面向 GPU 计算平台的归约算法的性能优化研究

张逸 x^1 陈 z^2 安向 z^2 颜深 d^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AMDW8000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-ran¹ CHEN Long² AN Xiang-zhe² YAN Shen-gen³ (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 interwork-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 has reached $3.91 \sim 15.93$ and $2.97 \sim 20.24$ 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 $2.25 \sim 5.97$ and $1.25 \sim 1.75$ times speedup respectively. And compared with OpenCL version of OpenCV library in AMD W8000, the algorithm has reached $1.24 \sim 5.15$ 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,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treebased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 workgroup 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 I GFU 归约异伍미蓥屾头坑

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 workgroup 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



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 \ll 1 do
- 8. testBit $(i \ll 1) 1$
- 9. if $(idx_loc \& testBit) = 0$ then
- 10. $lSum[idx_loc] \leftarrow lSum[idx_loc+i]$
- 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:

$$Num_{\rm work-group} = \frac{Num_{\rm global-threads}}{Num_{\rm local-threads}} / times$$
(1)

其中, Numwork-group 为 kernel 开启的 work-group 数目, Numglobal-thread 为全局线程数目, Numlocal-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 workgroup 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), ISum(local memory)

- Output:dest(Length is 1)
- 1. idx←get_global_id(0)
- 2. idx_loc←get_local_id(0)

3. globalSize←get_global_size(0)

4. //线程内归约

5.temp**←**0

6. for i=0 to times

7. temp-src[idx+i * globalSize]+temp

8. end for

9. $lSum[idx_loc] \leftarrow temp$

10. barrier(CLK_LOCAL_MEM_FENCE)

11. //然后进行 work-group 内归约和 work-group 间归约

2)Local-Stride Kernel

每一个线程以 work-group 内线程数(local stride)为步长,读取相距 local 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 4 所示。

算法 4 Local-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)
- 3. globalSize \leftarrow get_global_size(0)
- 4.//线程内归约
- 5. temp**←**0
- 6. idx idx_loc + idx_gro * lSize * times
- 7. for i=0 to times
- 8. if $(idx+i*lSize) \le data_len$
- 9. $temp \leftarrow src[idx + i * lSize] + temp$
- 10. end if
- 11. end for
- 12. lSum[idx_loc]←temp
- 13. barrier(CLK_LOCAL_MEM_FENCE)
- 14. //然后进行 work-group 内归约和 work-group 间归约

两个 kernel 的区别在于每个线程读取相邻数据的步长不同。步长为 global stride 和 local stride 时,两个 kernel 在不同的 GPU 硬件平台上各有优劣,本文将在 4.3.1 节进行实验分析。

3.2.2 work-group 内归约优化

1) Wavefront 优化和局部内存优化

由图 1 可知, Naïve Reduction 执行时 wavefront 内部线 程存在条件分支, 而且对 LDS 的 bank 利用率低。首先, Naïve Reduction 执行归约的线程 ID 并不连续, 意味着同一个 wavefront 的线程在 kernel 执行过程中存在条件分支, 一部分 线程负责归约操作, 另一部分线程则处于空闲状态。其次, 从 图 1 的第一层归约可以看出, 由于存在空转线程, 因此部分 bank 同样处于空闲状态, LDS 的利用率低。

图 3 为改进后的归约算法示意图。wavefront 内线程不存在条件分支,一个 wavefront 所能处理的数据将会翻倍,提升了 wavefront 的工作效率,有效地减少了实际工作的 wavefronts 数目,使其约为 Naïve Reduction 的一半。对于局部内存的访问,通过连续的线程访问连续的数据,连续的 32 个线程将会访问连续的 bank,在提升 LDS 利用率的同时,有效地

避免了 bank conflict,进一步提升了算法性能。将完成 wavefront 优化和局部内存优化的算法版本定义为 Divergence-Free Kernel,其相对于 Naïve Kernel 取得了良好的性能提升, 详情参照 4.3.2节。



图 3 Divergence-Free Kernel 的归约过程

Fig. 3 Reduction process of Divergence-Free Kernel

2)循环展开

本节针对 work-group 内归约进行循环展开优化。首先 从硬件资源组织上分析,每一个 wavefront 由 64 个线程组成 (warp 由 32 个线程组成),wavefront 是 GPU 调度与执行的 基本单位,wavefront 内所有线程均执行相同的指令。由此可 知,在 work-group 内归约中的 for 循环中,当运行线程数小于 或等于 64,即运行线程都属于同一个 wavefront 时,可以省去 显式的本地同步操作,以提升算法性能。

考虑到本文设定 work-group 内部线程数为 256,可对 for 循环进行完全展开。这里需要注意的是,当 work-group 内实 际工作线程的数目大于 64(32,NVIDIA GPU)时,仍需要显 式的本地同步。

因此,在 Divergence-Free Kernel 的基础上提出循环展开 优化后的 work-group 内归约优化算法版本 Completely-Unroll Kernel,其 work-group 内归约的伪代码如算法 5 所示。

算法 5 Completely-Unroll Kernel

```
Input:src(Original data),lSum(local memory)
```

Output:dest(Length is 1)

- 1. //线程内归约
- 2. 采用算法 2 的线程内归约
- 3. //work-group内归约
- 4. volatile __local uint * ldata=lSum;

5. if idx_loc<128 then

- 6. $ldata[idx_loc] \leftarrow ldata[idx_loc+128]$
- 7. end if
- 8. barrier(CLK_LOCAL_MEM_FENCE);

9. if idx_loc<64 then

- 10. $ldata[idx_loc] + = ldata[idx_loc+64]$
- 11. ldata[idx_loc] +=ldata[idx_loc+32]
 12. ...
- 13. ldata[idx loc] + = ldata[idx loc+1]
- 14. end if

15. //Work-group 间归约

16. 采用算法 2 的 work-group 间归约

3.2.3 work-group 间归约优化

归约算法中的 work-group 间归约主要负责完成对每一 个 work-group 在第二层中得到的局部归约结果的再归约操 作,最终得出原始数据集的归约结果。work-group 间归约共 有 3 种方法:1)将所有 work-group 得到的局部归约结果写入 位于 global memory 中的临时数组,然后再重新启动归约 kernel,进行递归归约操作,直至得到最终的归约结果。然而,由 于启动 kernel 是一个十分耗时的操作,因此不建议使用。 2)将局部归约结果临时数据回传至 CPU 内存中,在 CPU 端 完成最后的归约操作。但由于数据的回传需要经过 PCI-E 总 线,非常耗时,因此这种方法需要适当限制开启的 workgroup 数目。3)采用原子操作求得最终的归约结果,这也是 本文采用的方法。

本文在归约算法第三层的 work-group 间归约采用原子 操作的主要原因有两点:1)本文实现采用了线程内归约优化, 可大大减少开启的 work-group 数目,从而减少了需要进行 work-group 间归约的局部归约结果的数量;2)虽然开启的 work-group 数目较多,但在 GPU 目前的调度机制中能够同 时进行 work-group 间归约,调用原子操作的 work-group 数 量的最大值为硬件 CU 的个数。同时,因为这些 work-group 在最终执行时会存在一定的时间间隔,调用原子操作对性能 的影响会进一步较小。所以,相对于前两种方法,使用原子操 作来完成 work-group 间归约可大大提升整体性能。

3.2.4 其他优化方法

除此之外,本文还采用了以下通用优化方法。

1)指令选择

由于指令本身的特点和处理器元件数目的差异,不同指 令的吞吐量也存在差异,选择具有高吞吐量的计算指令可有 效提高 GPU 程序的性能。本文工作的指令选择优化主要体 现在:①使用位运算指令代替乘法和除法指令;②使用乘加指 令代替乘法和加法指令;③避免取余运算。

2)去除 clFinish()

本文实验设置命令队列中的命令是顺序执行的,由于 kernel执行后还需调度 clEnqueueReadBuffer()把最终归约结 果读回 CPU 端,此处存在隐式的同步操作,因此无需显式地 调用 clFinish()函数进行阻塞。阻塞函数十分费时,应尽量避 免使用阻塞函数。

3.3 PCL-Reduction

我们称采用以上优化方法的 GPU 归约算法为 PCL-Reduction。PCL-Reduction 采用了基于 stride 的线程内归约优 化、work-group 内归约优化和 work-group 间归约等优化方 法,算法性能得到了提升。

4 实验性能评估

4.1 实验环境搭建

4.1.1 硬件环境

本文实验选择两个不同厂商的 GPU 硬件测试平台:

