面向GPU计算平台的归约算法的性能优化研究

张逸然[1] 陈 龙[2] 安向哲[2] 颜深根[3]
(北京信息科技大学 北京 100049)[1]
(中国石油集团东方地球物理勘探有限责任公司 河北 涿州 072751)[2]
(深圳市商汤科技有限公司 广东 深圳 518000)[3]

摘 要 归约算法在科学计算和图像处理等领域有着十分广泛的应用,是并行计算的基本算法之一,因此对归约算法进行加速具有重要意义。为了充分挖掘异构计算平台下GPU的计算能力以对归约算法进行加速,文中提出基于线程内归约、work-group内归约和work-group间归约3个层面的归约优化方法,并打破以往相关工作将优化重心集中在work-group内归约上的传统思维,通过论证指出线程内归约才是归约算法的瓶颈所在。实验结果表明,在不同的数据规模下,所提归约算法与经过精心优化的OpenCV库的CPU版本相比,在AMD W8000和NVIDIA Tesla K20M平台上分别达到了3.91～15.93和2.97～20.24的加速比;相比于OpenCV库的CUDA版本与OpenCL版本,在NVIDIA Tesla K20M平台上分别达到了2.25～5.97和1.25～1.75的加速比;相比于OpenCL版本,在AMD W8000平台上达到了1.24～5.15的加速比。文中工作不仅实现了归约算法在GPU计算平台上的高性能,而且实现了在不同GPU计算平台间的性能可移植。

关键词 归约算法,GPU,线程内归约,OpenCL
中图法分类号 TP391 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2019.02.047

Study on Performance Optimization of Reduction Algorithm Targeting GPU Computing Platform

ZHANG Yi-ran[1] CHEN Long[2] AN Xiang-zhe[2] YAN Shen-gen[3]
(Beijing Information Science & Technology University, Beijing 100049, China)[1]
(BGP INC., China National Petroleum Corporation, Zhuozhou, Hebei 072751, China)[2]
(SenseTime, Shenzhen, Guangdong 518000, China)[3]

Abstract Reduction algorithm has wide application in scientific computing and image processing, and it is one of the basic algorithms of parallel computing. Hence, it is significant to accelerate reduction algorithm. In order to fully exploit the capability of GPU for general-purpose computing under heterogeneous processing platform, this paper proposed a multi-level reduction optimization algorithm including inner-thread reduction, inner-work-group reduction and inter-work-group reduction. Different from the traditional way of reduction algorithm optimization of putting more emphasis on inner-work-group reduction, this paper proved that inner-thread reduction is the true bottleneck of reduction algorithm. The experimental results demonstrate that the performance of proposed reduction algorithm has reached 3.91～15.93 and 2.97～20.24 times speedup respectively in AMD W8000 and NVIDIA Tesla K20M under different sizes of data set, compared with carefully optimized CPU version of OpenCV library. In NVIDIA Tesla K20M, compared with CUDA version and OpenCL version of OpenCV library, the algorithm has reached 2.25～5.97 and 1.25～1.75 times speedup respectively. And compared with OpenCL version of OpenCV library in AMD W8000, the algorithm has reached 1.24～5.15 times speedup. This work not only realizes high performance of reduction algorithm on GPU platform, but also reaches the portability of performance between different GPU computing platforms.

Keywords Reduction algorithm, GPU, Inner threads reduction, OpenCL

图形处理器(Graphics Processing Unit,GPU)是面向吞吐量、采用统一架构单元设计的大规模细粒度并行处理器。与CPU相比,GPU具有更强的计算能力和更大的存储带宽,更加适合当前大规模数据并行处理的需求[1]。近年来,随着

到稿日期:2018-09-12 返修日期:2018-11-20

张逸然(1999－),男,硕士生,主要研究方向为高性能计算、众核编程与优化方法;**陈 龙**(1973－),男,硕士,主要研究方向为高性能计算、人工智能在勘探行业的应用;**安向哲**(1974－),硕士,主要研究方向为云计算;**颜深根**(1985－),男,博士,主要研究方向为GPU编程与优化、自适应性能调优、大规模分布式深度学习等。

GPU 计算能力和可编程性的不断增强，GPU 通用计算(General Purpose computing on Graphics Processing Units，GPGPU)[2-3]不仅在卫星数据、大气预报、基因工程、分子动力学模拟等传统科学领域得到越来越广泛的应用，而且在多媒体处理、图形图像处理、游戏等非传统领域也越来越受到重视，如图形图像领域应用最为广泛的 OpenCV 库(Open Source Computer Vision)已经推出其 GPU 版本，并取得了非常大的性能提升。总之，利用 GPU 对应用程序进行加速已经逐步成为提升程序性能的主要模式，越来越多的算法移植到 GPU 上，并取得了可观的加速比。因此，研究算法在 GPU 上实现和优化的关键方法和技术日益成为研究热点。

传统的串行归约算法以一维数组归约求和为例对数组数据进行压缩求和，直至只剩一个值，该值即为数组元素值总和。这样，对 N 个数求和需要进行 $N-1$ 次加法运算操作。除了归约求和，对归约算法进行简单修改后可扩展应用于求最大值、最小值等场景。串行归约求和算法 Serial Reduction 的伪代码如算法 1 所示。

算法 1　Serial Reduction

```
Input:src(Original data),DATA_NUM(Length of src)
Output:sum(Sum of src array)
1. sum←0
2. for i=0 to DATA_NUM do
3.    sum ←sum+src[i]
4. end for
```

由算法 1 可知，串行归约求和算法的实现逻辑十分简单，通过 for 循环对每一个元素进行迭代累加即可求出数组元素之和。然而，一方面，归约算法中每一次迭代需要利用上一次迭代的中间结果，即相邻的循环迭代间的累加操作存在数据依赖，限制了算法的并行性；另一方面，当采用传统的 tree-based 并行归约算法实现该算法在 GPU 上的移植和优化时，随着归约的进行，越来越多的线程将处于空闲状态，这对于采用大规模细粒度并行架构的 GPU 来说，无疑会成为主要的性能瓶颈。因此，简单地将传统并行归约算法直接移植到 GPU 上，并不能保证取得良好的性能提升。

本文采用层次式归约的思想，将归约算法在 GPU 上的实现和优化分为线程内归约、work-group 内归约和 work-group 间归约 3 个层次，并打破以往相关工作将优化重心集中在 work-group 内归约上的传统思维。本文研究发现，线程内归约才是归约算法在 GPU 上取得性能提升的关键因素。本文在研究传统 work-group 内归约优化的基础上，提出了基于 stride 的线程内归约和基于原子操作的 work-group 间归约等优化方法和策略。实验结果表明，在不同的数据规模下，本文提出的 PCL-Reduction 在 AMD W8000 和 NVIDIA Tesla K20M 平台上，与经过精心优化的 OpenCV 库的 CPU 版本相比，分别达到了 3.91～15.93 和 2.97～20.24 的加速比；在 NVIDIA Tesla K20M 平台上，相比于 OpenCV 库的 CUDA 版本与 OpenCL 版本分别达到了 2.25～5.97 和 1.25～1.75 的加速比；在 AMD W8000 平台上，相比于 OpenCL 版本，达到了 1.24～5.15 的加速比。

本文的主要贡献如下：

1)提出了归约算法并行化的 3 个层次，并将其抽象定义为线程内归约、work-group 内归约以及 work-group 间归约。

2)打破了以往工作将优化重心集中在 work-group 内归约的传统思维，论证了线程内归约才是归约算法在 GPU 上取得良好性能提升的关键因素，是归约算法的瓶颈所在；并且提出基于 stride 的线程内归约算法，大大提升了归约算法在 GPU 计算平台上的了性能。

3)完成了归约算法在不同 GPU 计算平台上的高效实现，不仅实现了高性能，而且实现了性能可移植。

本文第 1 节介绍并行归约算法设计与实现的相关工作；第 2 节简要介绍异构计算和 OpenCL 框架[4]的基本概念；第 3 节介绍并行归约算法的基本实现及本文采用的多种优化策略；第 4 节对采用不同优化策略实现的并行归约算法进行性能评估并分析性能提升的原因；最后对全文进行总结。

1　相关工作

归约算法在科学计算和图像处理等领域有着十分广泛的应用和发展，是并行计算的基本算法之一，因此将归约算法在 GPU 上进行实现和优化具有重要意义。

Tree-based 并行归约算法是最常用的并行归约算法，如图 1 所示。归约算法在 GPU 上的并行化实现与优化均以此为基础。

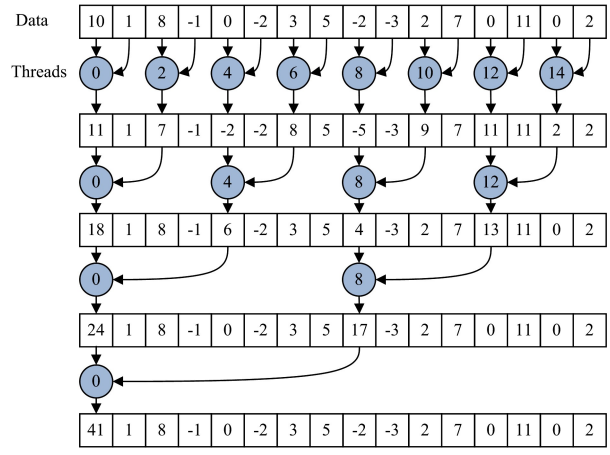


图 1　GPU 归约算法的基础实现

Fig. 1　Underlying implementation of GPU reduction algorithm

Harris[1]在 NVIDIA GPU 上完成了并行归约算法的设计与实现，并给出了较为深入的性能剖析。其优化方法可简单概括为：work-group 内归约优化(warp 内线程条件分支优化、LDSbank conflict 优化、循环展开)和简单线程内归约优化；结合归约算法特性及 GPU 硬件特性进行优化，尽量减少不规则访存和本地同步操作的使用。其中，Bryan[2]论述了归约算法符合加法交换律和结合律的特性，进而为 work-group 内归约提供了理论支撑。

Yan 等[3]通过设计和实现归约算法的 OpenCL 实现版本，在 NVIDIA 和 AMD 的 GPU 上得到了良好的性能加速。其侧重于研究如何利用 GPU 不同存储层次的访存特性、向

量化和固定 work-group 数目等优化方法来提升归约算法的性能。

目前,OpenCV 库[4-5]已经实现了归约算法在 GPU 上的移植和优化,可以运行在不同厂商的 GPU 上。

以上工作更加侧重于归约算法的 work-group 内归约优化,不曾讨论跨步为全局线程数大小(global size)以及跨步为单位步长(向量化)这两种情况,以及线程内归约过程中每个线程负责归约的数据量对归约算法的影响。而本文将重点讨论线程内归约过程随着每个线程负责归约的数据量的增加而逐渐成为归约算法的瓶颈;此外,还将关注 work-group 间归约优化(累加局部归约结果)时使用原子操作所带来的影响。本文对上述问题进行了补充论证,以剖析不同方法给算法性能带来的影响。

本文工作参考了其他算法在 GPU 上的优化方法和策略,这些方法和策略虽然是针对不同算法,但有通用之处,能为本文工作提供很大的帮助。Prashant[6]针对 stencil 算法介绍了 GPU 寄存器的使用和优化策略,GPU 寄存器的使用会影响同时运行的线程的数目,进而影响程序性能。Cris[7]通过 FFT 算法在 GPU 上的实现和优化,介绍了如何减少 GPU 线程间的通信,从而降低通信开销,提升程序性能。Zhang 等[8]详细介绍和讨论了 GPU 的微体系架构特征以及这些特征对优化和性能的影响。Li 等[9]讨论了 GPU Cache 的架构特征以及对 GPU Cache 进行有效管理的方法和策略。David 等[10]在详细介绍 GPU 架构的基础上,给出了 GPU 程序实现和优化的通用方案。GPU 优化的本质就是实现算法特征到 GPU 硬件架构特征的高效映射,因此了解 GPU 的硬件架构特征是 GPU 优化的基础。本文对规约算法进行优化时,将参考上述工作的优化方法和策略。

2 异构计算和 OpenCL 框架

异构计算(Heterogeneous Computing)[11]被视为处理器继单核、多核之后的第三个时代,它通过使用不同类型的指令集和体系架构的计算单元进行协作,进而突破 CPU 发展的瓶颈,是解决可扩展性、能耗等问题的新途径。GPGPU[12-13]采用 CPU+GPU 的异构编程模式,日益成为异构计算的主流。在这种模式下,CPU 负责执行复杂逻辑处理和事务处理等不适合数据并行的计算,GPU 负责计算密集型的大规模数据的并行计算,以充分利用 GPU 强大的计算能力和高带宽弥补 CPU 性能的不足,进而充分挖掘计算机的潜在性能。

随着 CUDA[14-16]和 OpenCL[17-19]等编程框架的进一步发展和成熟,编程人员直接在 GPU 上进行通用计算编程得以实现。同时,OpenCL 作为一种跨平台的异构计算框架,进一步使 GPU 硬件编程突破了硬件厂商的限制,搭建起通用计算的协同计算平台,以支持不同厂商的 GPU,APU(Accelerated Processing Unit),Cell 等。

OpenCL 是一个在异构计算平台上进行并行程序开发的开放工业标准,它将 CPU、GPU 以及其他各类计算设备组织成一个统一的计算平台,开发者以统一而有效的方式使用这些计算资源[4,9,13,20]。OpenCL 有两个重要特点:1)跨厂商,其可以运行在任何支持 OpenCL 的厂商的设备上,OpenCL 已经得到了 Intel,NVIDIA 和 AMD 等主流芯片厂商的支持,各大厂商都有对 OpenCL 标准的实现;2)跨平台,OpenCL 程序不仅可以在各种异构的平台上自由移植,而且能够以统一的方式充分有效地利用异构系统中的各种计算资源。OpenCL 框架可划分为平台模型、内存模型、执行模型和编程模型。OpenCL 的平台模型由一个 Host(主机)连接一个或多个计算设备(Compute Device)构成,而每一个计算设备可进一步划分成一个或多个计算单元(Compute Unit),每一个计算单元可由一个或多个处理单元(Processing Element)组成。其中,OpenCL 设备可以是 CPU,GPU,APU 等。OpenCL 设备的内存模型分为 4 种:全局内存、常量内存、局部内存和私有内存。各存储层次的带宽由高到低依次为:私有内存>常量内存>局部内存>全局内存。其中,不同的存储层次对于不同的 PE,work-group 等均有不同的访问权限限制,而且不同存储层次在满足特定访存请求的情况下会带来性能的提升。OpenCL 的执行模型可分为两个部分:运行于 Host 端的主程序和运行于计算设备的内核程序(Kernel)。主程序通过定义和管理上下文(Context)来控制内核程序的执行,内核程序则在 OpenCL 设备上执行具体的计算任务。在 AMD 的 GPU 中,wavefront 为调度和执行的基本单位,每个 wavefront 包含 64 个线程,此数量关系由硬件决定,它提供了比 work-group 更细粒度的并行。若 CU 调度多个 wavefronts,任何一个 wavefront 因访存而阻塞时都可快速切换到下一个就绪的 wavefront,从而隐藏访存的延迟。OpenCL 编程模型支持数据并行和任务并行两种方式,数据并行模型指多个内存对象执行指令序列中定义的运算,在 OpenCL 中用户结合 work-item 的 global ID 和 local ID 进行内存元素的映射。本文归约算法使用数据并行方式。

注意,本文中 AMD GPU 与 NVIDIA GPU 存在相关对应术语,如 wavefront 与 warp 对应、CU 与 SM 对应等,本文统一以 AMD GPU 术语作为描述标准,读者可推广至 NVIDIA GPU 术语;OpenCL 与 CUDA 的对应术语,如 work-group 与 block 对应等,采用 OpenCL 术语作为描述标准。另外,在本文描述中,如无特殊说明,使用 LDS 表示片上共享内存(local memory,OpenCL;share memory,CUDA)。

3 GPU 归约算法的实现和优化

图 2 为本文提出的 GPU 归约算法的执行过程。GPU 归约求和算法的实现可以定义为 3 个层次:

1)线程内归约。线程从 global memory 中读取一个或多个数据进行归约操作,再把归约结果写入 LDS。

2)work-group 内归约。work-group 对 LDS 的数据进行内部归约操作,求出局部归约结果。

3)work-group 间归约。对每一个 work-group 所得的局部归约结果进行累加操作,得到最终归约结果。

本节以 Naïve Reduction 为起点,逐步探求并行归约算法的优化要素,以最大化提升算法性能。

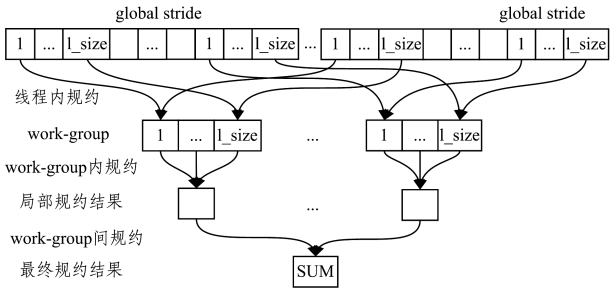


图 2　GPU 归约算法的执行过程

Fig. 2　Execution process of GPU reduction algorithm

3.1　GPU 归约算法的 Naïve 实现

GPU 归约算法的 Naïve 实现采用分治思想：首先将原始数据划分为多个块；然后对每个块进行局部归约操作，求出块内的局部归约结果；最后对局部归约结果进行全局归约操作，得到最终的归约结果。本文的归约算法优化均以 Naïve Reduction 为基础进行，其伪代码如算法 2 所示。

算法 2　Naïve Reduction

```
Input：src(Original data)，lSum(local memory)
Output：dest(Length is 1)
1. idx_loc←get_local_id(0)
2. lSize←get_local_size(0)
3. //线程内归约
4. lSum[idx_loc]←src[idx]
5.    barrier(CLK_LOCAL_MEM_FENCE)
6.    //Work-group 内归约
7.    for i=1 to lSize step i≪1 do
8.       testBit←(i≪1)－1
9.       if (idx_loc&testBit)=0 then
10.         lSum[idx_loc]←lSum[idx_loc+i]
11.      end if
12.      barrier(CLK_LOCAL_MEM_FENCE)；
13.   end for
14. //work-group 间归约
15. if idx_loc=0 then
16.   atom_add(dest，lSum[0])
17. end if
```

3.2　GPU 归约算法的优化

3.2.1　线程内归约优化

线程内归约在 GPU 的移植与优化中常常被忽视。绝大多数归约算法的 GPU 实现和优化都把 work-group 内归约优化作为算法优化的核心，然而，线程内归约才是影响 GPU 归约算法性能的关键因素（4.4 节将对归约算法 GPU 的实现与优化的性能瓶颈进行分析），本节将详细讨论与分析线程内归约的过程。

Naïve Reduction 没有进行线程内归约，一个线程仅对应一个数据，仅负责将数据从 global memory 加载至 LDS 中，然后在 LDS 中进行 work-group 内归约。由于没有进行线程内归约优化，因此从随之进行的 work-group 内归约的第一层归约开始，便有一半线程处于空闲状态，极大地浪费了计算资源。

为了更充分地利用计算资源，应尽可能使所有线程均参与归约操作，将空闲线程出现的时间尽可能往后“推移”。因此，在 work-group 内归约开始之前进行线程内归约操作：每个线程对应多个数据，线程从 global memory 依次读取多个数据并对其进行归约操作，然后把归约结果写入 LDS。线程内归约将每个线程简单的数据加载操作转变为加载归约操作（把原本每次只加载一个数据变成加载多个数据并归约累加，然后把累加结果写入 LDS 中）。这里需要注意的是，我们将每个线程进行线程内归约时所处理的数据量定义为线程内归约粒度。因此，在进行 work-group 内归约之前，所有线程均参与了归约操作，提升了线程计算量和资源利用率，从而挖掘出归约算法更大的并行潜力。

注意，线程内归约粒度的选择需要综合考虑两个条件：1）应尽可能选择大的粒度，以提高计算资源的利用率；2）粒度不能过大，以防止无法开启足够多的线程来隐藏访存延迟。不同的线程内归约粒度对性能的影响可参考 4.3.1 节的性能分析。

本文的线程内归约算法有两种不同的实现方式，其实现的区别主要在于每个线程跨步寻址取数的步长。本文依据步长的不同进行了两组实验，其中步长可设置为全局线程数目大小和 work-group 内线程数目大小，相应的归约算法的版本为 Global-Stride Kernel 和 Local-Stride Kernel。

Global-Stride Kernel 和 Local-Stride Kernel 这两个 kernel 均设定 work-group 内线程数目为 256，同时设定线程内归约粒度为 *times*，因此，kernel 所开启的 work-group 数目将由 work-group 内线程数和线程归约粒度共同决定，其关系如式（1）所示：

$$Num_{\text{work-group}}=\frac{Num_{\text{global-threads}}}{\text{Num}_{\text{local-threads}}}/\text{times} \tag{1}$$

其中，$Num_{\text{work-group}}$ 为 kernel 开启的 work-group 数目，$Num_{\text{global-thread}}$ 为全局线程数目，$Num_{\text{local-thread}}$ 为 work-group 内部线程数目，*times* 为线程内归约粒度。

此时，通过调整线程内归约粒度 *times*，来观察线程内归约过程对归约算法整体性能的影响程度。我们建议 *times* 应至少为 CU 数目的 4 倍以上，因为当 work-group 数量足以使所有 CU 保持忙碌时，才能够有效地隐藏访存延迟。

下面将分别给出 Global-Stride Kernel 和 Local-Stride Kernel 的算法伪代码。

1）Global-Stride Kernel

每一个线程以全局线程总数（global stride）为步长，依次读取相距 global stride 的多个数据（数据量由线程内归约粒度 times 控制），然后对这些数据进行归约处理，最后把归约结果写入位于 LDS 中的 lSum 数组，再进行下一层次的 work-group 内归约优化。其伪代码如算法 3 所示。

算法 3　Global-Stride Kernel

```
Input：src(Original data)，lSum(local memory)
Output：dest(Length is 1)
1. idx←get_global_id(0)
2. idx_loc←get_local_id(0)
```