基于Hadoop的高性能海量数据处理平台研究

翟岩龙[1] 罗 壮[1] 杨 凯[2] 徐晟晨[1]
(北京理工大学计算机学院 北京 100081)[1] (北京仿真中心 北京 100854)[2]

摘 要 海量数据高性能计算蕴藏着巨大的应用价值,但是目前云计算体系只具有海量数据处理能力,而不具有足够的高性能计算能力。将具有超强并行计算能力的GPU与云计算相融合,提出了基于CPU/GPU协同的异构高性能云计算体系结构。以开源Hadoop为基础,采用注释码的形式对MapReduce函数中需要并行的部分进行标记。通过定制GPU类加载器,将被标记代码转换为CUDA代码并动态编译运行。该平台将GPU的计算能力融合到MapReduce框架中,可高效处理海量数据。

关键词 CPU/GPU协同计算,Hadoop,海量数据处理,高性能计算
中图法分类号 TP309 **文献标识码** A

High Performance Massive Data Computing Framework Based on Hadoop Cluster

ZHAI Yan-long[1] LUO Zhuang[1] YANG Kai[2] XU Sheng-chen[1]
(School of Computer Science, Beijing Institute of Technology, Beijing 100081, China)[1]
(Beijing Simulation Center, Beijing 100854, China)[2]

Abstract HPC of massive data presents tremendous value. However, cloud systems still lack HPC computing power. This study improved the HPC ability of cloud computing technology by adding GPU to the cloud system. The proposed platform is based on Hadoop MapReduce programming model, and it defines some OpenMP like directives to annotate MapReduce program. The annotated code will try to be executed in parallel. A GPUClassloader was designed to convert annotated java code regions to CUDA code. With JNI, generated CUDA code and run on the GPUs. The computing results of GPUs can be transferred back to the map function, in the end, the map function finishes the rest computing. The platform can support the user to complete CPU, GPU collaborative large-scale data parallel processing programming conveniently.

Keywords CPU/GPU collaborative computing, Hadoop, Massive data processing, HPC

1 引言

云计算技术,尤其是谷歌提出的MapReduce计算框架被广泛应用于海量数据处理的各种应用。但是有一类非常重要的海量数据处理应用却很难用目前的云计算技术解决,这类应用同时具有数据密集(Data-intensive)和计算密集(Computational-intensive)两个特点,我们称之为海量数据高性能计算。云计算可以将海量数据分布在大规模集群上进行并行处理,但是集群中每个节点的运算能力却不能满足应用需求,云计算技术的高性能处理能力有待加强。GPU(Graphics Processing Unit)的并行计算能力已经远远超越CPU,并且越来越多地用于通用计算。因此本项目拟研究CPU/GPU协同的计算体系,将具有超强并行计算能力的GPU纳入到现有云计算MapReduce计算框架中,以增强其高性能处理能力。

2 相关工作

MapReduce是由Google提出的并行编程模型[1]。开发人员只需要提供自己的Map函数以及Reduce函数即可并行处理海量数据。由于MapReduce编程模型的简便性,它已经被广泛应用于数据挖掘、机器学习、文档聚类、统计机器翻译等领域。除了Google的MapReduce实现外,应用最广泛的是Hadoop开源项目的MapReduce实现。这两个实现都是基于大规模服务器集群,主要用于处理海量数据信息。虽然已经存在一些研究工作尝试实现高性能的MapReduce计算框架[2],但是他们或是没有考虑海量数据的处理问题,或是没有基于目前最通用Hadoop平台。为了降低在大规模共享主存环境中的编程复杂度,R. M. Yoo等研究人员提出了基于多核CPU实现的MapReduce框架Phoenix[3]。Phoenix实现了在多核多处理器上共享主存模式的MapReduce运行环境,通过自动化并行管理和任务调度,简化并行程序设计,并且达到了平均2.5倍的加速比。Mars[4]是由Wenbin Fang等研究人员基于Nvidia CUDA编程方法实现的、在GPU上运行的MapReduce框架。Mars通过在GPU上实现计数阶段和双调排序算法解决GPU不能动态分配内存等问题。同样,为了将GPU的强大计算能力与MapReduce的简单并行编程模型结合,Bryan Catanzaro等研究人员也在GPU上实现了MapRe-

到稿日期:2012-10-19 返修日期:2012-12-19
翟岩龙 博士,讲师,硕士生导师,主要研究方向为云计算、分布式系统、并行计算、移动机器人,E-mail:ylzhai@bit.edu.cn(通信作者)。

duce 框架[5]。MapCG[6]是清华大学信息科学与技术国家实验室开发的同时支持 GPU 与 CPU 的 MapReduce 框架。MapCG 定义了统一的编程语言，并将编程语言翻译成可以在 CPU 和 GPU 上执行的不同版本，然后用运行时库进行编译执行。

3 系统设计与实现

本系统基于开源 Hadoop 实现，遵循 MapReduce 编程模型，由一个 master 节点和多个 slave 节点组成。其中 master 节点肩负 HDFS NameNode 和 MapReduce JobTracker 的职责，slave 节点为 DataNode 和 TaskTracker。该系统通过借鉴 OpenMP 并行编程模式，设计了一套注释码。程序员只需要在 JAVA 编写的 MapReduce 源程序中标记出可并行执行的部分，系统会将该部分自动转换成 CUDA 代码并在 GPU 上运行。由于系统事先并不知道主机是否配置了 GPU 和 CUDA，这就要求要动态地生成 CUDA 代码。当主机不具备 GPU 时，系统应该执行原本的 MapReduce 程序。

因此我们定义了一个新的 JAVA 类加载器 GPUClassLoader 来控制代码转换流程。GPUClassLoader 将识别出 JAVA 字节码中被注释码标记的部分，生成相应的 CUDA 代码，编译连接 CUDA 代码生成动态链接库，并用 JNI 的方式调用相应的 CUDA 程序。

3.1 系统工作流程

该系统是以开源 Hadoop 中的 MapReduce 为基础架构，在 MapReduce 中设计一套注释码，这些注释码用于标记 Map 函数中需要并行的代码部分，并通过在传统的 JAVA class loader 进行改进，实现 GPU JAVA class loader。结合 CUDA 对程序进行处理，具体工作流程分为 4 个阶段。第一阶段为代码编写阶段，编写代码时，程序员对需要在 GPU 上并行的代码部分进行标记，标记方法是使用已定义的注释码，标记的代码内容只能是循环或特殊的数学函数，否则该句注释码将被视为无效。然后进入准备阶段，在准备阶段，系统将按照 MapReduce 中的方式将待处理数据划分为若干个 map 任务，将这些任务分配到各个计算节点，并启动相应数量的 reduce 任务。在接下来的编译阶段，系统对程序进行编译，编译 map 函数时，首先进行预处理，再使用 JAVA complier 完成编译，可以获得含有注释码的 JAVA 字节码。在最后的运行阶段，GPU JAVA class loader 将自动检测本地计算环境，检查 CUDA 是否可用，若不可用，则直接在 CPU 上进行计算；若可用，则检测 CUDA 的具体版本，并识别 JAVA Bytecode 中被注释的代码部分（即需要在 GPU 上运行的部分），GPU JAVA class loader 对于识别出的 JAVA Bytecode 中被注释的部分生成相应 CUDA 代码，包括一段功能函数代码和一段执行代码，并编译这两段代码。用 JNI 的方式来调用编译后的 CUDA 代码，相关数据被拷贝到 GPU 存储器上，CUDA 代码在 GPU 上运行。GPU 计算结束后，CUDA 代码的运算结果被拷贝回本地主存，map 函数获取这些运算结果。map 函数中未被标记的代码部分在 CPU 上运行。在运行过程中，Hadoop 调度节点跟踪所有 map 任务的运行状态，对于运行失败的 map 任务重新运行，直到所有 map 任务完成，Map 过程结束。接下来进行 Reduce 阶段，汇总 Map 阶段的运算结果。图 1 为系统流程示意图。

