

引力波 cWB 处理流水线的 GPU 加速

都志辉¹ 林璋熙¹ 顾彦祺¹ Eric O. LEBIGOT^{2,3} 郭翔宇¹

(清华大学计算机科学与技术系 北京 100084)¹ (清华大学信息技术研究院 北京 100084)²

(巴黎第七大学天体粒子与宇宙学实验室 巴黎 75205)³

摘要 引力波是爱因斯坦广义相对论的一个重要预言。大爆炸,特别是双黑洞、双中子星等双星系统是理论上最容易探测到的引力波波源。因为可以通过引力波了解这些重大的天体现象,所以对引力波的探测具有十分重要的科学意义。为此,建造了多个耗费巨资的基于激光干涉原理的引力波观测站(Laser Interferometer Gravitational-Wave Observatory, LIGO),以期能够首次直接探测到引力波。cWB(coherent Wave Burst)是一条能对多个观测站的数据进行实时分析处理的流水线。如何提高cWB程序的计算能力,成为了探测引力波的道路亟待解决的问题。在分析cWB流水线特点的基础上,找到其性能瓶颈,设计并实现了一种有效的并行方法,在具有很强并行处理能力的GPU硬件上实现了对cWB流水线的加速。实验结果表明,与原来基于SSE优化加速的CPU实现相比,该CPU可以达到至少10倍的加速,这对于实现多个站点引力波信号的实时处理具有重要意义,在实时数据处理技术上为使用高精度的探测设备发现引力波提供了支持。

关键词 引力波, cWB流水线, GPU, 并行处理

中图分类号 TP338 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.10.005

GPU Accelerated cWB Pipeline for Gravitational Waves Discovery

DU Zhi-hui¹ LIN Zhang-xi¹ GU Yan-qi¹ Eric O. LEBIGOT^{2,3} GUO Xiang-yu¹

(Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)¹

(Research Institute of Information Technology, Tsinghua University, Beijing 100084, China)²

(Laboratoire Astroparticule et Cosmologie (APC), Université Paris Diderot-Paris 7, Paris 75205, France)³

Abstract Gravitational wave (GW) is an important prediction of Einstein's general relativity theory. