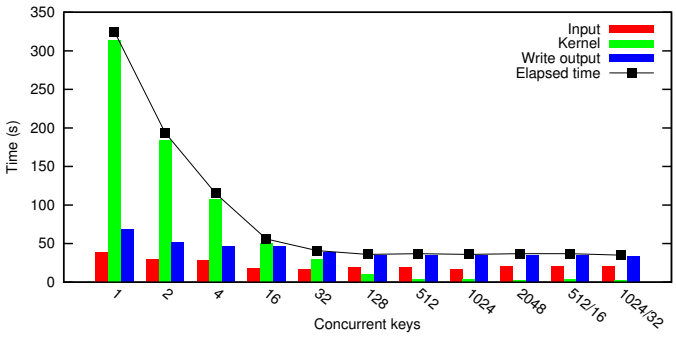

Fig. 5. *WC* reduce pipeline time breakdown in seconds for varying number of keys to be processed concurrently

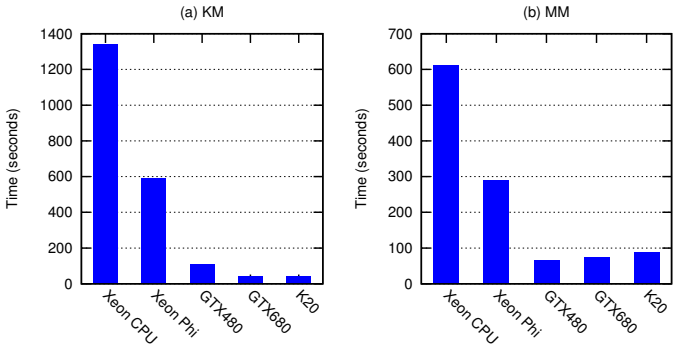

Fig. 6. Glasswing on one node with different accelerators

Figure 5 shows that the kernel execution time decreases sharply with the number of concurrent keys per kernel invocation. The two right-hand bar groups present the optimization to sequentially execute multiple keys per thread. Compared to the same number of concurrent keys to the left in the table, kernel time is improved again because it saves on kernel invocations. The decrease in total elapsed time in the reduce phase flattens off beyond 128 keys per kernel invocation, because the Output writer stage then becomes the dominant stage in the pipeline. The Output writer stage is relatively expensive because the output is formatted into human-readable form. We noted that if the output is generated in binary form, the dominant reduce stage becomes Input, given enough Kernel parallelism.

C. Vertical Scalability

We tested Glasswing with a variety of accelerators to assess its ability to scale vertically when given a more capable compute device. To this end, we ran the compute-intensive applications on the accelerators in *Type-2* nodes: an Intel Xeon Phi, an NVidia GTX680 and K20m. For comparison, we also present measurements of the *Type-1* executions.

Single node execution times for *KM* are presented in Figure 6(a) for a reduced data set size of 256×1024^2 points instead of 1024^3. *KM* allows fine tuning of task granularity which results in good efficiency on all architectures. From instrumented runs, we found that execution time is completely dominated by the map kernel stage for *KM* on all architectures. Xeon Phi is faster than the Xeon CPUs as it has a larger number of slower cores. The execution times on the NVidia GPUs scale with the number of compute cores indicating near perfect scalability.

Single node execution times for *MM* are given in Figure 6(b) for matrices of size 16384^2 instead of 32768^2. The Xeon Phi again outperforms the Xeon CPU. The GTX480 performance matches that of GTX680 even though the latter has more compute capacity. This is attributed to lower memory bandwidth per core and less fast memory per core. Similarly, the GTX480 performance matches the K20m. From instrumentation, we find that the kernel execution times are almost equal, which is again attributed to memory properties. The difference in total time is caused by a slower disk on the K20m node.

We also performed vertical scalability tests on a number of different Amazon EC2 instance types; the results, described in [2], corroborate our findings here.

V. Conclusion

Glasswing is a scalable MapReduce framework that is capable of improving the utilization of modern hardware. It allows for a combination of coarse-grained and fine-grained parallelism without sacrificing the programming model's abstractions. Glasswing was designed to handle out-of-core data set sizes and thus complies with the true purpose of the MapReduce model.

We experimentally evaluated Glasswing and demonstrated its ability to overlap kernel executions with memory transfers, network communication and disk access. A study into horizontal scalability of five applications revealed that Glasswing exhibits superior performance and scalability compared to Hadoop. Glasswing's architecture boosts applications on multicore CPUs, and offers further acceleration with GPUs for compute-intensive applications. Glasswing also outperforms GPMR, a platform that especially targets GPUs. Additionally, we presented the effects of the various optimizations built into Glasswing and the contribution of each to performance. Furthermore, Glasswing demonstrated its ability to leverage a variety of compute devices and vertically scale its performance based on the available resources.

We learned several lessons from our experience designing and developing Glasswing. MapReduce applications exhibit a high variation in compute intensity, data intensity and key-space sparseness necessitating different combinations of optimizations. Allowing the application to specify the granularity of parallelism improves the utilization of the available resources. Moreover, our usage of managed buffer pools allows for predictable and reproducible performance and scalability, also for big data. Finally, MapReduce frameworks need to consider novel methods of harnessing the power of the available resources.

Acknowledgment

The authors would like to thank the anonymous reviewers, Thilo Kielmann for his critical feedback and Kees Verstoep for excellent administration of the VU/DAS4 cluster. This work was partially funded by the Netherlands Organization for Scientific Research (NWO).

References

[1] R. V. Van Nieuwpoort, G. Wrzesińska, C. J. H. Jacobs, and H. E. Bal, "Satin: A high-level and efficient grid programming model," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 3, pp. 9:1–9:39, Mar. 2010.

[2] 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*, ser. HPDC '14. ACM, 2014, pp. 295–298. [Online]. Available: http://doi.acm.org/10.1145/2600212.2600706

[3] Ganjisaffar, Yasser and Debeauvais, Thomas and Javanmardi, Sara and Caruana, Rich and Lopes, Cristina Videira, "Distributed Tuning of Machine Learning Algorithms Using MapReduce Clusters," in *Proceedings of the Third Workshop on Large Scale Data Mining: Theory and Applications*, ser. LDMTA '11. ACM, 2011, pp. 2:1–2:8. [Online]. Available: http://doi.acm.org/10.1145/2002945.2002947

[4] Tamano, Hiroshi and Nakadai, Shinji and Araki, Takuya, "Optimizing Multiple Machine Learning Jobs on MapReduce," in *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, ser. CLOUDCOM '11. IEEE Computer Society, 2011, pp. 59–66. [Online]. Available: http://dx.doi.org/10.1109/CloudCom.2011.18

[5] Zein, Ahmed and Mccreath, Eric and Rendell, Alistair and Smola, Alex, "Performance Evaluation of the NVIDIA GeForce 8800 GTX GPU for Machine Learning," in *Proceedings of the 8th International Conference on Computational Science, Part I*, ser. ICCS '08. Springer-Verlag, 2008, pp. 466–475. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69384-0_52

[6] Raina, Rajat and Madhavan, Anand and Ng, Andrew Y., "Large-scale Deep Unsupervised Learning Using Graphics Processors," in *Proceedings of the 26th Annual International Conference on Machine Learning*, ser. ICML '09. ACM, 2009, pp. 873–880. [Online]. Available: http://doi.acm.org/10.1145/1553374.1553486

[7] "Apache Software Foundation. Hadoop." http://hadoop.apache.org/.

[8] J. Stuart and J. Owens, "Multi-GPU MapReduce on GPU Clusters," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, May 2011, pp. 1068 –1079.

[9] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. USENIX Association, 2004, pp. 10–10.

