Design and Optimization of a Big Data Computing Framework based on CPU/GPU Cluster

Yanlong Zhai*, Ying Guo*, Qiurui Chen†, Kai Yang† and Emmanuel Mbarushimana*
*Beijing Engineering Research Center of Massive Language Information Processing and Cloud Computing Applications
School of Computer Science, Beijing Institute of Technology, Beijing, China 100081
Email: ylzhai@bit.edu.cn
†Beijing Simulation Center, Beijing, China
Email: yangkaigege@163.com

***Abstract*—Big data processing is receiving significant amount of interest as an important technology to reveal the information behind the data, such as trends, characteristics, etc. MapReduce is one of the most popular distributed parallel data processing framework. However, some high-end applications, especially some scientific analyses have both data-intensive and computation-intensive features. Therefore, we have designed and implemented a high performance big data process framework called Lit, which leverages the power of Hadoop and GPUs. In this paper, we presented the basic design and architecture of Lit. More importantly, we spent a lot of effort on optimizing the communications between CPU and GPU. Lit integrated GPU with Hadoop to improve the computational power of each node in the cluster. To simplify the parallel programming, Lit provided an annotation based approach to automatically generate CUDA codes from Hadoop codes. Lit hid the complexity of programming on CPU/GPU cluster by providing extended compiler and optimizer. To utilize the simplified programming, scalability and fault tolerance benefits of Hadoop and combine them with the high performance computation power of GPU, Lit extended the Hadoop by applying a GPUClassloader to detect the GPU, generate and compile CUDA codes, and invoke the shared library. For all CPU-GPU co-processing systems, the communication with the GPU is the well-known performance bottleneck. We introduced data flow optimization approach to reduce unnecessary memory copies. Our experimental results show that Lit can achieve an average speedup of 1x to 3x on three typical applications over Hadoop, and the data flow optimization approach for the Lit can achieve about 16% performance gain.**

I. Introduction

Throughout the past decade, there has been an unprecedented growth of data as information is being continuously and rapidly generated. It is expected that the data volumes processed by applications will cross the zetta-scale threshold. Timely and cost-effective process such "big data" is increasingly prevalent because "big data" is not only a matter of data size, but also an opportunity to answer questions that were previously considered beyond your reach or to find out more information behind the data, such as trends, characteristics, etc. MapReduce framework is considered as the most effective way for such big data analytics due to its high scalability and the ability of parallel processing of non-structured or semi-structured data[1]. More and more applications are relying on MapReduce framework to process their constantly increasing data. Therefore, a set of MapReduce based algorithms are implemented for different areas, such as astronomy, particle physics, bioinformatics, etc.

A class of emerging high-end applications need to perform efficient data computing on massive datasets. This kind of applications require the data processing technology to have both data-intensive and computation-intensive abilities. However, MapReduce framework is not originally designed for high performance scientific applications. The MapReduce cluster is composed of cheap commodity machines, that do not have enough computational power. To address the data-intensive and computation-intensive requirement, many application scientists have initiated efforts to integrate data-intensive computing into computational-intensive HPC facilities. Due to the differential of architecture and semantics, it is difficult to integrate data-intensive computing technologies into current HPC clusters. Another way to approach this problem is to increase the computation power of stat of the art data-intensive processing frameworks, like Hadoop MapReduce. One straightforward idea is to apply some accelerators to these data processing frameworks. General purpose computation on GPUs, or GPGPU, has emerged in various HPC application domains, such as bioinformatics [2],medical imaging[3], embedded system design[4], machine learning[5], data mining[6], and so on. Therefore, GPU is the ideal choice to be the accelerator for massive data processing framework. Although some effort has been expended to integrate GPU with MapReduce, there are still lots of challenges to conquer in both architecture and algorithm aspects.

In this paper, we present a CPU/GPU cluster based high performance massive data computing framework called "Lit", which leverages the computational power of GPUs and the parallel data processing ability of MapReduce. Our aim in this work is to utilize the simplified programming, scalability and fault tolerance benefits of the most popular MapReduce framework–Hadoop and combine them with the high performance computational power of GPU. The framework is composed of a modified Java compiler and an extended Hadoop runtime. A set of directives is firstly designed for the programmers to annotate the source code. After annotating the potential parallel sections in the source code using these directives, the compiler translates the annotated part into CUDA codes. At the same time, some optimization techniques are performed to optimize the data transfer between CPU and GPU. The most unique characteristics of our work is that, different from other GPU based MapReduce frameworks, Lit is fully compliant with Hadoop MapReduce programming model at the source code level. All the programmers need to do is to add some predefined comments to the source code. This

978-0-7695-5088-6/13 $31.00 © 2013 IEEE
DOI 10.1109/HPCC.and.EUC.2013.147

makes it much easier to transfer those massive data processing applications to our high performance processing platform.

II. Related Works

In this section, we introduce some related work on data-intensive high performance computing and compare some GPU based MapReduce frameworks with Lit. Optimization approaches for heterogeneous programming models are also discussed.

A. *GPU based MapReduce frameworks*

On the observation that MapReduce programming model has good scalability, many GPU based MapReduce frameworks have been proposed to harness the power of GPUs. Table I shows the differences of these frameworks. Mars[11] is the first proposed GPU based MapReduce framework. It yields up to 16 times performance improvement over Phoenix[12], a MapReduce framework on multi-core CPUs. Mars hides the programming complexity of the GPU by the simple and familiar MapReduce interface. However, it is not designed for data-intensive applications and does not support GPU cluster. MapCG[13] proposed a MapReduce framework to provide source code level portability between CPU and GPU. Grex[14] is another MapReduce framework implemented on single GPU. Compared with Mars and MapCG, Grex provided some optimization methods like parallel input splitting, evenly data distribution and memory management scheme, thus gain up to 12.4x and 4.1x performance speedup over Mars and MapCG. The most similar work with Lit is Mithra[15], which is also using Hadoop as a backbone to cluster GPU nodes for MapReduce applications. However, Hadoops Map merely acts as a distribution mechanism for the workload across the nodes of the clusters. Programmers still need to write the CUDA kernel to work on the GPU. As we can see from Table I, although there are some GPU based MapReduce frameworks, none of them can support all the features required by data-intensive and computation-intensive applications. Lit is the only one who leverages Apache Hadoop along with NVIDIA CUDA technology to produce scalable high performance gains using the MapReduce programming model.

B. *CPU-GPU System Optimization*

As shown in the previous section, more and more CPU-GPU based MapReduce frameworks are proposed for data-intensive and computation-intensive applications. Modern G-PUs are very efficient at manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel. GPUs can be regarded as massively parallel processors with faster computation and higher memory bandwidth than CPUs. However, the device memory has both high bandwidth and high access latency. For example, the NVIDIA C2050 GPU which used in our experimental has an access latency of 400-800 cycles, and the peak memory bandwidth between the device memory and the multiprocessors is around 177 GB/second. Except the access latency, the communications used to exchange data via PCIe bus is generally considered to be the most important bottleneck for CPU-GPU co-processing systems. Figure 1 shows the memory hierarchy of NVIDIA Fermi GPUs. In the Fermi memory architecture, each SM has 32KB of registers with very high speed of 8,000 GB/s. Each thread has access to its own registers and not those of other threads. The Fermi architecture provides also local memory in each SM, and has the ability to use some of this local memory as a first-level (L1) caches for global memory. This 64 KB memory can be configured as either 48 KB of shared memory with 16 KB of L1 cache, or 16 KB of shared memory with 48 KB of L1 cache. Shared memory is accessible by the threads in the same thread block. It has high speed access (10-20 cycles) and very high bandwidth (1,600 GB/s) to moderate amounts of data. Each Fermi GPU is also equipped with an L2 cache with 768KB that services all load and store from/to global memory, including copies to/from CPU host, and also texture requests.

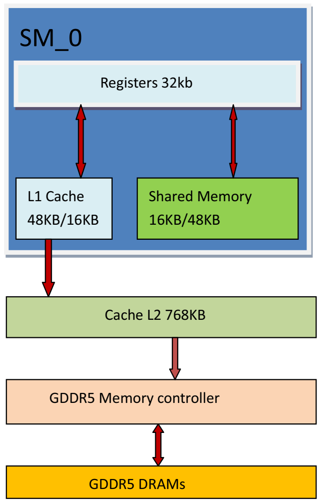


Fig. 1. Nvidia Fermi Memory Hierarchy

With the release of several GPU programming languages, including NVIDIA CUDA[16], Brook+, OpenCL[17], etc., the general purpose programming on GPU is improving rapidly. Therefore, more and more effort is being dedicated for optimizing the codes and the communication between CPU and GPU. CGCM[18] is the the first fully automatic system for managing and optimizing CPU-GPU communication. Automatically managing and optimizing communication increases programmer efficiency and program correctness.It also improves the applicability and performance of automatic GPU parallelization. CGCM out-performs inspectorCexecutor systems on 24 programs and enables a whole program speedup of 5.36x over best sequential. DSM system proposed by Baojiang[19] provides a logic shared memory space and manages data communication between CPUs and GPUs. They have shown to us how to optimize the DSM system using a compiler-assisted data-prefetching scheme. From data prefetching using the NPB SP benchmark, SP achieves 1.25 speedup on a 4-core Intel Xeon Linux system with one Nvidia GTX 285 and a Tesla C1060. Mehdi[20] presented an automatic ,static program transformation that schedules and generates efficient memory transfers between a computer host and its hardware accelerator, addressing a well-known performance bottleneck. They implemented this transformation as a middle-end compilation pass in the PIPS / PAR4ALL compiler. They

TABLE I. COMPARISONS OF THE GPU-BASED MAPREDUCE FRAMEWORKS

Property	Lit[7]	Grex	Mars	MapCG	Mithra	GPMR[8]	Ji[9]	Chen[10]
Load balancing	no	yes	no	no	no	yes	no	no
Fault tolerant	yes	no	no	no	yes	no	no	no
GPU Cluster	yes	no	no	yes	yes	yes	no	no
Optimization	yes	no	no	no	no	yes	yes	yes
Code partition	yes	yes	no	no	no	no	no	no
Hadoop suport	yes	no	no	no	yes	no	no	no

obtained an average speedup of 4 to 5 when compared to naive parallelization using a modern GPU with PAR4ALL,HMPP, and PGI and 3.5 when compared to an OPENMP version using a 12-core multiprocessor.

III. LIT FRAMEWORK

In this section, we present the design and implementation of Lit framework. Our design is guided by the following three goals.

1) Compliance with Hadoop MapReduce programming model. This will encourages massive-data application developers to use the GPU to improve the performance.
2) Simplified programming. Our programming framework will use original Java language and leverage the parallel programming concepts from OpenMP. All the programmers have to do is to add some pre-defined annotations to the MapReduce program.
3) High performance. The overall performance of program on our framework should be comparable to the CUDA program and better than MapReduce program on CPU clusters.

A. Architecture Overview

Lit is designed to excel at executing massive data, high performance computing tasks. Using Hadoop as a backbone, the architecture of Lit is similar as typical Hadoop cluster, composed of HDFS and extended MapReduce runtime. Furthermore, a modified Jikes Java compiler is designed for the code generation and optimization. In the Lit cluster, there is one dedicated front-end machine that acts as the HDFS name node and MapReduce Job-tracker. A number of back-end machines act as the HDFS data node(or compute node) and MapReduce Task-tracker, which run the map and reduce tasks. Different with the typical Hadoop cluster, each compute node in Lit equips a nVIDIA GPU, so that these compute nodes can run both CPU and GPU MapReduce tasks. Note that our design does not require homogeneous compute nodes in a cluster, however we expect this to be the common case. In our system, if the compute node does not have GPU, the job-tracker will not schedule a GPU task to this node. Instead, the job-tracker will distribute a CPU task to the node. According to the MapReduce model, the reduce tasks will not start until all the map tasks are completed. So the drawback of a heterogeneous cluster is that the execution time of the Map phase will be the most long running map task, which most likely is a CPU map task. Figure 2 shows a high level view of the system architecture.

Same as the Hadoop MapReduce framework, our system has two stages, Map and Reduce. Always user should firstly upload the input data to HDFS. HDFS will divide the input

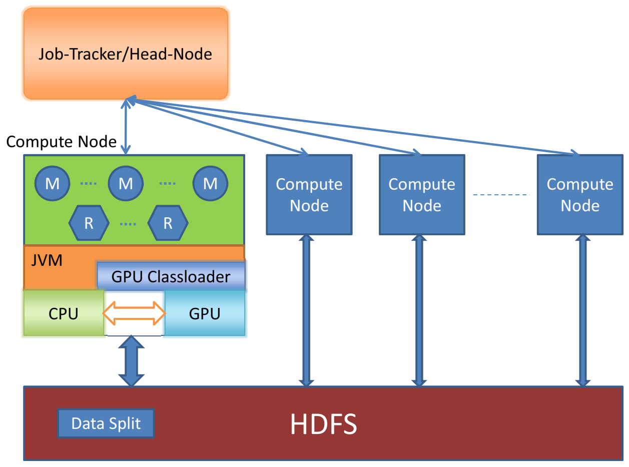


Fig. 2. System Architecture

data into blocks according to the HDFS configuration. After the user submit the job to the JobTracker, the JobTracker will launch some map tasks on the compute nodes, where the required input blocks located.

In the Map stage, a input split operator divides the input data into multiple splits. Then a RecordReader will read a split and parse the split into key/value pairs. These key/value pairs are the input for the map function. The TaskTracker will start a JVM to run the map function. In our framework, we add some pre-defined annotations to the source code to indicate some of the codes will be executed on the GPU. If the map function does not contain any of these annotations, the map function will be executed as a normal Hadoop Map task. Otherwise, the TaskTracker will start a translation process to translate the Java code to CUDA code and compile the CUDA code on-the-fly. The translation process is managed by a new defined Java class loader, called GPUClassLoader. The GPUClassLoader is also responsible for the memory management and optimization. It will explicitly copy the required data from main memory to the graphics cards memory. After the computation on the GPU, the results will be copied back to the main memory to continue the rest of the map function. After the Map stage is finished, the intermediate key/value pairs will be sorted so that the pairs with the same key are stored consecutively. The Reduce stage of our framework is similarly as the Map stage, if the Reduce function is annotated, it will call the GPU to do the computation, otherwise, it will act as a normal Hadoop Reduce function.

B. Lit Workflow

In this section, CUDA code generation workflow is described. Figure 3 shows the CUDA code generation and execution workflow.

Firstly, a programmer creates a Java MapReduce program annotated by some directives. These directives indicate some code or functions can be executed in parallel. This code is translated by the Java compiler to annotated bytecode. Because the target GPU hardware is not known at compile time, our framework must dynamically generate CUDA code at run-time. This can be accomplished by three different ways: JIT compiler; an extended Java class loader or Java annotation(requires Java 5.0 or later). For the adaptiveness of different Java versions and simplification of implementation, we chose Java class loader based approach in the framework. When the map or reduce task is distributed to the compute-node, the Task-Tracker will start a sub-JVM to run the map or reduce task. As described in the previous section, we defined the GPUClassloader to manage the code translation process. So,we extend the Hadoop MapReduce framework to start the sub-JVM with some necessary parameters to use the GPUClassloader instead of the regular class loader. The GPUClassloader will check the bytecode to see if it contains some GPU directives. If the bytecode does not contain any of these directives, the GPUClassloader will load the bytecode as a normal java class. Otherwise, the GPUClassloader will translate the annotated part into CUDA source code. The CUDA source code then be compiled on the fly into shared library and invoked by the GPUClassloader. Finally, the computing results generated on the GPU will be transferred back to the host machine and combined with the results generated on the CPU. For details and examples of the CUDA code generation, please refer to our previous paper[7].

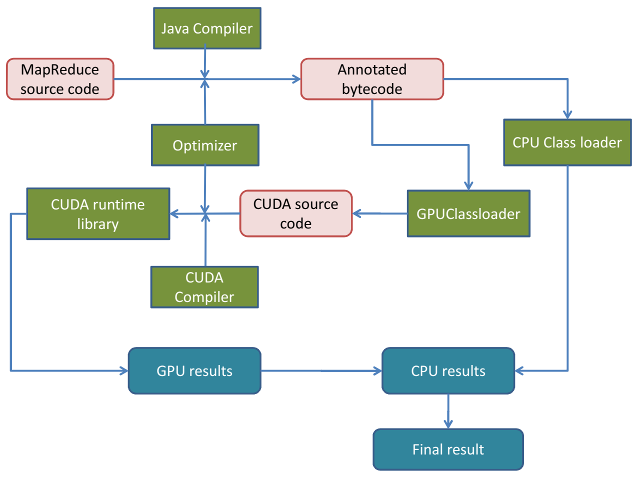


Fig. 3. Execution flow of Lit

C. Directives Design

Inspired by the mechanism of OpenMP execution flow, we designed this annotation based code generation approach. OpenMP is efficient at expressing loop-level parallelism, which is an ideal target for source code parallel translation. Therefore, the design of the directives used for annotating parallelable source code is very important.

For simplicity, we only define some most frequently used directives as shown in Table II. We leverage the directive translation approach discussed in [21] and will not go into detail in this paper.

TABLE II. DEFINED DIRECTIVES

Directive	Semantic	Translation
gmp parallel	Specify parallel regions in the source code.	One or more CUDA function calls.
gmp parallel for	Specify a paralleable for loop. Each iteration of the loop is assigned to a GPU thread.	One CUDA kernel function.
gmp sync	Request a global synchronization in the CUDA programming model.	Will splite the parallel region into two regions.
gmp shared	The data will be shared by all GPU threads.	The data will be copied to global memory and copied back using *cudaMemcopy*.
gmp copyin	The data will be copied into the GPU memory.	*cudaMemcopy* function call with parameter cudaMemcpyHostToDevice.
gmp copyout	The data will be copied out from GPU memeory.	*cudaMemcopy* function call with parameter cudaMemcpyDeviceToHost.

IV. OPTIMIZATION

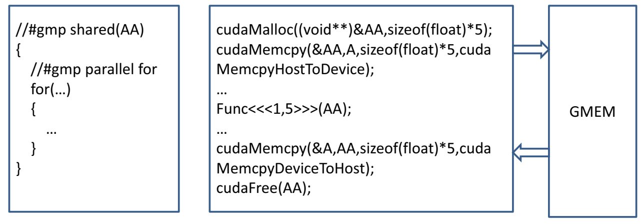
Currently, we perform optimization on two aspects of our framework. One is to optimize the code generation and execution workflow. The other is to optimize the memory copy between CPU and GPU.

A. Workflow Optimization

In order to integrate the GPU with Hadoop MapReduce, we have designed a new class loader called GPUClassloader. This class loader will be loaded when the machine has GPU and CUDA. At run-time, the GPUClassloader will call the CUDA compiler to generate the run-time library, so that the native function can be executed on the GPUs. However, this compilation process introduce some overhead. Because it will store the generated CUDA source code to disk, invoke the CUDA compiler to compile the source code, generate the dynamic library and load the share library into the JVM. We observe that, by default, Hadoop will lunch a new JVM for each Map or Reduce task. This could take some time for the JVM to startup. So we could leverage this period of time for compiling and building the CUDA library. Therefore, to reduce the compilation overhead, we modify the Hadoop to start a new thread when lunching the JVM for Map or Reduce tasks. This thread will compile all the parallel regions in one go, so that compilation, assembling and linking are shared.

