HeteroDoop : A MapReduce Programming System for Accelerator Clusters

Amit Sabne
Purdue University
asabne@purdue.edu

Putt Sakdhnagool
Purdue University
psakdhna@purdue.edu

Rudolf Eigenmann
Purdue University
eigenman@purdue.edu

ABSTRACT

The deluge of data has inspired big-data processing frameworks that span across large clusters. Frameworks for MapReduce, a state-of-the-art programming model, have primarily made use of the CPUs in distributed systems, leaving out computationally powerful accelerators such as GPUs. This paper presents HeteroDoop, a MapReduce framework that employs both CPUs and GPUs in a cluster. HeteroDoop offers the following novel features: (i) a small set of directives can be placed on an existing sequential, CPU-only program, expressing MapReduce semantics; (ii) an optimizing compiler translates the directive-augmented program into a GPU code; (iii) a runtime system assists the compiler in handling MapReduce semantics on the GPU; and (iv) a tail scheduling scheme minimizes job execution time in light of disparate processing capabilities of CPUs and GPUs. This paper addresses several challenges that need to be overcome in order to support these features. HeteroDoop is built on top of the state-of-the-art, CPU-only Hadoop MapReduce framework, inheriting its functionality. Evaluation results of HeteroDoop on recent hardware indicate that usage of even a single GPU per node can improve performance by up to 2.78x, with a geometric mean of 1.6x across our benchmarks, compared to a CPU-only Hadoop, running on a cluster with 20-core CPUs.

Categories and Subject Descriptors

D.1.3 [**Software**]: PROGRAMMING TECHNIQUES—*Concurrent Programming, Distributed programming*; D.3.3 [**Software**]: PROGRAMMING LANGUAGES—*Frameworks*; D.3.4 [**Programming Languages**]: Compilers

Keywords

Distributed Frameworks; MapReduce; Accelerators; Source-to-source Translation; Scheduling

1. INTRODUCTION

A growing number of commercial and science applications in both classical and new fields process very large data volumes. Dealing with such volumes requires processing in parallel, often on systems that offer high compute power.

For this type of parallel processing, the MapReduce paradigm has found popularity. The key insight of MapReduce is that many processing problems can be structured into one or a sequence of phases, where a first step (Map) operates in fully parallel mode on the input data; a second step (Reduce) combines the resulting data in some manner, often by applying a form of reduction operation. MapReduce programming models allow the user to specify these map and reduce steps as distinct functions; the system then provides the workflow infrastructure, feeding input data to the map, reorganizing the map results, and then feeding them to the appropriate reduce functions, finally generating the output.

The large data volumes involved may not fit on a single node. Thus, distributed architectures with many nodes may be needed. Among the systems that support MapReduce on distributed architectures, Hadoop [1] has gained wide use. Hadoop provides a framework that executes MapReduce problems in a distributed and replicated storage organization (the Hadoop Distributed File System – HDFS). In doing so, it also deals with node failures.

Big-data problems pose high demands on processing and IO speeds, often with emphasis on one of the two. For general compute-intensive problems, accelerators, such as NVIDIA GPUs and Intel Phis, have proven their value for an increasing range of applications. To obtain high performance, their architectures make different chip real-estate tradeoffs between processing cores and memory than in CPUs. A larger number of simpler cores provide higher aggregate processing power and reduce energy consumption, offering better performance/watt ratios than CPUs. In GPUs, intra-chip memory bandwidth is high and multithreading reduces the effective memory access latency. These optimizations come at the cost of an intricate memory hierarchy, reduced memory size, data accesses that are highly optimized for inter-thread contiguous (a.k.a. coalesced) reference patterns and explicitly parallel programming models. Using these architectures therefore requires high programmer expertise.

While accelerators tend to perform well on compute-intensive applications, IO-intensive MapReduce problems may not always benefit. Previous research efforts on MapReduce-like systems employ either GPUs [2, 3, 4] alone, disregarding IO-intensive applications, or CPUs [1, 5, 6, 7] alone, losing out on GPU acceleration. In this paper we present our

HeteroDoop[1] system, which exploits *both* CPUs and GPUs in a cluster, as needed by the application.

The first challenge in developing such a heterogeneous MapReduce system is the programming method. In a naive scheme, the programmer would have to write two program versions, one for CPUs and the second for GPUs. This need arises as accelerators rely on explicitly parallel programs, be it either low-level programming models such as CUDA [8] and OpenCL [9], or high-level ones, such as OpenACC [10] and OpenMP 4.0 [11]. Although available high-level programming models relieve the user from having to learn model-specific APIs, such as in CUDA or OpenCL, they still require explicit parallel programming. On the other hand, in CPU-oriented MapReduce systems, programmers write only sequential code; the underlying framework automatically employs all cores in the cluster by concurrently processing the input data, which is split into separate files. Previous research on GPU uses for MapReduce has either relied on explicitly parallel codes with accelerator-specific optimizations [3, 4, 12, 13], and/or on specific MapReduce APIs [2, 3, 13, 14]. Programmability in both approaches is poor; the former requires learning low-level APIs and the latter necessitates application rewriting. To overcome these limitations, our contribution enables programmers to port already available sequential MapReduce programs to heterogeneous systems by annotating the code with *HeteroDoop directives*. Inserting such directives is straightforward, requires no additional accelerator optimizations, and leads to a single input source code for both CPUs and GPUs. Furthermore, the resulting code is portable; it can still execute on CPU-only clusters.

The second key challenge in exploiting accelerators is their limited, non-virtual memory space. The default parallelization scheme used in MapReduce/Hadoop engages multiple cores by processing separate input files in parallel – typically one per core, as an individual task. Data input is appropriately partitioned into separate *fileSplits*, which are fed to the different compute nodes and their threads. Simultaneous accesses by many threads to their fileSplits require a large memory. This size requirement is not a problem in today's typical CPUs with 4 to 48 cores and virtual memory support; however, in GPUs with several hundred cores and possibly thousands of threads, the available, non-virtual memory is insufficient. Our second contribution addresses this challenge by processing the data records *within* a fileSplit in parallel on the accelerator, while retaining the default processing scheme on the CPU.

The third challenge is to translate the annotated, sequential source code in a way that maximizes performance. Our contribution includes an optimizing source-to-source translator, assisted by a runtime system, that generates CUDA code and splits the work across CPUs and GPUs. In doing so, several issues arise. The first one is load imbalance across GPU threads, as the records processed in parallel may be of different size. A global work-stealing approach would incur high overheads, due to excessive atomic accesses by the GPU threads. HeteroDoop overcomes this issue by using a novel record-stealing approach that partitions the records statically across threadblocks but dynamically within threadblocks. Another issue is that GPU memory is statically allocated but the size of the map phase output, the key-value pairs, is not known a priori. Over-allocation would lead to inefficient sorting of the key-value pairs, which follows the map phase. To resolve this issue, HeteroDoop includes a fast runtime compaction scheme, resulting in efficient sort. Furthermore, the runtime system executes the sort operation on the GPU rather than on the slower CPU. Other optimizations include efficient data placement in the complex GPU memory hierarchy and automatic generation of vector load/store operations.

A final challenge in developing such a heterogeneous MapReduce scheme is to account for the different processing speeds of CPUs and GPUs for work partitioning. While prior work [15, 16] has dealt with load balancing across nodes, intra-node heterogeneity has remained an issue. HeteroDoop's *tail scheduling* scheme addresses this issue. Our contribution is based on a key observation: the load imbalance only arises in the execution of the final tasks in a job; careful GPU-speedup-based scheduling of the tailing tasks can avoid the imbalance.

We have evaluated the HeteroDoop framework on eight applications, comprising well-known MapReduce programs as well as scientific applications. We demonstrate the utility of HeteroDoop on a 48-node, single GPU cluster with large datasets, and on a 64-node, 3-GPU cluster with in-memory datasets. Our main results indicate that the use of even a single GPU per node can speed up the end-to-end job execution by up to 2.78x, with a geometric mean of 1.6x, as compared to CPU-only Hadoop, running on a cluster with 20-core CPUs. Furthermore, the execution time scales with the number of GPUs used per node.

The remainder of the paper is organized as follows: Section 2 provides background on GPUs, MapReduce, and Hadoop. Section 3 describes the HeteroDoop constructs, followed by the compiler design in Section 4. Section 5 describes the overall execution flow of the HeteroDoop framework and details the runtime system. The tail scheduling scheme is explained in Section 6. Section 7 presents the experimental evaluation. Section 8 discusses related work, and Section 9 concludes the paper.