AMD W8000 和 NVIDIA Tesla K20M。CPU 硬件测试平台 为 Intel Xeon E5-2620 v2。相关硬件平台的主要性能参数如 表 1 所列。

表1 主要性能参数

硬件平台	AMD W8000	NVIDIA Tesla K20M	Intel Xeon E5-2620 v2
主 频 / MHz	880	706	2100
计算核心/个	1792	2496	6
带 宽/(GB/s)	176	208	—
单精度浮点/TFLOPS	3.23	3.52	_

4.1.2 软件环境

本文提出的 GPU 归约算法采用 OpenCL1.2 和 CUDA 7.0 作为开发环境,并选取 OpenCV2.4.11 中的 CPU 版本, CUDA 版本和 OpenCL 版本的归约求和函数(分别简称为 CPU-Reduction, CUDA-Reduction 和 OpenCL-Reduction)作 为性能参照对象。其中,在 CPU 版本的编译中,采用了"-O3" 编译选项。OpenCV 库中 3 个版本均进行了精心优化,与它 们进行对比,本文的工作将更具说服力。

此外,在本文的测试中,实验数据类型为 unsigned int;若 无特别说明,实验采用的数据规模均为 2²⁴(用户可根据自身 应用需求将数据规模调整为其他任意大小的数据规模); work-group 内线程数目设定为 256。

4.2 总体性能评估

通过将本文的最终优化版本 PCL-Reduction 在不同的数据规模下分别与 OpenCV 库的 CPU 版本(CPU-Reduction)和 GPU 版本(CUDA-Reduction 和 OpenCL-Reduction)进行性能对比,来验证本文优化工作的高效性;并通过在 AMD W8000 和 NVIDIA Tesla K20M 两个不同厂商 GPU 上的实验,来验证本文工作能实现在不同 GPU 硬件平台间的高性能。

4.2.1 并行归约与串行归约的性能对比

本节在不同数据规模下,以加速比(串行归约时间/并行 归约时间)为性能评价标准,将本文的最终优化版本与 OpenCV库中归约求和算法的CPU版本(CPU-Reduction)进 行性能对比分析,结果如图4所示。





根据图 4,当数据规模较小(如小于 2¹⁴)时,OpenCV 的串 行归约算法的速度优于本文的最终优化版本。这是因为当处 理数据量过少时,开启的线程数量少,导致 CPU 不能充分隐 藏访存延迟,并且不足以充分利用 GPU 丰富的计算资源。 随着数据规模的增长,开启的线程数量逐渐增多,wavefront 通过相互间的切换来隐藏访存开销,使 GPU 保持高效的工 作效率,从而使性能得到提升。PCL-Reduction 无论在 NVIDIA GPU还是 AMD GPU上,均取得了良好的性能加 速。其中,在 NVIDIA Tesla K20M 上的加速比可达 2.97~ 20.24,在 AMD W8000 上的加速比可达 3.91~15.93。

4.2.2 并行归约算法间的性能对比

本节以归约函数运行时间作为性能评价标准,进一步在 两种 GPU 平台上将 PCL-Reduction 分别与 OpenCV 库中的 CUDA-Reduction 和 OpenCL-Reduction 进行性能对比分析, 以更合理地对 PCL-Reduction 算法的性能进行评估。

在 NVIDIA Tesla K20M 上, CUDA-Reduction, OpenCL-Reduction 和 PCL-Reduction 三者在 NVIDIA Tesla K20M 硬 件平台上的性能如图 5 所示。在不同的数据规模下, PCL-Reduction 的性能均优于 CUDA-Reduction 和 OpenCL-Reduction, 相比于 CUDA-Reduction 版本, 其性能加速了 2.25~ 5.97倍, 相比于 OpenCL-Reduction 版本, 其性能加速了 1.25~ 1.75倍。



图 5 GPU 归约算法在 NVIDIA Tesla K20M 上的性能对比 Fig. 5 Performance comparison of GPU reduction algorithm on NVIDIA Tesla K20M

在 AMD W8000 上,由于 CUDA-Reduction 函数无法运行于非 CUDA 架构的 GPU 上,因此只对 OpenCL-Reduction 函数和 PCL-Reduction 两者在 AMD W8000 硬件平台上进行了性能测试,结果如图 6 所示。在不同数据规模下,PCL-Reduction 的性能均优于 OpenCL-Reduction 函数,加速比达到 1.24~5.15。