B. Memory Copy Optimization

As we discussed in Section II-B, the communication between CPU and GPU is generally considered to be the most important bottleneck for CPU-GPU co-processing systems. The reason is that, to copy memory (via DMA) to and from the GPU over the PCIe bus involves expensive context switches that reduce the available bandwidth considerably. Therefore, the optimization of the communications between CPU and GPU becomes very important for CPU-GPU co-processing system. In Lit framework, the CPU-GPU communications are explicitly annotated by directives in the source code. The directive *gmp shared* is designed in this framework for identifying the data to be transferred from main memory to GPU memory and transferred back after computation. This directive will be translated to relevant memory transfer calls, like *cudaMalloc,cudaMemcpy,cudaFree*, etc. A basic transfer strategy is to copy all the shared data to the GPU memory, and copy back the shared data that are modified by kernel functions. However, not all shared data are required by the CPU after the kernel completes. And also, not all shared data in the GPU global memory are persistent across kernel calls. Therefore, we designed two other data management directives, *gmp copyin* and *gmp copyout*, to identify the single direction data movements. If the data is marked *gmp copyin*, the data needs only to be copied from host to GPU but not back again. If the data is marked *gmp copyout* the data needs only to be copied out of the GPU, never into the GPU. Although mark all the data by these two directives can significantly reduce unnecessary data transfers, it induce much complexity to the programming. So we adopt some compiler optimization techniques to analyze the data flow and eliminate redundant data transfers. Based on our previous research on the data flow optimization of parallel language[22], we find that data flow analysis and data flow optimization technique can be used to optimize the data transfer in our case. Data flow analysis is a versatile technique that can be used to address a variety of static program analysis problems. Following is an illustration example. In Figure 4, the programmer annotated an array of floating numbers as *gmp shared*, the array *AA* then be transferred to the GPU by some memory management functions. The programming is very easy for programmers, but the communication may not be efficient. If the array AA will not be used in the succeeding code, it should not be copied back to the main memory. Based on this observation, the memory optimization goal will be twofold: (1)to eliminate unnecessary memory copies; and (2) to reduce the bytes transferred in the communication.



```
//#gmp shared(AA)
{
   //#gmp parallel for
   for(...)
   {
      ...
   }
}
```

```
cudaMalloc((void**)&AA,sizeof(float)*5);
cudaMemcpy(&AA,A,sizeof(float)*5,cuda
MemcpyHostToDevice);
...
Func<<<1,5>>>(AA);
...
cudaMemcpy(&A,AA,sizeof(float)*5,cuda
MemcpyDeviceToHost);
cudaFree(AA);
```

Fig. 4. Memory copy optimization

Our optimization algorithm is based on the copy propagation algorithm[23]. Copy propagation algorithm will propagates the right argument of an assignment statement to the places where the left argument is used. It is used to optimize the data flow by eliminating some unnecessary assignment statements in the program. Following is the formalized precondition of this algorithm. For an assignment $x = y$, we can substitute y for x at every use u_x of x if: 1) the assignment $x = y$ is the only definition of x reaching u_x, and 2) there are no assignments to y between $x = y$ and u_x in any program execution Firstly, we build the DAG representation of the program. Each kernel region is considered as a basic block. Then, we calculate the data flow functions and the Use-Definition(UD) chains. The UD chains and data flow information, like IN and OUT sets, are the input for the copy propagation algorithm. After the optimization, all the data that has no references in the kernel region will not be copied to the GPU memory, and all the data defined or modified in the kernel region has no further references after the kernel region will not be copied back to the CPU memory. Our experimental results show that this data flow optimization is able to reduce memory transfers sufficiently. Figure 5 shows a copy propagation example. Assignment $x = y$ in block $B2$ is a definition of variable x. This definition has two references in block $B3$ and $B4$. After the copy propagation, the assignment $x = y$ has been deleted as an unnecessary statement. In CPU-GPU co-processing systems, the memory copies between CPU and GPU can be considered as variable assignments. So the reduction of assignments can efficiently reduce the memory communication.

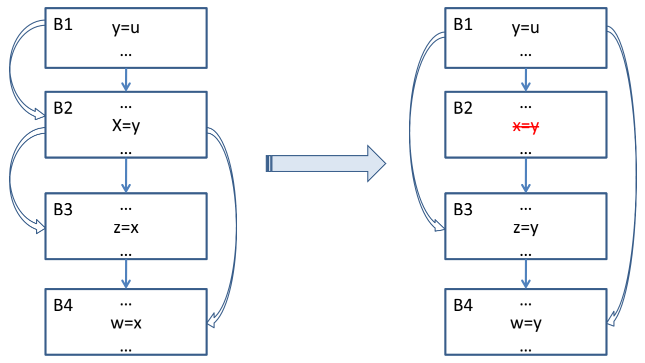


Fig. 5. Copy propagation optimization

The basic steps for data flow optimization are: (1)represent the program as a CFG(Control flow graph);(2)build the data flow equations; (3)calculate the data flow equation to get the IN and OUT sets; (4)perform the optimization algorithm. The data flow information is represented as four frequently used data flow sets:
— IN is the set of definitions that reach the beginning of a node
— OUT is the set of definitions that reach end of node
— GEN is the set of definitions generated within the node and reach the end of the node
— $KILL$ is the set of definitions outside of the node that also have definitions within the node(killed by the node).

The generated code in Lit framework can be divided into two parts, Java based sequential part and CUDA based parallel part. For the sequential part, the way to calculate the IN set is to solve following data flow equations for each node in the CFG.

$$OUT[n] = GEN[n] \cup (IN[n] - KILL[n]) \quad (1)$$
$$IN[n] = \cup_{p \in pred(n)} OUT[p] \quad (2)$$

Equation (1) is the universal data flow equation, which means the information at the end of the node n is the information generated in n or information reach the entry of n and not killed in n. $pred(n)$ is the set of immediate predecessors of n. Equation(2) means the information at the entry of the node n is the union of information at the end of all predecessor nodes of n.