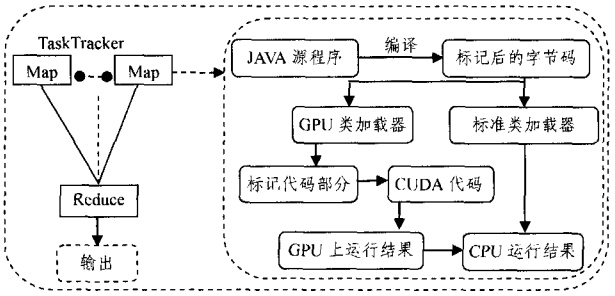

图 1 系统流程示意图

3.2 注释码

本系统采用注释码的形式来标记并行执行的代码，这主要参考了 OpenMP 的设计思想。在 OpenMP 中，程序员通过在源程序中添加特定的注释码来标记一些并行操作，通过 OpenMP 编译器进行编译，生成可以在多线程环境中并行执行的程序。OpenMP 将注释码分为 4 类，第一类对代码部分进行并行划分（omp parallel）；第二类用于标注在不同的线程间共享的操作（omp for，omp sections）；第三类用于标记同步操作（omp barrier，omp flush，omp critical 等）；第四类用于标记数据属性（omp shared，omp threadprivate 等）。系统中的注释码采用 JAVA 注解的方式进行实现。系统会用 CUDA 实现一定数量的函数，这类函数的特征是数据传输量小，但计算量大，从而适宜在 GPU 下进行运算。目前本系统只定义了一些常用的注释码，如表 1 所列。其中，gmp parallel 和 gmp parallel for 用于标记源码中可被并行执行的代码部分。而 gmp sync 则用于实现全局同步，由于 CUDA 并不支持全局同步机制，因此系统会将被标记部分分割为两个子部分，以实现同步。*gmp shared* 和 *gmp private* 用于实现数据在 CPU memory 和 GPU memory 之间的数据传输。

表 1 定义的注释码

注释码	语义说明	转换规则
gmp parallel	标记源代码中的并行部分	一个或多个 CUDA 函数调用
gmp parallel for	标记 for 循环，每一轮循环由一个 GPU 线程执行	生成一个 CUDA 核心函数
gmp sync	标记全局同步部分	将并行部分分割为两个子部分
gmp shared	标记全局共享部分	数据将被存放到全局内存
gmp private	标记线程私有部分	数据将被存放寄存器或局部内存

3.3 GPUClassLoader 实现

GPUClassLoader 继承自 AppClassLoader 类。通过覆盖 defineClass 和 loadClass 方法，GPUClassLoader 可以从 JAVA 字节码中找到被指定注释码标记的部分代码；然后启动代码生成流程，生成相应的 CUDA 代码后，并将其保存到磁盘文件中；接着调用 CUDA 编译器，编译、连接 CUDA 代码生成动态链接库，最后通过 JNI 调用的方式调用这部分 CUDA 代码，从而实现其在 GPU 上的运算。

3.4 转换实例

本节通过一个具体的程序示例演示本系统的代码转换过

程。下面程序所示为一段简单的 JAVA 循环代码，编程人员通过 gmp parallel for 将 for 循环标记为可以在 GPU 上并行执行。在这个 for 循环中，每一轮循环都相互独立。

```
//原始代码
class Demo{
  void work (int [] arr){
    //#gmp parallel for
    for (int i=S;i<E;i+=K){
      arr[i]++;
    }
  }
}
```

在进行转换时，首先编译器将被注释码标记部分进行封装。代码如下所示，循环被封装在一个 ForLoopJob 中，ForLoopJob 是 CudaJob 的子类，主要用于处理 for 循环。在 ForLoopJob 中循环代码被添加到 run 函数中，通过 ForLoopJob 的 initData 函数实现数据向 GPU 显存的拷贝。通过 CUDA TaskManager 启动这个 CudaJob 实例。在编译时，CUDA TaskManager 的启动 CudaJob 的代码将被替换为 JNI 调用，表明具体的并行化代码将放到 GPU 上执行。

```
//转换代码
class ForLoopJob extends CudaJob{
  int arr[],S,E,K;
  void run (){
    Job w;
    ForLoopJob. initData(S,E,K);
    for(int i=w. start;i<w. end;i+=w. step)
    {
      arr[i]++;
    }
  }
}
class Demo{
  void work(int [] arr){
    CUDATaskM. start(new ForLoopJob(S,E,K));
  }
}
```

在系统运行时，GPUClassloader 会加载 MapReduce 程序的字节码文件，当检测到系统具有 GPU 运行环境并且代码中包含一些 CudaJob 实例时，就会启动 CUDA 代码生成流程。GPUClassloader 首先调用 JAVA 编译器生成 JNI 调用需要的头文件，然后将 CudaJob 转换为相应的 CUDA 代码，接着调用 CUDA 编译器编译生成动态库文件，交由 GPU 类加载器载入 JVM。下面所示为生成的 CUDA 代码片段。

```
const int N=1 * (E-S)/k;
const int threadsPerBlock=1;
const int blocksPerGrid=imin(32,N);
-global-void kernel(int * out,const int * in)
{
  const uint tidx=-umul24(blockDim. x,blockIdx. x)+threadIdx. x;
  extern-shared-int smem[];
  smem[threadIdx. x+blockDim. x]=in[tidx];
  smem[threadIdx. x+blockDim. x]++;
  out[tidx]=smemsmem[threadIdx. x+blockDim. x];
}
int main(void)
{
  …
  in=(int * )malloc(N * sizeof(int));
  cudaMalloc((void * * )&dev_in,N * sizeof(int));
//init in data
… cudaMemcpy(dev_in,in,N * sizeof(int),cudaMemcpyHostToDevice);
kernel<<<blocksPerGrid,threadsPerBlock>>>(dev_in);
```