依据图 5 及图 6 可知,通过在不同 GPU 硬件平台与 OpenCV 库的并行归约函数进行对比,PCL-Reduction 算法均 实现了较好的加速,具有更优的性能。GPU 编程的难点不仅 在于算法实现,更在于算法性能的可移植性。本小节在反映 PCL-Reduction 更充分地挖掘和利用了 GPU 潜在能力的同 时,也验证了其代码可移植性和性能可移植性。

4.3 性能评估

4.3.1 线程内归约优化

根据 3.2.1 节关于步骤 1 线程内归约过程的优化分析, 本节将对步长不同的两个线程内归约优化算法版本即 Global-Stride Kernel 与 Local-Stride Kernel 进行性能比较,以寻 找两者的性能差异。值得注意的是,Harris 提到以局部线程 数为跨步进行取数归约,未曾考虑以全局线程数为跨步进行 取数归约。在 2²⁰ 的数据规模下,通过在 AMD W8000 和 NVIDIA Tesla K20M上进行实验,来展示两者的性能特征, 实验结果如图 7 所示。



图 7 线程内归约算法在不同 GPU 平台上的性能对比

Fig. 7 Performance comparison of inner-threads reduction algorithm on different GPU platforms

需要注意的是,线程内归约粒度 times 为 1 时,两个 kernel 均将退化为 Naïve Reduction 的线程内归约过程,即只简 单地从 global memory 拷贝数据至 LDS,而不进行任何归约 操作。

由图 7 可知,在 NVIDIA Tesla K20M 平台上,Global-Stride Kernel 在多数情况下的性能略优于 Local-Stride Kernel;而在 AMD W8000 平台上,Local-Stride Kernel 在多数情 况下的性能优于 Global-Stride Kernel。因此,在不同的 GPU 架构下,采用不同的跨步对归约算法性能的影响是有差异的。 然而更值得引起重视的是,两种算法在不同平台上的性能曲 线的变化趋势是相同的,随着线程内归约粒度 times 的逐渐 增大,算法的运行时间呈现先下降后上升的趋势。以图 7 中 运行于 AMD W8000 平台上的 Global-Stride Kernel 为例,当 粒度 times 为 64 时出现拐点,粒度小于拐点粒度时,算法运 行时间随着粒度的增加而减少;当粒度大于拐点粒度后,算法 运行时间随着粒度的增力而增加。

性能提升的主要原因有 3 点:1)当数据规模足够大时,即 使每个线程处理多个数据,开启的线程数量也足以隐藏访存 延迟。2)并行归约算法在执行过程中数据量会逐渐减少,对 应的空闲线程也会逐渐增多。而在进行线程内归约时,随着 线程内归约粒度的增大,所需开启的线程数目逐渐减少,而每 一个线程的工作量随之增加,在减少了空闲线程数量的同时, 增加了每个线程的工作量,在一定程度上减缓了空闲线程"到 来"的时间。3)随着线程内归约粒度的增大,每个 workgroup 所负责归约的数据量越来越大,从而使得 work-group 的数目越来越少,减少了局部归约结果的数量。这样在一定 程度上减少了 3.2.2 节步骤 2 中的 work-group 内归约过程 (涉及同步操作)和 3.2.3 节步骤 3 中的 work-group 间归约 的原子操作,进而使性能得到提升。

然而,随着线程内归约粒度 times 的持续增大, work-

313

group 开启的数目越来越少。当 work-group 数目减少至无法 充分隐藏访存延迟或者无法充分利用 GPU 丰富的计算资源 时,算法性能将会降低,运行时间将会增加。因此,粒度 times 不能无限量地增大,理应综合考虑到 work-group 数目,以充 分利用资源。因此,起初线程内归约算法的性能会随着粒度 的增大而提升,当粒度增大至一定程度从而导致 work-group 数目过少时,性能变化将会出现拐点,此后算法性能将随着粒 度的增加而下降。在此,建议 work-group 的数目应至少为 CU 数目的 4~8 倍。

4.3.2 work-group 内归约优化

根据 3.2.2 节,对 work-group 内归约过程进行优化分 析。需要注意的是,work-group 内归约优化阶段的所有优化 版本均不包含线程内归约优化。本节实验以 Naïve Reduction 基础版本为参照,给出 work-group 内归约优化版本 Divergence-Free Kernel 和 Completely-Unroll Kernel 在 AMD W8000 和 NVIDIA Tesla K20M 上的性能对比详情,如图 8 所示。