For the parallel part, the way to calculate the IN set is different from the sequential part, because CUDA is a parallel language with barrier synchronization. So the computation is based on following fundamental concepts:

- Information in the GPU global memory can only be synchronized at the explicit barrier synchronization point.
- A definition is reachable at a point p in a parallel branch i if it is available at the end of it's previous synchronization point and is not killed in any block that executed before point p in branch i.
- If the definition is not serially reachable at point p, then it can still be available at point p if
  1) the definition is serially available at some point q in another parallel branch j, and
  2) q will be executed before p, and
  3) the definition is not killed by any other parallel branch by a statement to be executed before p.

Accordingly, the equation to calculate IN set of nodes in parallel branches is composed of two parts. The first part of the equation is $s\_in[n] - \cup_{p \in pred_{sy}(n)} AKillOut[p]$, which is used to calculate the definitions that are serially available and is not killed in any of the previous synchronization points. $s\_in[n]$ represent the serial IN set and is supposed to be computed previously in sequential data flow analysis. $pred_{sy}(n)$ is the set of immediate synchronization predecessors of n. The $AKillOut[n]$ set accumulates the definitions that must have been killed since the start of the parallel branch. The data flow equations to calculate the $AKillOut$ set is defined as follows.

$$AKillOut[n] = AKillIn[n] \cup KILL[n] \quad (3)$$
$$AKillIn[n] = \cup_{p \in pred_{all}(n)} AKillOut[p] \quad (4)$$

$pred_{all}(n)$ is the set of all predecessors, which includes serial predecessors and synchronization predecessors. The second part of the equation is $\cup_{p \in pred_{sy}(n)} OUT[p] - \cup_{p \in pred_{sy}(n)} AKillOut[p]$, which is designed to calculate the definitions that are not serially reachable but can still be available via the parallel and synchronization edges. Overall, the data flow equations to calculate the IN sets are summarized as follows:

$$\begin{aligned} OUT[n] &= GEN[n] \cup (IN[n] - KILL[n]) && (5) \\ IN[n] &= (s\_in[n] - \cup_{p \in pred_{sy}(n)} AKillOut[p]) \\ &\quad \cup (\cup_{p \in pred_{sy}(n)} OUT[p] - \\ &\quad \cup_{p \in pred_{sy}(n)} AKillOut[p]) && (6) \end{aligned}$$
$$AKillOut[n] = AKillIn[n] \cup KILL[n] \quad (7)$$
$$AKillIn[n] = \cup_{p \in pred_{all}(n)} AKillOut[p] \quad (8)$$

After calculating the data flow equations, the data flow sets are annotated on the CFG.

Figure 6 is the algorithm to optimize the data flow of Lit program, which extends the typical copy propagation algorithm. The purpose of a typical copy propagation algorithm is to eliminate unnecessary statements. So it will not substitute the definition to its references if it is not a unnecessary statement. In our circumstances, the propagation itself can optimize the data flow of the program. So we will substitute all the definition to its references.

Algorithm 1 Data flow optimization

```
Require: Let G be the CFG represented program; U is the set
of all assignment statements; use[s] is the set of statements
reference the definition s.
Input: CFG G and in set for each node of G.
Output: Optimized CFG G'
01: foreach assignment s: x=y ∈U
02:     safe_delete←true
03:     if use[s]=∅ then
04: delete statement s
05:     else
06:         foreach use u of definition s, u∈use[s]
07:             if s∈in[u]then
08:                 substitute y for x at u
09:             else safe_delete←false
10:         endforeach
11:         if safe_delete=ture then
12:             delete statement s
13: endforeach
```

Fig. 6. Optimization algorithm

V. Experimental Results

In this section, we evaluate the performance of Lit framework in comparison with its CPU-based Hadoop framework using three typical algorithms: Matrix Multiplication(MM), Fast Fourier Transform(FFT) and Scan.

A. Experimental Setup

Our experiments were performed on four x86 servers. Table III shows their hardware configuration. All of these servers run 64-bit CentOS 5.5 with kernel 2.6.18-308.13.1.e15, NVIDIA CUDA 4.2, GPU driver 4.2, Hadoop 1.0.3, and Java 1.6. One of the servers is configured as the Hadoop Job-Tracker(name node), others are Task-Tracker(compute nodes).

TABLE III. Hadware Configuration

	Server1
GPU	Tesla S2050
GPU Core	4*448cores 1.15GHz
GPU MEM	12GB
GPU PCI	PCI-E x8
CPU	Intel Xeon E7520 1.87GHz*32
MEM	64GB

B. Benchmark and Evaluation

Matrix Multiplication (MM): Matrix multiplication is used frequently in many signal processing, machine learning and scientific applications. Different as the MM algorithm evaluated in [11], we use partitioned matrix multiplication algorithm. This is because Mars is running on GPU and our framework is running on a cluster. In order to ensure there is enough computation task on a GPU node, we distribute some parts of the big matrix to one Map or Reduce task instead of each Map task computes multiplication for one row from left matrix and one column from right matrix.

Fast Fourier Transform(FFT): FFT is an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse. FFT is widely used in many signal processing, matrix operation and data compression applications. It is the most time-consuming algorithm in many HPC applications.

Scan(prefix sum) is a process to compute the all-prefix-sums of an array. Scan is often used in lexical analysis,sorting,compression,string comparison and polynomial evaluation. It is a time-consuming process in many natural language processing applications.