2. PRELIMINARIES

We introduce the basic terminology used in this paper for the GPU/CUDA architecture and programming model as well as the MapReduce and Hadoop concepts. We keep the discussion of GPUs and CUDA brief, assuming reader familiarity, but we do refer to introductory material [17].

2.1 CUDA Basics

In the CUDA [8] GPGPU architecture, the many GPU cores are structured into multiple Streaming Multiprocessors (SM). CUDA threads execute in SIMD fashion, where a *warp* consisting of 32 threads executes a single instruction and multiple warps time-share an SM in multi-threading mode. Storage is separate from the CPU address space; it is structured into the global (or device) memory, the per-SM shared memory (user managed cache), and specialized memories. The latter include the read-only constant and texture memories as well as the registers.

CUDA programming is explicitly parallel with user-driven *offloading*; i.e, the user identifies compute-intensive code sections, *kernels*, which are to be run on the GPU (the device) and inserts data transfers between CPU and GPU. The threads are organized into threadblocks. All threads within a threadblock execute on the same SM. The programmer

[1]Download from: `http://bit.ly/1BwxGER`

manages most of the storage hierarchy explicitly, including fitting the data into the limited-size memories. There is no virtual memory support.

2.2 MapReduce and Hadoop

In the MapReduce model, programmers write a *map* and a *reduce* function, with the system organizing the overall execution workflow. The input data is placed on a distributed file system, such as the HDFS [18] (Hadoop Distributed File System). HDFS stores the input in blocks, or *fileSplits*. A job consists of a set of map and reduce tasks. In Hadoop, one node usually acts as master and the others as slaves. The master node runs a *JobTracker*, while each slave runs a *TaskTracker* – together they orchestrate the necessary map and reduce tasks in a way that exploits data locality, engages all available nodes and cores, and provides fault tolerance. The total number of concurrent tasks in a Hadoop cluster is typically the same as the number of available cores.

In Hadoop, each map task processes one fileSplit. The map function applies a map operation to each data record in this fileSplit. The map task emits a set of <key, value> pairs (or *KV pairs*). The framework puts these KV pairs into different partitions, each partition targeted at a particular reduce task. To form these partitions, the KV pairs are sorted by their keys and split by a default or user-provided partitioning function.

Each partition is then sent to its target reduce task. This task typically resides on a different node, making this a costly step. To reduce the cost, a task-local, user-provided reduction operation, known as the *combiner*, is applied first on each partition, minimizing the size of the communicated data. This communication is also known as the *shuffle phase*.

In the next phase, the *sort phase*, each reduce task merges the incoming partitions into a sorted list. Then the *reduce phase* applies the reduce function to these data. The output is written back to the HDFS. As the data volumes involved are typically very large, the intermediate results between map and reduce are typically written to the local disk.

Hadoop uses a *heartbeat* mechanism for communication between the JobTracker and TaskTracker. TaskTrackers send heartbeats to the JobTracker at regular intervals. These heartbeats include items such as status of tasks and free cores or *slots*. If the JobTracker finds that a TaskTracker has free slots, it schedules new tasks on the particular TaskTracker in the heartbeat response.

Hadoop is written in Java, and hence the baseline system supports input functions written in Java. Hadoop Streaming [19] is an extension that supports other languages for writing MapReduce programs. It is implemented so that the map, combine, and reduce functions obtain their input from standard input and write the output onto standard output; map, combine, and reduce can be written as unix-style "filter" functions, using a language of choice.

3. HETERODOOP DIRECTIVES

Recall that the HeteroDoop directives are designed to exploit both CPUs and GPUs in a cluster with a single source program. They identify the *map* and *combine* functions and their attributes in a serial, CPU-only MapReduce program. From this information, our translator generates code that exploits both the CPUs and GPUs available in the nodes of a distributed architecture. While the concept of HeteroDoop directives is language-independent, our HeteroDoop prototype supports programs written in C. Our implementation makes use of the Hadoop Streaming framework, which we extend to enable efficient execution on both CPUs and GPUs.

Listing 1: Wordcount Map Code with HeteroDoop Directives

```
1  int main() {
2    char word[30], *line;
3    size_t nbytes = 10000;
4    int read, linePtr, offset, one;
5    line = (char*) malloc(nbytes*sizeof(char));
6    #pragma mapreduce mapper key(word) value(one) \\
7    keylength(30) kvpairs(20)
8    while( (read = getline(&line, &nbytes, stdin)) != -1) {
9      linePtr = 0;
10     offset = 0;
11     one = 1;
12     while( (linePtr = getWord(line, offset, word,
13         read, 30)) != -1) {
14       printf("%s\t%d\n", word, one);
15       offset += linePtr;
16     }
17   }
18   free(line);
19   return 0;
20 }
```

3.1 HeteroDoop Directives with an Example

A key insight used in the design of HeteroDoop constructs is the observation that both map and combine functions iterate over a non-predetermined number of records. The bulk of the computation of the map and combine functions is performed inside a *while* loop. HeteroDoop directives identify these loops and express attributes that allow the translator to generate efficient parallel code for the GPUs.

Listing 1 shows an example *map* code, written in C, for *Wordcount*. This application counts the occurrences of each word in a set of input files. The code reads each input line and applies to it an elementary map operation. For each word in the line, this map operation emits a KV pair < word, 1>. Notice that the input is read from STDIN using the *getline* function, while the output is written to STDOUT via *printf*.

In general, the elementary map operation is applied to every record. By default a record is a line of input. The bulk of the map computation lies within the loop iterating over these records – lines 8–17 in the example. The *mapreduce* directive on line 6 with the *mapper* clause tells the compiler that this loop applies the map operation. The *key* and *value* clauses identify the variables used for emitting KV pairs. The *keylength* and *vallength* clauses indicate the lengths of the respective variables; the clauses are needed if these variables do not have a compiler-derivable type. Table 1 lists all HeteroDoop directives and clauses. Some of these clauses are optional; users need to provide them only for further optimizations and tuning of the generated GPU code.

Listing 2 shows an example *combine* code for the same *Wordcount* application. Recall that, before the combiner is run, the underlying HeteroDoop system sorts the KV pairs emitted by the map according to their keys and places them into different partitions. The combiner function operates on each partition and sums up the occurrences of each word. Note that the *while* loop in the map function can execute in parallel, but the one in the combine function cannot, except for reduction-style parallelism. The only readily available parallelism in the combine and reduce executions exists across partitions. Typically, the number of partitions is not high enough to exploit the GPU completely. For this rea-

Listing 2: Wordcount Combine Code with HeteroDoop Directives

```
1  int main() {
2    char word[30], prevWord[30]; prevWord[0] = '\0';
3    int count, val, read; count = 0;
4    #pragma mapreduce combiner key(prevWord) value(count)
5    keyin(word) valuein(val) keylength(30) vallength(1)
6    firstprivate(prevWord, count) {
7      while( (read = scanf("%s %d", word, &val)) == 2 ) {
8        if(strcmp(word, prevWord) == 0 ) {
9          count += val;
10       } else {
11         if(prevWord[0] != '\0')
12           printf("%s\t%d\n", prevWord, count);
13         strcpy(prevWord, word);
14         count = val;
15       }
16     }
17     if(prevWord[0] != '\0')
18       printf("%s\t%d\n", prevWord, count);
19   }
20   return 0;
21 }
```

son, HeteroDoop provides no directives for reduce functions and executes them on the CPUs only. For combine functions, however, as the data is already present in the GPU memory, HeteroDoop employs an economical way for GPU execution (Section 4.2).

Two extra clauses are necessary on the combiner function : *keyin* specifies the variable that receives the map-generated key and *valuein* does the same for the map-generated value.

3.2 Clauses for Memory and Thread Attributes

HeteroDoop directives also allow for clauses that improve performance by specifying the attributes of the data and threads. The following constructs exist:

Read-only variables: The *sharedRO* clause lists read-only variables. The compiler places such variables in faster GPU memories, such as the constant memory and the texture memory. By default, our compiler places read-only scalars in the constant memory. Arrays whose sizes are known at compile time are placed in the texture memory, otherwise in the global memory. The *texture* clause forces placement in the texture memory. Placing data in the texture memory is especially useful for random array accesses, as this memory comes with a separate on-chip cache.

Firstprivate: This clause specifies variables that can be privatized during the map or combine operation but are initialized beforehand. In the absence of this clause, the compiler tries to identify such variables automatically. It issues a warning if the analysis is inaccurate, e.g., due to aliasing.

Notice that in the MapReduce programming model, all written variables are privatizable during the map and combine operations. There is no shared written data. The translator performs the privatization without the need for user directives.

Space Allocation for KV Pairs: The space needed for the KV pairs output by map is not known at translation time. The translator allocates all free GPU memory for storing these KV pairs. In practice, this leads to an over-allocation. To reduce the memory required for storing these KV pairs, the users can indicate the maximum number of KV pairs emitted by each record using the *kvpairs* clause. This reduction in the storage space required for the KV pairs improves the aggregation efficiency for this storage, as will be described in Section 4.3.

Thread Attributes: The *blocks* and *threads* clauses allow programmers to choose the number of threadblocks and number of threads in a threadblock on the GPU, resp. These clauses help tune the map and combine kernel performance.

Table 1: HeteroDoop Directives

Clause	Arguments	Description	Optional
mapper		Specifies that the attached region performs map operation	No
combiner		Specifies that the attached region performs combine operation	No
key	Variable name	Specifies the variable that contains the key	No
value	Variable name	Specifies the variable that contains the value	No
keyin	Variable name	Specifies the variable that contains the incoming key. Valid only on the combiner	No
valuein	Variable name	Specifies the variable that contains the incoming value. Valid only on the combiner	No
keylength	Integer variable	Specifies the length of the key being emitted	No
vallength	Integer variable	Specifies the length of the value being emitted	No
firstprivate	A set of variable names	These variables are initialized before being used in the attached region	No
sharedRO	A set of variable names	These are read-only inside the map or combine regions	Yes
texture	A set of variable names	These variables are read-only and hence can be placed in GPU texture memory	Yes
kvpairs	Integer variable	This is an optional clause on map region specifying the maximum KV pairs that can be emitted from a single record	Yes
blocks	Integer variable	Specifies the number of threadblocks to use	Yes
threads	Integer variable	Specifies the number of threads in a threadblock	Yes

4. COMPILER

This section describes the HeteroDoop source-to-source translator, which converts an input MapReduce code annotated with HeteroDoop directives into a CUDA program. It is built using the Cetus [20] compiler infrastructure. Translating directly into CUDA, rather than a higher-level model such as OpenACC or OpenMP 4.0, provides direct access to GPU-specific features which are exploited by the compiler optimizations. For the code running on the CPUs, the same Hadoop Streaming code is compiled using the *gcc* backend compiler. In this manner, a single MapReduce source is sufficient for the execution on both CPUs and GPUs. The next two subsections describe the key translation steps of generating GPU kernel code for the map and the combine functions, respectively. Section 4.3 describes the code generation for the host CPU, which offloads the kernels.

4.1 Map Kernel Generation

Each thread in the map kernel fetches a record, performs the elementary map operation, and stores the generated KV pairs into its portion of a central GPU storage, called the *global KV store*. The process repeats until all records are done.

The compiler begins by locating the annotation with the *mapper* clause. The annotated *while* loop is the target region for kernel generation. The compiler extracts this region into a new function, *newGPUKernel*, which contains the kernel code. A key step in doing this is the transformation of the

Algorithm 1 Handling Variables in Generated Kernels

Input: *region* - Region attached to the annotation
Input: *newGPUKernel* - Extracted region copy (kernel)
Input: *sharedROSet* - Set of shared read-only variables inside *newGPUKernel*
Input: *textureSet* - Set of shared read-only variables to be placed on texture memory
Input: *firstPrivateSet* - Set of firstprivate variables
Output: Correct placement of variables in *newGPUKernel*, along with necessary GPU memory allocation and data transfer generation

```
1:  function HANDLEVARIABLES(region, newGPUKernel,
    sharedROSet, textureSet, firstPrivateSet)
2:    usedVars = newGPUKernel.getUsedVars()
3:    For each var ∈ usedVars do
4:      if (sharedROSet contains var) then
5:        if (var is scalar) then
6:          newVar = addParameter(var, newGPUKernel)
    //gets placed on constant memory
7:        else   //a shared read-only array
8:          newVar = addParameter(var, newGPUKernel)
9:          insertMallocAndCpyIn(region, newVar, var)
10:       end if
11:     else if (textureSet contains var) then
12:       tex = createNewTexture(var)
13:       newVar = addParameter(var, newGPUKernel)
14:       bindTexture(tex, newVar, region)   //cudaBindTexture
15:       insertMallocAndCpyIn(region, newVar, var)
    //inserts cudaMalloc and cudaMemCpy
16:     else   //a private variable
17:       newVar = addPrivateVar(newGPUKernel, var)
18:       if (firstPrivateSet contains var) then
19:         newFPCopy = addParameter(var, newGPUKernel)
20:         if (var is array) then
21:           insertMallocAndCpyIn(region, newFPCopy, var)
22:         end if
23:         insertInKernelCopyCode(newFPCopy, newVar);
24:       end if
25:     end if
26:   end for
27: end function
```

different variable types. Algo. 1 shows the procedure. From the user annotations, the compiler generates *sharedROSet*, which lists the shared read-only variables. *TextureSet* consists of read-only array variables that are to be placed in the texture memory. *FirstPrivateSet* contains all variables that are firstprivate, either specified by the programmer or identified by the compiler. The scalar *sharedRO* variables are passed as arguments to the kernel; the underlying CUDA compiler places these arguments in *constant* memory (lines 5–6). A pointer to each array *sharedRO* variable is passed through a kernel parameter (lines 8–9); storage is allocated on the GPU and the array data is copied from the CPU into this space. The transformation for using the texture memory is essentially the same as for *sharedRO* arrays (lines 11–15). The functions *addParameter* and *addPrivateVar* create new variables and automatically rename the previous variable names in the *newGPUKernel* respectively.

The remaining variables inside the *newGPUKernel* are private (lines 17–24). For each private variable, a new variable is created inside the kernel body using the *addPrivateVar* function. For firstprivate variables, there is an extra step. For scalars, the initial value is passed through a kernel parameter. For arrays, storage is allocated and a reference is passed via kernel parameter. The initial values of the array elements are copied into the corresponding memory locations before the kernel. Inside the kernel body, each thread copies these values into the private spaces using the *insertInKernelCopyCode* function.

Listing 3 shows the translated CUDA map kernel for *Wordcount* (Listing 1). The kernel generation procedure

Listing 3: Translated Wordcount Mapper Code

```
1  __global__ void gpu_mapper(char *ip, int ipSize,
2  int *recordLocator, char * devKey, int * devVal,
3  int storesPerThread, int * devKvCount,  int keyLength,
4  int valLength, int * indexArray, int numReducers) {
5    char gpu_word[30];
6    int gpu_read, gpu_one, gpu_offset, gpu_linePtr;
7    int index, tid, start;
8    __shared__ unsigned int recordIndex;
9    mapSetup(&start, &tid, &index, ipSize, storesPerThread,
10   ip, devKvCount, numReducers, &recordIndex);
11   while( ( gpu_read = getRecord(ip, &recordIndex, &start,
12         recordLocator) )!= - 1) {
13     gpu_linePtr = 0;
14     gpu_offset = 0;
15     gpu_one = 1;
16     while( (gpu_linePtr = getWord(ip + start, gpu_offset,
17           gpu_word, gpu_read, 30)) != -1) {
18       emitKV(gpu_word, &gpu_one, devKey, devVal, &index,
19       devKvCount, keyLength, valLength, numReducers,
20       storesPerThread, indexArray);
21       gpu_offset += gpu_linePtr;
22     }
23   }
24   mapFinish(index, storesPerThread, devKey, keyLength,
25   indexArray, numReducers, devKvCount);
26 }
```

adds internal parameters for bookkeeping. *Ip* represents the input file buffer; *ipSize* is the input file size. *RecordLocator* is a data structure that keeps a list of starting addresses of all input records. *DevKey* and *devVal* are the GPU variables for keys and values of the global KV store, respectively. *StoresPerThread* holds the storage size allocated to each thread in the global KV store. *DevKvCount* array keeps a count of the KV pairs emitted by each thread.

The first GPU-specific translation step is the insertion of a *mapSetup* function call (line 9) for the map execution. This function sets up internal variables for GPU execution e.g. *tid*, which stores the thread ID. Next, the algorithm replaces the CPU record input function, such as *getline*, with a GPU equivalent *getRecord* function (line 11). Similarly, the KV-emitting function, which is *printf* in the CPU code, is replaced with a GPU equivalent *emitKV* function (line 18). The *emitKV* function puts the generated KV pairs into the global KV store. The translation scheme replaces the calls to all C standard library functions with GPU counterparts. Since the current GPUs do not support all C standard library calls, their equivalent implementations are provided by the runtime system. The runtime system also includes other functions used in the translated code, such as *mapSetup*, *getRecord*, and *emitKV*. At the end of the map kernel execution, the *indexArray* holds the locations of individual KV pairs in the global KV store. This array is useful in indirectly accessing the global KV store.

The *mapFinish* function (line 25) performs bookkeeping after a thread is done with the map execution. Most importantly, it keeps a count of the total number of KV pairs emitted by each thread.

Record Stealing: The execution time taken by the map operation for each record can vary greatly among different records in certain MapReduce applications due to the differences in the amount of data in each record. For example, in the *kmeans* application, where each record contains a list of movie ratings, some records have fewer reviews than others. This leads to load imbalance among threads if the records are statically partitioned. A better strategy is therefore to perform dynamic record distribution, or *record stealing*. However, a global record distribution scheme would require global atomic functions on the GPU, which are expensive. We therefore devise a scheme where the records are

statically and equally split among the threadblocks used in the map kernel, and threads of a given threadblock steal a new record from the threadblock's pool. As it is common for larger records to get distributed across threadblocks, record stealing implemented at the threadblock level is effective. The variable *recordIndex* (Listing 3, Line 9) acts as a counter for the records used by the threads of a threadblock. This variable is placed in the GPU shared memory for fast atomic increment operations. The maximum record stealing that a thread can perform is limited by the *storesPerThread* it has in the global KV store.

Using Vector Data Types: For array keys/values, the generated code uses an optimization of internally using CUDA-specific vector data types, such as *char4*, which increase the memory accessing performance. The functions that exploit such a vectorization include *emitKV*, and string functions called within the map code, e.g. *strcpy*.

4.2 Combine Kernel Generation

As the combine function operates on one partition at a time, there is no explicit parallelism. However, different partitions can be processed in parallel. Unfortunately, we have found that the number of partitions i.e., the number of reducers can be low in certain MapReduce applications. Therefore, the degree of exploitable parallelism can be low, leading to underutilization of the GPU. Our scheme therefore exploits in-partition, reduction-style parallelism, while sacrificing full functional equivalence with respect to the CPU code. The idea for this approach is simple: e.g. for the *Wordcount* code, a particular partition received the following KV pairs <a,1>, <a,1>,<a, 1>, <b,1>. The output from a CPU combiner would be <a,3>, <b,1>. However, if two different threads were to operate on the partition, with the first operating on the first two KV pairs, and the second on the last two, the intermediate output would be <a,2>, <a,1>, <b,1>. In this manner, the functional equivalence of the combiner is traded off for parallelism. Due to the presence of the global reducers, however, this trade-off is legal; the global reducer will eventually produce the same output. In practice, as the number of KV pairs in each partition is typically high, this small dissimilarity has a negligible impact on the communication volume.

A second design choice in the combine kernel generation deals with the fact that the degree of exploitable in-partition parallelism is still usually much less than the number of GPU threads. The compiler-generated code forces all threads in a warp to execute the same combine function redundantly. This way, intra-warp thread divergence is eliminated. Listing 4 shows the generated kernel for the *Wordcount* combine code of Listing 2. The compiler inserted a number of parameters: *Keys* and *values* hold the KV pairs emitted by the map for the particular reducer. *OpKey* and *opVal* store the combine-emitted KV pairs. The lengths of keys and values for both map and combine functions are passed as parameters. The *handleVariables* pass (Algo. 1) adds two parameters *prevWordFP* and *countFP* for firstprivate variables.

Similar to the map setup function, the *combineSetup* function (line 11) initializes the internal variables of the combiner operation. The compiler performs an optimization of placing the private array variables in the faster shared memory for each warp. In this example, *gpu_prevWord* and *gpu_word* are placed in the shared memory. The scalars are placed

Listing 4: Translated Wordcount Combiner Code

```
1  __global__ void gpu_combiner(char *keys, int *values,
2  char *opKey, int *opVal, int *indexArray, int *finalCount,
3  int size, int mapKeyLength, int mapValLength,
4  int combKeyLength, int combValLength,
5  char *prevWordFP, char *countFP) {
6    int laneID, kvsPerThread, warpID, ptr, gpu_val;
7    int high, kvCount, index, gpu_read, gpu_count;
8    //WARPS_IN_TB = number of warps in a threadblock
9    __shared__ char gpu_prevWord[WARPS_IN_TB][30];
10   __shared__ char gpu_word[WARPS_IN_TB][30];
11   combineSetup(kvsPerThread, &laneID, &warpID, &ptr,
12     &high, &kvCount, &index, size);
13   for(int i=0; i < 30; i++) { //init firstprivate data
14     gpu_prevWord[warpID][i] = prevWordFP[i];
15   }
16   gpu_count = countFP;
17   while(getKV(gpu_word, keys, &gpu_val, values, ptr,
18    high, indexArray, mapKeyLength, mapValLength)!= -1) {
19     if(strcmpGPU(gpu_word, gpu_prevWord[warpID],
20         mapKeyLength)==0){
21       gpu_count += gpu_val;
22     } else {
23       if(gpu_prevWord[0] != '\0') {
24        storeKV(gpu_prevWord[warpID], &gpu_count, &index,
25        combKeyLength, combValLength, opKey, opVal,
26        &kvCount);
27       }
28       strcpyGPU(gpu_prevWord[warpID], gpu_word,
29         mapKeyLength);
30       gpu_count=gpu_val;
31     }
32   }
33   if (gpu_prevWord[0] != '\0') {
34    storeKV(gpu_prevWord[warpID], &gpu_count, &index,
35    combKeyLength, combValLength, opKey, opVal, &kvCount);
36   }
37   finalCount[warpID]=kvCount;
38 }
```

in the thread registers. The compiler replaces calls to *scanf* that read in the KV pairs with a GPU-specific *getKV* call (line 18). The *keyin* and *valuein* clauses are necessary for this function. The original *printf* function for outputting the KV pairs is replaced by the *storeKV* function (lines 24, 34). *FinalCount* keeps track of the KV pairs emitted by each thread so that the final output can be combined and written back to the CPU.

While each thread in the warp executes the combiner function redundantly, for *getKV* and *storeKV* the threads perform vectorized loading and storing of KV pairs, respectively. This method loads/stores one element in an array key/value from one thread. It improves performance as it enables coalesced memory accesses. This same optimization is also performed on string functions, such as *strcpy* in the combiner kernels. Note that in order to dynamically switch between the vector and non-vector modes, all threads in the warp must be active for the entire code execution, making redundant execution in non-vectorizable code sections necessary. This redundant execution comes without any side-effects, owing to the warp-level SIMD model. If neither the key nor the value is an array, the compiler would generate a code where only a single thread per warp is active.

4.3 Host Code for Offloading

Since the GPU code execution is orchestrated by the host, the compiler must generate the necessary host code. Fig. 1 displays a flowchart for the structure of the generated code. First, the code copies the input fileSplit from HDFS into the GPU memory. A GPU kernel is then launched to collect and count the records in the input. Next, necessary storage is allocated on the GPU for map and combine kernels. To allocate the global KV store, all available GPU memory is used in the absence of the *kvpairs* clause. Otherwise, the global KV store memory allocation can be reduced, because the to-

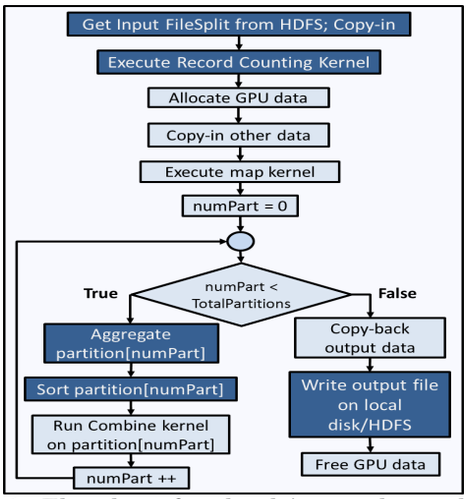


Figure 1: Flowchart for the driver code on the host CPU : Dark boxes indicate functions provided by the runtime system

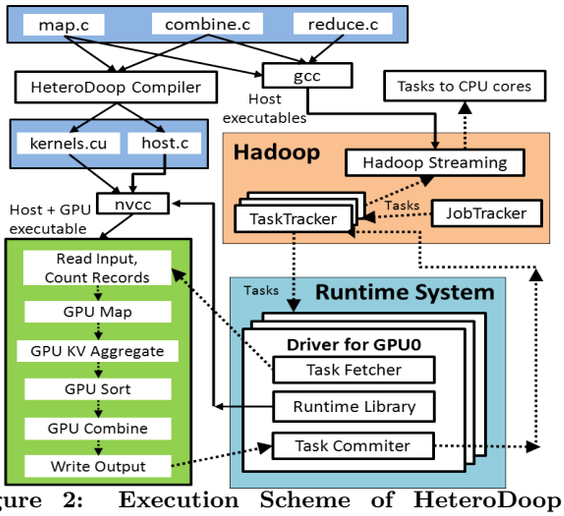


Figure 2: Execution Scheme of HeteroDoop : Dashed arrows represent the execution flow; solid arrows represent the compilation flow

tal number of records is already known. Each GPU thread in the map kernel generates KV pairs in its own portion of the global KV store. Note that each thread may not completely use its own portion of the global KV store, leading to *whitespaces*, which are the empty slots in the global KV store. This leads to a scattering of the KV pairs belonging to each partition. These KV pairs must be aggregated by removing the whitespaces before they are sorted in the next phase, reducing the sort size. The indirection array (*indexArray*) is useful for performing this aggregation as the KV pairs do not need to be shuffled directly. Note that smaller global KV store size results in better aggregation efficiency. Next, sort, followed by the combine kernel, are run on each partition. The generated output is written back to a local disk. For map-only jobs, the output is written directly to the HDFS. Finally, the allocated memories are freed.

5. OVERALL EXECUTION AND RUNTIME SYSTEM

This section describes the overall flow of the HeteroDoop framework and the runtime system. The runtime system assists the compiler-optimized code to run on a GPU by providing a number of library functions (Section 5.2), and helps implement the MapReduce semantics (Section 5.3).

5.1 Overall Execution

Fig. 2 shows the overall workflow in HeteroDoop and the roles of the runtime system. First, the user-written map and combine functions are translated by the HeteroDoop source-to-source compiler. Next, the generated CUDA code is compiled by the *nvcc* compiler for the GPU executable. The original source is compiled by *gcc* for the CPU executable. The executables are inserted into the Hadoop Streaming mechanism. When the execution starts, the JobTracker sends tasks to TaskTrackers, which schedule them either on the CPU cores, using native Hadoop Streaming, or on a GPU. The latter is coordinated by the *GPU driver* of the runtime system. The GPU driver fetches new tasks and invokes the GPU executables to perform map, combine and other MapReduce semantic-specific operations. Upon completion of a task, the GPU driver informs the TaskTracker about the task completion details, which comprise execution time, task log, etc.

Hadoop Integration and Fault Tolerance: Fault tolerance of HeteroDoop for GPU tasks manifests itself in two ways: i) a task failure is communicated to the Hadoop scheduler so that it can reschedule the task; ii) the failed GPU is revived so that future tasks can still be issued to it. To obtain this fault tolerance and to achieve issuing of tasks to GPUs, we have modified Hadoop's *MapTask* class implementation to signal a new task to be run on the GPU to the GPU driver. TaskTrackers on each slave keep one slot reserved per GPU. Note that these slots simply offload the tasks on GPUs; no CPU time is consumed. Scheduling decisions on the GPU are described in the next section. Internally, the driver runs one thread for each GPU on the node. The driver assures that only a single task runs on the GPU at a time. If a thread's execution fails, the error is communicated to Hadoop TaskTracker, and the driver restarts the thread. In this manner, the GPU driver is made fault tolerant.

5.2 Assisting GPU Execution

The following runtime library functions help facilitate the GPU kernel execution.

File Handling: HDFS is not POSIX API compliant for file handling; directly reading the input fileSplit from a C-code is not supported. The runtime system uses libHDFS [21], which provides a C/C++ function for fetching the input fileSplits from HDFS. The output of the map + combine execution is written to the local disk in a Hadoop-compatible binary format (*SequenceFileFormat*). For map-only jobs, the output is written directly to the HDFS.

Record Handling: The records in an input file must be pre-determined to support record stealing. The runtime system implements a GPU kernel that pre-determines and counts the records in the input file. This kernel is executed before the map kernel. The runtime system provides a thread-safe function, *getRecord*, which can be called by each thread of the map kernel to fetch a new record.

5.3 Implementing MapReduce Semantics

Performing Partition Aggregation: After the map execution, KV pairs in each partition of the global KV store

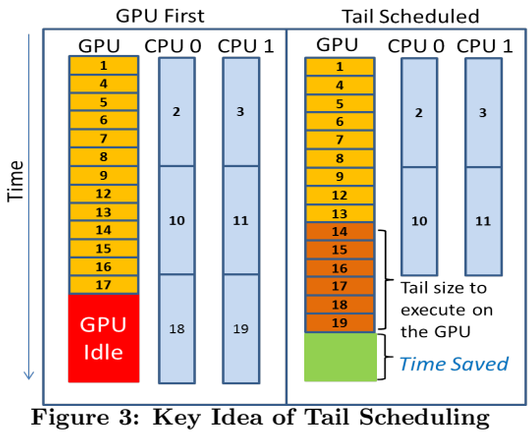


Figure 3: Key Idea of Tail Scheduling

must be compacted to get rid of the whitespaces, resulting in improved sorting efficiency. The runtime system provides an aggregation function that operates in parallel. It utilizes the count of the KV pairs emitted by each thread of the map kernel and an indirection array to locate KV pairs in the global KV store. A fast, parallel scan method for GPUs [22] is used to compute the prefix sum of the emitted KV pairs for each thread. From this information and the count of KV pairs emitted by each thread, a GPU kernel converts the old indirection array into a new one, wherein the generated KV pairs are aggregated.

Intermediate Sort: Recall that the MapReduce model sorts the KV pairs after the map operation. HeteroDoop uses a modified version of the efficient GPU merge sort implementation [23]. The original implementation operates on fixed, short-length integer keys and values, making efficient use of the GPU shared memory. For long keys and values, this method would make inefficient use of the shared memory and thus restrict the partial merge size. Our implementation therefore forgoes direct key-based sorting. We instead modify the original method [23] to employ indirection for accessing the KV pairs. The use of indirection avoids expensive movements of the KV pairs in the device memory. We chose this implementation over an alternative merge sort approach [24] for variable-length keys. This alternative approach hashes the first few characters of the variable-length keys, breaking ties by comparing the next characters. We chose not to use this method, since hashing requires additional memory on the GPU.

6. TAIL SCHEDULING

Hadoop's default scheduler does not take into account the disparity in the computational capabilities of the processors. When employing both CPUs and GPUs, task execution times vary substantially, requiring more advanced scheduling decisions. This section describes the HeteroDoop scheduler, aiming at optimal execution on accelerator clusters where each node has a CPU and one or more GPUs.

6.1 GPU-First Execution vs Tail Scheduling

A simplistic scheduler for CPUs and GPUs would act as follows. Whenever a new task is issued on a node, the task is scheduled on a GPU if such a device is free; otherwise, the CPU is chosen. We refer to this method as "*GPU-first*."

GPU-first scheduling keeps all CPU and GPU cores busy until the final tasks show up. Consider the example scenario in Fig. 3, which uses 1 GPU and a 2-core CPU for scheduling a total of 19 tasks. The GPU slot is 6x faster than the CPU slot. The figure shows the sequence number of each task. In GPU-first scheduling, shown on the left, the 1st task would go on the GPU and the 2nd and 3rd on the CPU. Since the GPU finishes early, the 4th task would go again on the GPU. Continuing in this manner, the 17th task goes on the GPU, while the 18th and 19th stay on the CPU. Since the CPU tasks are much slower, the faster GPU remains idle at the end stage, which is sub-optimal. A smarter scheme, as shown on the right side of Fig. 3, would force tasks 18 and 19 to execute on the GPU, saving on overall execution time. This is the key idea of tail scheduling.