4 系统优化

当主机配置了 GPU 和 CUDA 运行环境时，GPUClassloader 会启动代码转换和编译流程生成动态链接库，并将其加载到 JVM 中。但是这一编译加载过程引入了较大的时间延迟。因此本文在系统流程方面进行了相应的优化。在 Hadoop 启动新的 JVM 执行 Map 或 Reduce 任务的同时，启动一个新的线程进行代码转换和编译工作。此线程一次性地将所有并行区域都转换成 CUDA 代码，将 CUDA 代码保存在磁盘文件中，然后进行编译、连接，生成动态链接库。

本文还对提出的计算框架的内存拷贝进行了优化。众所周知，在 CPU 主存和 GPU 内存之间拷贝数据效率非常低，因此需要尽量减少数据交换的次数和数据交换的数量。目前的代码转换规则将并行区域内可能用到的数据全部拷贝到 GPU 的共享内存中，但实际上并不是全部的数据都需要拷贝到 GPU 内存中，也不是所有在 GPU 共享内存中的数据都需要拷贝回 CPU 内存。基于作者前期的研究成果[7]，本文采用数据流分析的方法优化 CPU 与 GPU 的内存交换。如果变量在 CUDA 程序内没有引用，则不进行拷贝；如果 CUDA 程序中的变量在后续 CPU 代码中没有引用，则不拷贝回 CPU 内存。此方法有效地减少了 CPU 与 GPU 内存数据的传输。

5 系统性能实验

为了验证本文提出的海量数据高性能计算系统的性能，搭建了由 4 台服务器组成的 Hadoop 集群实验环境。集群的每个节点都是 IBM System X3850 服务器，其中 3 台配备了 NVIDIA 的 S2050 GPU 服务器、64 位 RHEL 5.5 操作系统、Sun JDK 1.6。性能验证程序采用了经济度量模型中常用的矩阵乘法和快速傅里叶变换两种算法，并生成了 3 种不同规模的测试数据集：小数据集(S，10MB)、中数据集(M，50MB)和大数据集(L，100MB)。

针对矩阵乘法，采用矩阵分块乘法，首先将矩阵分块，将不同的分块相乘交由不同的 Map 任务进行计算，然后通过 Reduce 任务进行最终结果的汇总。针对快速傅里叶变换，采用递归式的快速傅里叶变换算法，将数列分为奇偶两个序列，每个单独进行傅里叶变换。由于这两种算法都具有数据并行的特点，因此可以通过 MapReduce 程序的方式实现。

图 2 为针对不同数据集，采用 Hadoop 和集成了 GPU 的

Hadoop 计算矩阵乘法和快速傅里叶变换的时间对比。当采用小数据集时，采用 Hadoop 实现的矩阵乘法的运行时间为 40s，采用本文提出的集成了 GPU 的 Hadoop 计算时间为 23s，加速效果并不是特别明显，这主要由于 GPU 启动比较耗时造成的。当采用大数据集时，GPU 加速的效果变得很明显。如当采用中数据集时，采用 Hadoop 计算时间为 680s，采用本系统的计算时间为 120s。快速傅里叶变换的运行时间和加速效果与矩阵乘法类似。可见对于那些数据密集型同时是计算密集型的应用，集成了 GPU 的 Hadoop 平台，其加速效果好。

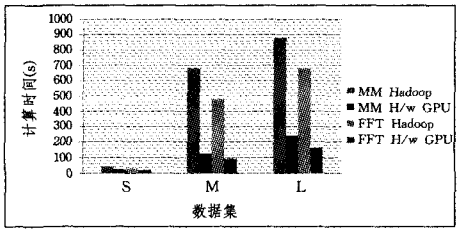

图 2　不同数据集时两种算法的计算时间

图 3 为采用集成了 GPU 的 Hadoop 系统计算矩阵乘法时进行系统优化的效果图。当不采用系统优化时，CUDA 代码的自动生成和编译占用时间约为整个计算过程的 10%（采用 S 数据集时为 20%）。当采用了系统优化后，CUDA 代码的生成和编译时间占整个计算过程的百分比几乎可以忽略。可见，系统通过采用专用线程进行代码转换和编译连接工作可以有效提高系统的运行效率，以降低非计算任务占用的系统时间。