图 8 work-group 内归约算法在不同 GPU 平台上的性能对比 Fig. 8 Performance comparison of inner-work-group reduction algorithm on different GPU platforms

根据图 8 的性能数据,与 Naïve Reduction 相比,workgroup 内归约优化算法通过解决 LDS 的 bank conflict 和 wavefront 内线程的条件分支,有效地提升了 LDS 的访问效 率及 wavefront 内部线程操作的并行性,从而提升了 kernel 的性能。循环展开优化是在 work-group 内归约优化版本 的基础上结合硬件调度和执行单元的特点进行优化的算法 版本。循环展开优化对 work-group 内归约过程进行循环展 开,一方面可减少归约过程的本地同步操作,进而减少时 间开销;另一方面消除了循环体内的冗余操作。因此,kernel 的性能可得到有效提升。最终,经过优化的 work-group 内归约版本相对于 Naïve 版本,在 NVIDIA Tesla K20M 平台 上的性能提升了 48.9%,而在 AMD W8000 平台上的性能提 升了42.2%。

因此,在不考虑线程内归约优化的情况下,通过对 workgroup 内归约过程的优化,能够使归约算法的整体性能得到 明显的提升。

4.4 归约算法的瓶颈分析

虽然 work-group 内的归约优化可有效地提升 kernel 的 性能,但是其并不是归约算法在 GPU 计算平台上实现和优 化的瓶颈,本文指出,线程内归约才是归约算法的性能瓶颈。 随着线程内归约粒度的不断增大,work-group 内归约优化对 整体性能提升的贡献越来越小。本节将以 PCL-Reduction 为 参考,对归约算法中线程内归约(以 Global-Stride Kernel 为 例)的重要性进行分析,进而衡量线程内归约优化在归约算法 性能提升部分所占的比重。需要特别注意的是,PCL-Reduction与Global-Stride Kernel的唯一区别在于是否包含 workgroup内归约优化,PCL-Reduction包含 work-group内归约优 化,而Global-Stride Kernel 仅仅进行了线程内归约。

下面将通过逐步增大线程内归约的粒度来分析 PCL-Reduction 与 Global-Stride Kernel 的性能差距变化,详情如图 9 和图 10 所示。



图 9 Global-Stride Kernel 与 PCL-Reduction 在 NVIDIA Tesla K20M 平台上的性能对比

Fig. 9 Performance comparison between Global-Stride Kernel and PCL-Reduction on NVIDIA Tesla K20M

图 9 为 Global-Stride Kernel 与 PCL-Reduction 在不同线 程内归约粒度下在 NVIDIA Tesla K20M 硬件平台上的性能 对比图。可以看出,当线程内归约粒度较小如 times = 1 时, PCL-Reduction 的运行时间明显少于 Global-Stride Kernel 的 运行时间,这说明在粒度较小的情况下,work-group 内归约 优化的效果明显。然而,随着线程内归约粒度的逐渐增大, Global-Stride Kernel 与 PCL-Reduction 的性能差距逐渐减 小,如当线程内归约粒度 times = 32 时,Global-Stride Kernel 所达到的性能与 PCL-Reduction 的性能相差不超过 1%。



图 10 Global-Stride Kernel 与 PCL-Reduction 在 AMD W8000 平台上的性能对比

Fig. 10 Performance comparison between Global-Stride Kernel and PCL-Reduction on AMD W8000

图 10 为 Global-Stride Kernel 与 PCL-Reduction 在不同 线程内归约粒度下在 AMD W8000 硬件平台上的性能对比 图。可以看出,图 9 与图 10 具有相同的规律,随着粒度的逐 渐增大,Global-Stride Kernel 的性能逐渐逼近 PCL-Reduction 的性能,如当粒度 times=16 时,两者的性能相差不到 1%。

由图 9 和图 10 可知,当逐步增大线程内归约粒度至合适 值时,线程内归约优化所取得的性能提升与 PCL-Reduction 几乎等同,即 work-group 内归约优化对归约算法的整体性能 结束语 本文以归约求和函数为例,详细叙述了归约算法在 GPU 计算平台上的并行设计与优化,并通过论证指出: 线程内归约为归约算法的瓶颈所在,是归约算法优化的关键 因素。本文提出的 PCL-Reduction 通过线程内归约、workgroup 内归约和 work-group 间归约 3 个层次的优化,最终获 得了非常好的性能提升。实验结果表明:在不同数据规模下, PCL-Reduction 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;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。