A key challenge in realizing tail scheduling in HeteroDoop is for the slaves to know when their tails begin. As this information is not available to the slaves in the baseline Hadoop, in our algorithm, the JobTracker will estimate the number of remaining tasks for each node and communicate this information to the TaskTrackers, as described next.

Algorithm 2 Tail Scheduling on JobTracker and TaskTracker

```
1: function TailScheduleOnJT(numGPUs, maxSpeedup, numSlaves) //this function executes on the JobTracker before sending heartbeat response
2:   jobTail = numGPUs × maxSpeedup × numSlaves
3:   if (jobTail < getNumRemainingMaps()) then //identifying tail
4:     taskSet = scheduleNumGPUTasksAtMax()   //assures fairness
5:   else
6:     taskSet = useHadoopDefaultScheduling();
7:   end if
8:   numMapsRemainingPerNode = getNumRemainingMaps()/numSlaves
9:   heartBeatResponse.add(numMapsRemainingPerNode, taskSet)
10: end function
    //TaskTracker calculates GPU speedup over CPU for each task
11: function TailScheduleOnTT(task, numGPUs, aveSpeedup, numMapsRemainingPerNode)
12:   taskTail = numGPUs × aveSpeedup
13:   if (taskTail <= numMapsRemainingPerNode) then
14:     forceGPUexecution(task) //tail is forced on GPU
15:   else
16:     useGPUFirst(task);
17:   end if
18: end function
```

6.2 Tail Scheduling Algorithm

We implement tail scheduling in HeteroDoop at two levels: on the JobTracker and on the TaskTracker. Algorithm 2 describes the mechanism. The TaskTracker continually calculates the average GPU slot speedup over the CPU slot and informs the JobTracker. The Hadoop heartbeat mechanism has been modified to carry this information. The JobTracker remembers the maximum speedup from the TaskTrackers. For the TaskTracker, the tail size (*taskTail*) is the number of GPU tasks that execute in the same amount of time as one CPU task. It is computed as the number of GPUs on the current node (*numGPUs*) multiplied by the average GPU speedup (*aveSpeedup*) seen on that TaskTracker.

The JobTracker notifies each TaskTracker about the remaining number of map tasks to be scheduled (*numMapsRemainingPerNode*). As task scheduling on the JobTracker is on a first-come-first-serve basis, it does not exactly know how many tasks would go on a given TaskTracker. The JobTracker estimates *numMapsRemainingPerNode* as the total number of remaining tasks divided by the number of

Table 2: Description of the Benchmarks Used

Benchmark	%Exec. Time Map + Combine are Active	Nature (Intensiveness)	Combiner	Total Reduce Tasks		Total Map tasks		Input Size (GB)	
				Cluster1	Cluster2	Cluster1	Cluster2	Cluster1	Cluster2
Grep (GR)	69	IO	Yes	16	16	7632	2880	902	340
Histmovies (HS)	91	IO	Yes	8	8	4800	640	1190	159
Wordcount (WC)	91	IO	Yes	48	32	5760	1024	844	151
Histratings (HR)	92	Compute	Yes	5	5	4800	2560	591	160
Linear Regression (LR)	86	Compute	Yes	16	16	2560	3840	714	356
Kmeans (KM)	89	Compute	No	16	16	4800	NA	923	NA
Classification (CL)	92	Compute	No	16	16	4800	3200	923	72
BlackScholes (BS)	100	Compute	No	0	0	3600	5120	890	210

slaves (line 8). Whenever *taskTail* is greater than *numMapsRemainingPerNode*, *GPU-first* scheduling is used (lines 13–17). Otherwise, all the next tasks - across all slots of the TaskTracker - are forced for GPU execution. As all slots on a TaskTracker force their tasks on the GPU(s) once the *taskTail* begins, queuing might occur on the GPU(s). To counter this effect, the JobTracker only schedules at most *numGPUs* tasks on a TaskTracker per heartbeat once the *jobTail* begins. *JobTail* is the total number of tasks all GPUs in the cluster can finish in the same amount of time as a single CPU core. It is estimated as $numGPUs \times maxSpeedup \times numSlaves$, where *numSlaves* is the total number of slave nodes in the cluster and *maxSpeedup* is the maximum GPU speedup seen across the cluster. *ScheduleNumGPUTasksAtMax* replaces the original Hadoop scheduling scheme, which schedules tasks up to the number of empty GPU and CPU slots per heartbeat (lines 3–7).

7. EVALUATION

This section evaluates the HeteroDoop system. We present the overall performance achieved on our heterogeneous CPU-plus-GPU system versus a CPU-only scheme, i.e. baseline Hadoop. To analyze the performance in further detail, we show individual task speedups of GPU over CPU execution. We also present the performance benefits of HeteroDoop compiler optimizations on individual kernels.

Table 3: Cluster Setups Used

	Cluster1	Cluster2
#nodes	48 (+1 master)	32 (+1 master)
CPU	Intel Xeon E5-2680	Intel Xeon X5560
#CPU cores	20	12
GPU(s)	Tesla K40 (Kepler)	3×Tesla M2090 (Fermi)
RAM	256GB	24GB
Disk	500GB	none
Communication	FDR InfiniBand	QDR InfiniBand
Hadoop Version	Hadoop 1.2.1	Hadoop 1.2.1
CUDA Version	CUDA 6.0	CUDA 5.5
HDFS Block Size	256MB	256MB
HDFS Replication Factor	3	1
Max. Map Slots Per Node	20 (+1 for GPU runs)	4 (+1/GPU for GPU runs)
Max. Reduce Slots Per Node	2	2
Speculative Execution	Off	Off
% map tasks to finish before reduce starts	20	20

7.1 Benchmarks

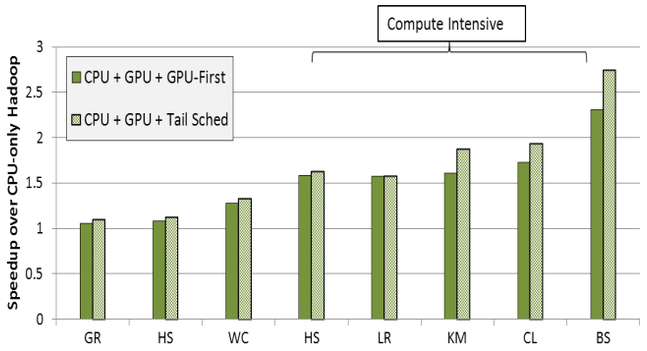
We used eight benchmarks for our experiments. *Grep*, *wordcount*, *kmeans*, *classification*, *histmovies* and *histratings* are taken from the PUMA benchmark suite [25]. They represent typical MapReduce applications, including both IO-intensive and compute-intensive ones. Apart from these, we evaluate two scientific computation applications, *blackScholes* and *linear regression*, that have been shown to be amenable to MapReduce [12, 3]. *Grep* and *wordCount* are well-known MapReduce applications, and both are IO intensive. *Kmeans* is an iterative clustering application. While *kmeans* is a centroid-based clustering algorithm, other variants such as distribution-based or density-based clustering are popular as well. All these clustering algorithms are known to be compute intensive. *Classification* benchmark is derived from statistical classification algorithms. It is similar to *kmeans*; however, there is no clustering involved. The application ends after classifying the input dataset to respective centroids in a single iteration. Histograms are common in big-data applications. *Histmovies* and *histratings* are two such applications. *Histmovies* averages the review ratings for each movie in a given dataset and puts these averages into bins. *Histratings* directly bins each review rating for all movies in the dataset. Since the combiner receives larger data to operate on, *histratings* becomes more compute intensive than *histmovies*. *BlackScholes* is a financial pricing model, with explicit parallelism. Other financial applications with similar workloads include *binomial options* and *SOBOL quasi random number generator*. *Linear regression* is commonly used in curve-fitting applications. For *blackScholes*, we ran 128 iterations per option. For *linear regression*, each input file contained data for 12 regressors, with 32 rows each. The details of the benchmarks, data sizes and tasks are presented in table 2.