[10] B. Palanisamy, A. Singh, L. Liu, and B. Jain, "Purlieus: locality-aware resource allocation for MapReduce in a cloud," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. ACM, 2011, pp. 58:1–58:11.

[11] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin, "Improving MapReduce performance through data placement in heterogeneous Hadoop clusters," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, April 2010, pp. 1 –9.

[12] Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal, "Hadoop acceleration through network levitated merge," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. ACM, 2011, pp. 57:1–57:10.

[13] J. Tan, A. Chin, Z. Z. Hu, Y. Hu, S. Meng, X. Meng, and L. Zhang, "DynMR: Dynamic MapReduce with ReduceTask Interleaving and MapTask Backlling," in *Proceedings of the 9th ACM European Conference on Computer Systems*, ser. EuroSys '14. ACM, 2014.

[14] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. USENIX Association, 2008, pp. 29–42.

[15] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*, ser. EuroSys '10. ACM, 2010, pp. 265–278.

[16] K. Shirahata, H. Sato, and S. Matsuoka, "Hybrid Map Task Scheduling for GPU-Based Heterogeneous Clusters," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, Dec 2010, pp. 733 –740.

[17] F. Ji and X. Ma, "Using Shared Memory to Accelerate MapReduce on Graphics Processing Units," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, May 2011, pp. 805 – 816.

[18] L. Chen and G. Agrawal, "Optimizing MapReduce for GPUs with effective shared memory usage," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '12. ACM, 2012, pp. 199–210.

[19] L. Chen, X. Huo, and G. Agrawal, "Accelerating MapReduce on a coupled CPU-GPU architecture," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. IEEE Computer Society Press, 2012, pp. 25:1–25:11.

[20] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, Feb. 2007, pp. 13 –24.

[21] J. Talbot, R. M. Yoo, and C. Kozyrakis, "Phoenix++: modular MapReduce for shared-memory systems," in *Proceedings of the second international workshop on MapReduce and its applications*, ser. MapReduce '11. ACM, 2011, pp. 9–16.

[22] R. Chen, H. Chen, and B. Zang, "Tiled-MapReduce: optimizing resource usages of data-parallel applications on multicore with tiling," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10. ACM, 2010, pp. 523–534.

[23] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT '08. ACM, 2008, pp. 260–269.

[24] 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*, ser. PACT '10. ACM, 2010, pp. 217–226.

[25] R. Farivar, A. Verma, E. Chan, and R. Campbell, "MITHRA: Multiple data independent tasks on a heterogeneous resource architecture," in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, 31 2009-Sept. 4 2009, pp. 1 –10.

[26] M. Grossman, M. Breternitz, and V. Sarkar, "HadoopCL: MapReduce on Distributed Heterogeneous Platforms Through Seamless Integration of Hadoop and OpenCL," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, ser. IPDPSW '13. IEEE Computer Society, 2013, pp. 1918–1927. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2013.246

[27] AMD. Aparapi. [Online]. Available: http://developer.amd.com/tools-and-sdks/opencl-zone/aparapi/

[28] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "FPMR: MapReduce framework on FPGA," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '10. ACM, 2010, pp. 93–102.

[29] M. Rafique, B. Rose, A. Butt, and D. Nikolopoulos, "CellMR: A framework for supporting mapreduce on asymmetric cell-based clusters," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, May 2009, pp. 1 –12.

[30] A. Papagiannis and D. Nikolopoulos, "Rearchitecting MapReduce for Heterogeneous Multicore Processors with Explicitly Managed Memories," in *Parallel Processing (ICPP), 2010 39th International Conference on*, Sept. 2010, pp. 121 –130.

[31] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," *SIGPLAN Not.*, vol. 43, no. 3, pp. 287–296, Mar. 2008.

[32] "WikiBench: Web Hosting Benchmark," http://www.wikibench.eu/.

[33] "Wikipedia: Database download," http://en.wikipedia.org/wiki/Wikipedia:Database_download.