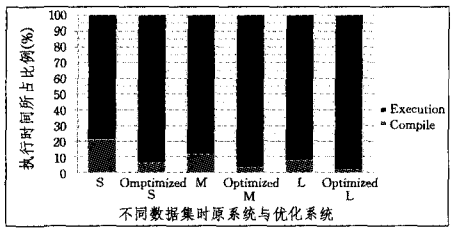

图 3　矩阵乘法算法系统优化效果

结束语　本文的核心内容是实现一种面向海量数据高性能计算的 CPU、GPU 协同计算方法，其实现形式是设计一个可以同时利用 CPU 和 GPU 计算能力的、基于计算机集群的平台。该方法可以以便捷的方式整合计算机集群中的 CPU、GPU 计算资源，从而提高计算机集群的海量数据处理性能。本文提出的计算框架已经作为核心计算方式应用于具体科研项目中，在北京市科技计划课题“能源行业海量数据成像云计算系统产业化”中用于地震数据的叠前偏移计算。实践结果表明，原本采用 CPU 计算需要数小时、采用 GPU 加速后需要 20 分钟的数据量，在此框架中（5 节点）仅需要约 5 分钟即可计算完毕，可见此框架在不增加程序设计难度的前提下，将 Hadoop 的海量数据处理能力和 GPU 的高性能计算能力良好地结合在一起，具有较好的应用价值。

参 考 文 献

[1] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[C]// Communications of the ACM. vol. 51, 2008: 107-113

[2] Vecchiola C, Pandey S, Buyya R. High-performance cloud computing: A view of scientific applications[C]// 2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks. 2009: 4-16

[3] Yoo R M, Romano A, Kozyrakis C. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system[C]// IEEE International Symposium on Workload Characterization, 2009 (IISWC 2009). 2009: 198-207

[4] Fang W, He B, Luo Q, et al. Mars: Accelerating MapReduce with Graphics Processors[J]. IEEE Transactions on Parallel and Distributed Systems, 2011, 22: 608-620

[5] Catanzaro B, Sundaram N, Keutzer K. A map reduce framework for programming graphics processors[C]// Workshop on Software Tools for MultiCore Systems. 2008

[6] Hong C, Chen D, Chen W, et al. MapCG: writing parallel program portable between CPU and GPU[C]// Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques. New York, NY, USA: ACM, 2010: 217-226

[7] Zhai Yan-long, Su Hong-yi, Zhan Shou-yi. A Data Flow Optimization based approach for BPEL Processes Partition[C]// IEEE International Conference on e-Business Engineering (ICEBE 2007). HongKong, China, 2007: 410-413

(上接第 73 页)

[13] Bloom B. Space/time trade-offs in hash coding with allowable errors[J]. Communications of the ACM, 1970, 13(7): 422-426

[14] Dharmapurikar S, Krishnamurthy P, Sproull T, et al. Deep packet inspection using parallel Bloom filters[J]. IEEE Micro, 2004, 24(1): 52-61

[15] Dharmapurikar S, Lockwood J. Fast and scalable pattern matching for network intrusion detection systems[J]. IEEE Journal on Selected Areas in Communications, 2006, 24(10): 1781-1792

[16] Snort. Home Page[EB/OL]. http://www.snort.org/, 2012-07-10

[17] Morris R. Scatter storage techniques[J]. Communication of the ACM, 1968, 11(1): 38-44

[18] Broder A, Mitzenmacher M. Network applications of bloom filters: A survey[J]. Internet Mathematics, 2003, 1(4): 485-509

[19] MIT DARPA Intrusion Detecion Data Sets[DB/OL]. http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/1999/training/week1/monday/outside.tcpdump.gz, 2011-12-02

[20] 刘威，郭渊博，黄鹏. 基于 Bloom filter 的多模式匹配引擎[J]. 电子学报，2010，38(5)：1095-1099

[21] 李伟男，鄂跃鹏，葛敬国，等. 多模式匹配算法及硬件实现[J]. 软件学报，2006，17(12)：2403-2415