7.2 Cluster Setups

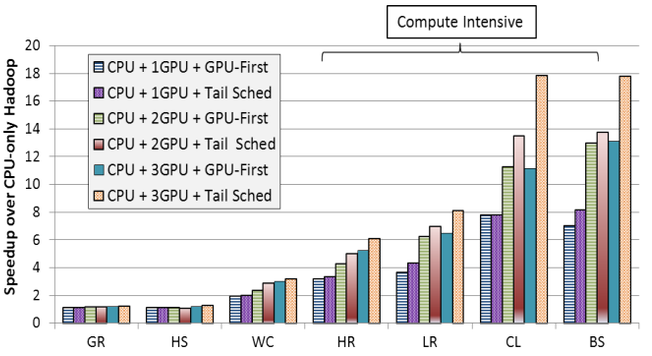
Table 3 shows the configurations of the two XSEDE [26] clusters used in this evaluation. Cluster1 is our primary platform. Each node has one GPU, which is a newer generation device. To evaluate the scalability of HeteroDoop on a multi-GPU system, we use Cluster2, which includes three older-type GPUs per node. Cluster2 is an in-memory system, i.e., there are no disks. Input, output, and temporary storage exist in the main memory of the Cluster2 nodes.

7.3 Overall Improvements

Fig. 4 shows the overall performance of HeteroDoop over the CPU-only Hadoop baseline. We ran each experiment three times, and report the best run. The variation across runs was low (<5%). The benchmarks are arranged by increasing speedup. As expected, HeteroDoop greatly outperforms CPU-only Hadoop on compute-intensive benchmarks owing to the GPU acceleration. Compute-intensive applications are therefore ideal candidates for HeteroDoop. For IO-intensive benchmarks, the use of CPUs brings about most of the achievable performance, indicating that execution on CPUs is essential in IO-intensive applications. In these benchmarks, HeteroDoop speedups are higher for Cluster2



(a) Performance gains of HeteroDoop normalized to CPU-only Hadoop on Cluster1



(b) Performance gains of HeteroDoop normalized to CPU-only Hadoop on Cluster2

Figure 4: Performance evaluation of HeteroDoop

than for Cluster1 because i) Cluster1 uses more CPU cores than Cluster2, and ii) Cluster2 is an in-memory system, reducing the IO-intensiveness of the applications. Fig. 4b shows multi-GPU scalability results of HeteroDoop. *KM* is not shown, as the memory requirement exceeds the capacity of Cluster2. The figures also show the effectiveness of *tail scheduling* over *GPU-first* scheduling. Note that tail scheduling improves performance only if the GPU(s) go idle at the end of the execution. For *LR* on Cluster1, this imbalance does not arise. For the IO-intensive benchmarks, the GPU speedup over a CPU core is low, resulting in only marginal benefits of HeteroDoop and tail scheduling.

7.4 Detailed Analysis

We present detailed analyses of the performance obtained on our primary cluster, Cluster1. Cluster2 showed similar trends. Fig. 5 shows the speedups obtained by a GPU task over a CPU task run by a single core, with the compiler-translated baseline code and with the optimizations. The optimizations include vectorization of map and combine kernels, using texture memory, record stealing and performing KV pair aggregation prior to sort. The GPU task comprises input copy, record counting, map, aggregate, sort, combine, and output write operations. The benchmarks are sorted by increasing speedups of their tasks. We attribute the increasing speedup to higher compute intensiveness. Each bar in the figure shows the speedup achieved by the baseline-translated code and the additional benefit of the optimizations. In *GR*, *KM*, *CL*, and *LR*, the optimizations gain substantial additional performance. Note that even for IO-intensive applications, such as *GR* and *HS*, the GPU achieves speedups over a single CPU core, and therefore, executing MapReduce tasks on GPUs along with CPU cores is still beneficial in these applications.

Fig. 6 breaks down the execution time for a single GPU task on each benchmark. Input reading time is the time for reading the HDFS file. Output writing time measures the time required for formatting the generated GPU output in Hadoop binary format, calculating the checksum, and writing the data on the HDFS/local disk, depending on whether or not the application is map-only. Fig. 6 shows that different stages of the MapReduce scheme can form a bottleneck on the GPU for different benchmarks. Note that the partition aggregation times are negligible in all benchmarks. Both these experiments (fig. 5, Fig. 6) made use of input data-local map + combine tasks. These figures indicate that a single-task speedup can be as high as 47x for

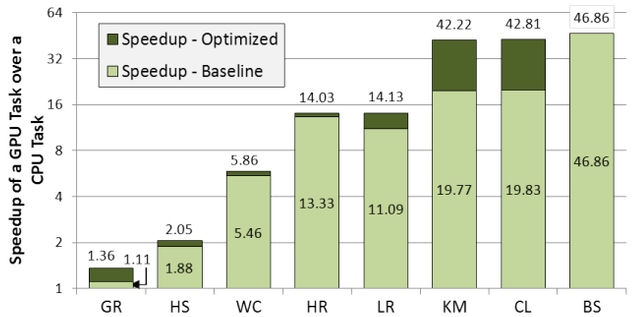


Figure 5: Speedup of a single GPU task over a CPU task : Optimizations have high impact on four benchmarks

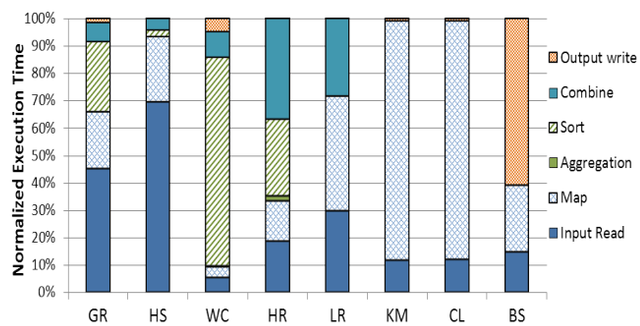
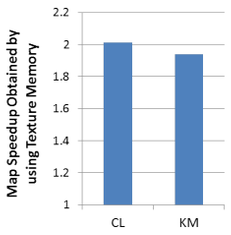


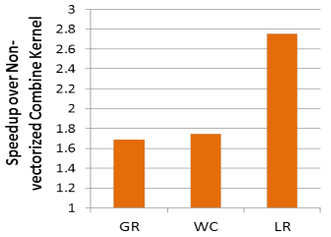
Figure 6: Execution time breakdown of a GPU task

BS, which is our most compute-intensive application. The GPU task of *BS* spends 62% of its execution time writing the output, which is up from 1% in a CPU task. Evidently, the high computational power of GPUs moves the bottleneck from computation to disk write. *CL* and *KM* are both map-heavy operations. *Wordcount* shows an interesting case where most of the execution time is spent in sorting since it emits many long-length keys. *HR* and *LR* spend substantial execution time in the combine operation.

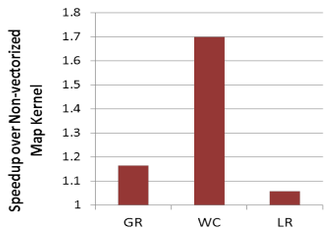
Fig. 7 shows the effects of individual optimizations. The first is the usage of texture memory, which can speed up the map kernel in *KM* and *CL* benchmarks by 2x (Fig. 7a). Vectorized read and write of the KV pairs in the combine kernel can improve this operation by 2.7x, as shown in Fig. 7b. Vectorized read and write in the map kernel can yield gains of up to 1.7x (Fig. 7c). A speedup of up to 1.36x is gained by the record stealing scheme on map kernels as shown in Fig. 7d. Aggregating KV pairs in each partition prior to



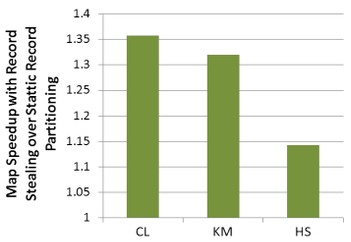
(a) Effects of using Texture Memory



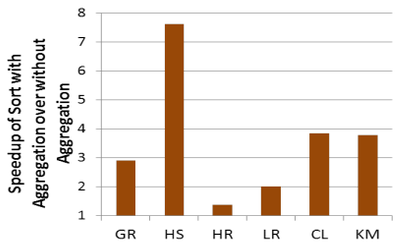
(b) Effects of Vectorized Read-Write on combine kernels



(c) Effects of Vectorized Read-Write on map kernels



(d) Effects of Record Stealing on map kernels



(e) Effects of KV Pair Aggregation on sort kernels

Figure 7: Effects of Optimizations : Only the benchmarks affected by the optimizations are shown

sorting can improve the sort kernel's performance by up to 7.6x (Fig. 7e).