在实验过程中发现一个现象:不论在 NVIDIA Tesla K20M 还是 AMD W8000 上, OpenCL 程序 kernel 被重复调用多次,其第一次的运行时间开销大于之后的运行时间开销,可能的原因在于 OpenCL 程序第一次调用 kernel 时隐含了部分初始化工作。

未来我们希望能将本文工作应用于其他 OpenCL 平台, 如在 CPU, APU, Cell 等处理器上进行测试,进一步验证归约 算法的瓶颈所在和本文算法的性能。

参考文献

- [1] HARRIS M. Optimizing parallel reduction in CUDA[OL]. http://developer. download. nvidia. com/assets/cuda/files/reduction. pdf.
- [2] CATANZARO B. Opencl optimization case study:Simple reductions[OL]. https://developer. amd. com/article/opencl-optimization-case-study-simple-reductions.
- [3] YAN S G, YUNQUAN Z, GUOPING L. Reduction algorithm optimization based on the OpenCL[J]. Journal of Software, 2011,22(S2):163-171.
- [4] OpenCV Library Open Source Project [OL]. http://opencv. org.
- [5] BRADSKI G, KAEHLER A. Learning OpenCV: Computer vision with the OpenCV library[M]. O'Reilly Media.Inc., 2008.
- [6] RAWAT P S.RASTELLO F.SUKUMARAN-RAJAM A.et al. Register optimizations for stencils on GPUs[C] // Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18). New York, NY, USA, 2018.
- [7] CECKA C. Low communication FMM-accelerated FFT on GPUs

[C]// Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC' 17). ACM, New York, NY, USA, 2017.

- [8] ZHANG X X, TAN G M, XUE S B, et al. Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning[C]// Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'17). New York, NY, USA, 2017.
- [9] LI B,SUN J,ANNAVARAM M,et al. Elastic-Cache:GPU Cache Architecture for Efficient Fine- and Coarse-Grained Cache-Line Management[C]// Proceedings of 2017 IEEE International Parallel and Distributed Processing Symposium (IP-DPS). Orlando, FL, 2017.
- [10] KIRK D B, WEN-MEI W H. Programming massively parallel processors: a hands-on approach[M]. Newnes, 2012.
- [11] WU E H,LIU Y Q. The General Computing based on GPU[J]. Proceeding of Journal of Computer-Aided Design and Computer Graphics, 2004, 16(5):601-612.
- [12] OWENS J D, HOUSTON M, LUEBKE D. et al. GPU Computing[J]. Proceedings of the IEEE, 2008, 96(5): 879-899.
- [13] OWENS J D.LUEBKE D.GOVINDARAJU N.et al. A survey of general-purpose computation on graphics hardware[J]. Computer Graphics Forum: Journal of the European Association for Computer Graphics, 2007, 26(1):80-113.
- [14] Compute unified device architecture programming guide[OL]. http://www.nvidia.com/object/cuda_home.html.
- [15] Nvidia cuda C. programming guide[OL]. https://docs. nvidia. com/cuda/cuda-c-programming-guide.
- [16] NVIDIA cuda C. Best practices guide [OL]. https://docs. nvidia.com/cuda/cuda-c-best-practices-guide.
- [17] The opencl 1. 2 speciffication[M]. Khronos OpenCL Working Group, 2012.
- [18] STONE J E,GOHARA D,SHI G. OpenCL: A parallel programming standard for heterogeneous computing systems[J]. Computing in Science & Engineering, 2010, 12(1-3):66-73.
- [19] GASTER B, HOWES L, KAELI D R, et al. Heterogeneous Computing with OpenCL: Revised OpenCL 1 [M]. Newnes, 2012.
- [20] ZOU H, WANG H Q, HUANG Y. GPU Accelerated Rainbow Tables Analysis of MD5 Hash Password [J]. Journal of Chongqing University of Technology(Natural Science), 2013, 27(7):61-66. (in Chinese)
 邹航,王华秋,黄勇.基于 GPU 加速的彩虹表分析 MD5 哈希密 码[J]. 重庆理工大学学报(自然科学), 2013, 27(7):61-66.