TABLE IV. EVALUATION DATA SETS

Algorithm	Small	Medium	Large
MM	size: 5MB	size: 10MB	size: 25MB
	1000x1000	1500x1500	3000x3000
FFT	size: 1.58MB	size: 25MB	size: 99MB
	65536 numbers	1048576 numbers	4194304 numbers
Scan	size: 7.3GB	size: 36.5GB	size: 146GB
	0.45G arrays	2.28G arrays	9.12G arrays

We prepared three different categories of data set (Small, Medium and Large) for the evaluation. As shown in Table IV, the Small data set for MM contains randomly generated real numbers. It contains two 1000×1000 matrices and the file size is 5MB. The Small data set for FFT contains a 1D array, which contains 65536 complex numbers. The Small data set for Scan contains about 0.45G arrays and the file size is 7.3GB. All these data sets are stored as files in the hard disk and will be uploaded to HDFS before evaluation.

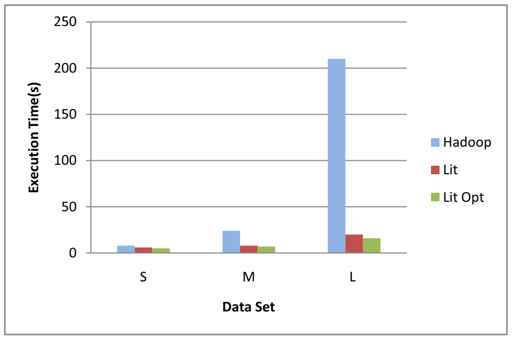


Fig. 7. MM processing time over Hadoop with increasing data size

Figure 7 depicts the comparison of MM computation time between Hadoop and Lit. MM is a typical computation-intensive algorithm. As we can see in the figure, the execution time increased rapidly along with the data size, because the complexity of MM algorithm is $O(n^3)$. When the data set is Small and Medium, the performance of GPUs is not developed adequately, so the execution time between Hadoop and Lit is similar. For the large data set, the execution time of two 3000*3000 matrices will be 600s, if it runs on a single machine. The execution time drop to 210s if it runs on a four-node Hadoop cluster. If we run it on a four-node Lit cluster, the execution time is only 20s. The data flow optimization that improves the performance by approximately 16.4 percent.

Figure 8 depicts the comparison of FFT computation time between Hadoop and Lit. Unlike MM algorithm, FFT is

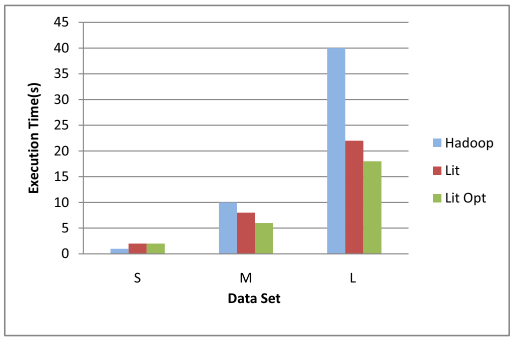


Fig. 8. FFT processing time over Hadoop with increasing data size

an optimized DFT algorithm, its complexity is $O(N \log N)$. Therefore, we can see in the figure, the performance gain is not as distinct as MM algorithm. When testing on the Small data set, the execution time on Hadoop cluster is 1s, whereas the execution time on Lit is 2s. In this kind of situation, the GPU initial time takes high proportion of the total execution time, thus the performance gain will be not notable. Along with the increasing data size, the performance gain will increases gradually. Therefore, when testing the Large data set, the execution time on Lit is 18s less than on Hadoop. The data flow optimization that improves the performance by approximately 14.4 percent.

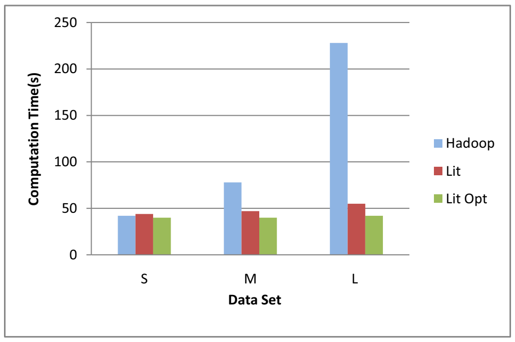


Fig. 9. Scan processing time over Hadoop with increasing data size

Figure 9 presents the comparison of Scan computation time between Hadoop and Lit. Since the time to read data from disk is same between Lit and Hadoop, we only measured the computation time. As we can see in figure 9, when processing the Small data set, the computation time on Lit is even longer than on Hadoop. This is because Lit induced some overhead to the computation process, such as code generation time, CUDA code compilation time and GPU device start up time. When we processing the Medium and Large data set, the computation time on Hadoop increased rapidly whereas the computation time on Lit increased slowly. The data flow optimization that improves the performance by approximately 16.7 percent.

VI. CONCLUSIONS

Motivated by the requirements of the high performance processing of big data, we have proposed the Lit, a Hadoop based high performance big data processing framework that can be used for data-intensive and computation-intensive applications. Lit integrated GPUs with regular Hadoop cluster, which leveraged both the scalability of Hadoop and the high performance of GPUs. Lit is composed of a set of directives, an extended Java compiler, a GPUClassloader and an Optimizer. The programmers annotate the Java based Hadoop source code by the directives to specify the parallelable codes. Lit will then translate the annotated codes into CUDA codes, which will be loaded by the GPUClassloader and executed on the GPU. Our experimental results show that the Lit framework can yield much better performance improvement over Hadoop. With the increasing of the data size, the performance speedup is getting higher, because the overhead introduced by the compilation and GPU initialization is getting negligible compare to the computation time.

In the current implementation, we have optimized the data flow to eliminate the unnecessary data transfers between CPU memory and GPU memory. This can effectively improve the performance of our framework. Meanwhile, we have also observed that if some data transfer instructions can be combined into one instruction, the data transfer will be efficiently reduced. Thus, part of our future work will be to study instruction scheduling based algorithm to optimize the data transfer between CPU and GPU.

Acknowledgment

The authors thank the anonymous reviewers for their insightful suggestions. This work is supported by National 863 Programme(No 2013AA01A212) "Kernel Software and System for Intelligent Cloud Service and Management Platform" and Young Outstanding Teacher Award, the Beijing Institute of Technology.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] P. Vouzis and N. Sahinidis, "Gpu-blast: using graphics processors to accelerate protein sequence alignment," *Bioinformatics*, vol. 27, no. 2, pp. 182–188, 2011.

[3] J. Chi, F. Liu, E. Weber, Y. Li, and S. Crozier, "Gpu-accelerated fdtd modeling of radio-frequency field-tissue interactions in high field mri," *Biomedical Engineering, IEEE Transactions on*, no. 99, pp. 1–1, 2011.

[4] U. Bordoloi, S. Chakraborty, and U. Bordoloi, "Accelerating system-level design tasks using graphics processors," *International Journal of Parallel Programming*, vol. 38, no. 3-4, pp. 225–253, 2011.

[5] D. Mayerich, J. Kwon, A. Panchal, J. Keyser, and Y. Choe, "Fast cell detection in high-throughput imagery using gpu-accelerated machine learning," in *Biomedical Imaging: From Nano to Macro, 2011 IEEE International Symposium on*. IEEE, 2011, pp. 719–723.

[6] L. Jian, C. Wang, Y. Liu, S. Liang, W. Yi, and Y. Shi, "Parallel data mining techniques on graphics processing unit with compute unified device architecture (cuda)," *The Journal of Supercomputing*, pp. 1–26, 2011.

[7] Y. Zhai, E. Mbarushimana, W. Li, J. Zhang, and Y. Guo, "Lit: A high performance massive data computing framework based on cpu/gpu cluster," in *in Proc. of IEEE Cluster*, 2013.

[8] J. Stuart and J. Owens, "Multi-gpu mapreduce on gpu clusters," in *Proceedings of the 2011 IEEE International Conference on Parallel Distributed Processing Symposium (IPDPS)*, 2011, pp. 1068–1079.

[9] F. Ji and X. Ma, "Using shared memory to accelerate mapreduce on graphics processing units," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 2011, pp. 805–816.

[10] 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. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012.

[11] W. Fang, B. He, Q. Luo, and N. Govindaraju, "Mars: Accelerating mapreduce with graphics processors," *Parallel and Distributed Systems, IEEE Transactions on*, no. 99, pp. 1–1, 2010.

[12] 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*. Washington, DC, USA: IEEE Computer Society, 2007, p. 1324, ACM ID: 1318097.

[13] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, "Mapcg: writing parallel program portable between CPU and GPU," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 217–226, ACM ID: 1854303.

[14] C. Basaran and K.-D. Kang, "Grex: An efficient mapreduce framework for graphics processing units," *Journal of Parallel and Distributed Computing*, vol. 73, no. 4, pp. 522–533, 2013.

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

[16] C. Nvidia, *NVIDIA CUDA programming guide*, 2011. [Online]. Available: http://computer4.net/NVIDIA-CUDA-Programming-Guide-download-w2894.pdf

[17] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[18] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic CPU-GPU communication management and optimization," *SIGPLAN Not. PLDI '11*, vol. 47, no. 6, p. 142151, 2011.

[19] L. Chen, B. Shou, X. Hou, and L. Huang, "A compiler-assisted runtime-prefetching scheme for heterogeneous platforms," in *Proceedings of the 8th international conference on OpenMP in a Heterogeneous World*, ser. IWOMP'12. Berlin, Heidelberg: Springer-Verlag, 2012, p. 116129.

[20] M. Amini, F. Coelho, F. Irigoin, and R. Keryell, "Static compilation analysis for host-accelerator communication optimization," in *In Proc. of the 24th International Workshop on Languages and Compilers for Parallel Computing*, 2011.

[21] S. Lee, S. Min, and R. Eigenmann, "Openmp to gpgpu: a compiler framework for automatic translation and optimization," *ACM Sigplan Notices*, vol. 44, no. 4, pp. 101–110, 2009.

[22] Y. Zhai, H. Su, and S. Zhan, "A Data Flow Optimization Based Approach for BPEL Processes Partition," in *Proc. of IEEE International Conference on e-Business Engineering (ICEBE)*, 2007.

[23] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: principles, techniques, and tools*. Pearson/Addison Wesley, 2007, vol. 1009.