8. RELATED WORK

We classify related work into the following five categories:

Single Node Systems: MapReduce has been studied as a programming model for several architectures. Two approaches [27, 28] target multi-cores. Catanzero et. al. [4] proposed a MapReduce system for GPUs. Mars [2] uses MapReduce as a programming model for a single GPU node. MapCG [14] offers a MapRedue API that is portable across CPUs and GPUs. But all these systems lack a distributed framework, and do not scale across a cluster.

Distributed Systems: GPMR [3] uses CUDA + MPI as a platform for writing applications, but does not employ CPUs in a cluster. As both CUDA and MPI require significant programmer expertise to organize the parallelism and manage communication, programmability of GPMR is low. MITHRA [12] is a system that shows the effectiveness of using Hadoop and GPUs together in two scientific applications. Unlike HeteroDoop, MITHRA relies on the programmer to write CUDA code for the application. It uses Hadoop Streaming as well, and utilizes only the GPUs in a cluster. Glasswing [13] uses either CPUs or GPUs for MapReduce. It requires the input map and reduce kernels to be written with OpenCL and offers a unified API to feed these kernels to the framework.

Employing both CPUs and GPUs: HadoopCL [29] presents a Java-based approach, where the programmer writes map and reduce functions in GPU-friendly classes. Java bytecode is translated into an OpenCL program. A key difference between HadoopCL and HeteroDoop is the parallelism exploitation: for a given fileSplit of a task running either on the CPU or GPU, HadoopCL launches many small, asynchronous OpenCL kernels. This leads to IO overheads, as the input data chunk of the fileSplit for each kernel must be copied separately into its buffer, and output must be written into a buffer which is shared across kernels, requiring serialization. Further, for multi-core CPUs, this strategy results in one task occupying all the CPU cores, and would require significant changes in the Hadoop scheduler when jobs have to share a cluster. Architecture-specific optimizations and CPU-GPU work partitioning decisions are not considered.

Scheduling Hadoop Tasks on a CPU-GPU Cluster: Shirahata et. al. [30] present a strategy where map tasks can be run on both CPUs and GPUs. The programmer is expected to provide the corresponding CPU/CUDA codes. This work employs a mathematical performance model for a heterogeneous scheduling strategy that minimizes the overall execution time for the given constraints. However, such minimization requires evaluation of the performance model throughout the job execution. By contrast, tail scheduling affects only the final tasks, reducing the scheduling overhead.

GPU-Specific MapReduce Optimizations: Chen et. al. present a system [31] for running MapReduce on CPU-GPU coupled architectures, where the CPU and GPU share the system memory. Another approach [32] presents a system that makes better use of the on-chip shared memory of the GPU for running reduction-intensive applications. Both these systems present important optimizations for MapReduce as a programming model on a single node. However, they are not directly applicable to distributed MapReduce since the map and reduce stages may run on different nodes. Also, GPU-shared-memory based optimizations for storing intermediate KV pairs would be precluded.

9. CONCLUSION

This paper has presented HeteroDoop, a MapReduce framework and system that employs both CPUs and GPUs in a cluster. It has introduced HeteroDoop directives, that can be placed on an existing sequential, CPU-only, MapReduce program for efficient execution on both the CPU and one or more GPUs on each node. The optimizing HeteroDoop compiler translates the directive-augmented programs into efficient GPU codes. HeteroDoop's runtime system assists in carrying out the compiler optimizations and handling MapReduce semantics on the GPUs. HeteroDoop is built on top of Hadoop, the state-of-the-art MapReduce framework for CPUs. The HeteroDoop runtime system plays

a vital role in this integration. A novel scheduling scheme optimizes the execution times of jobs on CPU-GPU heterogeneous clusters. Our experiments with HeteroDoop in eight benchmarks demonstrate that using a single GPU per node can achieve speedups of up to 2.78x, with a geometric mean of 1.6x, compared to a cluster running CPU-only Hadoop on 20-core CPUs. The proposed tail scheduling scheme works well for intra-node heterogeneity. We leave handling of extreme inter-node heterogeneity to future work, where the trade-off between data locality and execution speed will be an important additional consideration.

Acknowledgements

We thank T. N. Vijaykumar and anonymous reviewers for their valuable feedback that helped improve this paper. This work was supported, in part, by the National Science Foundation under grants no. 0916817-CCF and 1449258-ACI. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant no. ACI-1053575. We also thank NVIDIA corporation for the equipment grant.

10. REFERENCES

[1] Apache, "Hadoop." [Online]. Available: http://hadoop.apache.org/. (accessed December 30, 2014).

[2] 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*, PACT '08, pp. 260–269, ACM, 2008.

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

[4] B. Catanzaro, N. Sundaram, and K. Keutzer, "A map reduce framework for programming graphics processors," in *In Workshop on Software Tools for MultiCore Systems*, 2008.

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2010.

[6] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, (New York, NY, USA), pp. 59–72, ACM, 2007.

[7] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, (Berkeley, CA, USA), pp. 1–14, USENIX Association, 2008.

[8] NVIDIA, "CUDA." [Online]. Available: http://bit.ly/1xnvPXr, 2015. (accessed January 5, 2015).

[9] OpenCL, "OpenCL." [Online]. Available: http://www.khronos.org/opencl/. (accessed Jan. 4, 2015).

[10] OpenACC, "OpenACC: Directives for Accelerators." [Online]. Available: http://bit.ly/1Bx3pWk. (accessed June. 11, 2014).

[11] OpenMP, "OpenMP: Version 4.0." [Online]. Available: http://bit.ly/1Bx3IjT, 2013. (accessed July 31, 2014).

[12] R. Farivar, A. Verma, E. Chan, and R. H. Campbell, "Mithra: Multiple data independent tasks on a heterogeneous resource architecture.," in *CLUSTER*, pp. 1–10, IEEE, 2009.

[13] I. El-Helw, R. Hofman, and H. E. Bal, "Scaling mapreduce vertically and horizontally," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, (Piscataway, NJ, USA), pp. 525–535, IEEE Press, 2014.

[14] 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*, PACT '10, (New York, NY, USA), pp. 217–226, ACM, 2010.

[15] 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*, OSDI'08, (Berkeley, CA, USA), pp. 29–42, USENIX Association, 2008.

[16] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "Tarazu: Optimizing mapreduce on heterogeneous clusters," *SIGARCH Comput. Archit. News*, vol. 40, pp. 61–74, Mar. 2012.

[17] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st ed., 2010.

[18] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, (Washington, DC, USA), pp. 1–10, IEEE Computer Society, 2010.

[19] "Hadoop streaming." [Online]. Available: http://bit.ly/1Hcl7Rg. (accessed January 02, 2015).

[20] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *IEEE Computer*, vol. 42, no. 12, pp. 36–42, 2009.

[21] libHDFS, "API http://wiki.apache.org/hadoop/LibHDFS."

[22] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for gpu computing," in *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, (Aire-la-Ville, Switzerland, Switzerland), pp. 97–106, Eurographics Association, 2007.

[23] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–10, May 2009.

[24] A. Davidson, D. Tarjan, M. Garland, and J. Owens, "Efficient parallel merge sort for fixed and variable length keys," in *Innovative Parallel Computing (InPar)*, pp. 1–9, May 2012.

[25] F. Ahmad, S. Lee, M. Thottethodi, and T. N. Vijaykumar, "Puma: Purdue mapreduce benchmarks suite," Tech. Rep. Paper 437, School of Electrical and Computer Engineering, Purdue University, West Lafayette, Indiana, October 2012.

[26] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkens-Diehr, "Xsede: Accelerating scientific discovery," *Computing in Science and Engineering*, vol. 16, no. 5, pp. 62–74, 2014.

[27] W. Jiang, V. T. Ravi, and G. Agrawal, "A map-reduce system with an alternate api for multi-core environments," in *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID, (Washington, DC, USA), pp. 84–93, IEEE Computer Society, 2010.

[28] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, (Washington, DC, USA), pp. 13–24, IEEE Computer Society, 2007.

[29] 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*, IPDPSW '13, (Washington, DC, USA), pp. 1918–1927, IEEE Computer Society, 2013.

[30] K. Shirahata, H. Sato, and S. Matsuoka, "Hybrid map task scheduling for gpu-based heterogeneous clusters," *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 0, pp. 733–740, 2010.

[31] 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*, SC '12, (Los Alamitos, CA, USA), pp. 25:1–25:11, IEEE Computer Society Press, 2012.

[32] 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*, HPDC '12, (New York, NY, USA), pp. 199–210, ACM, 2012.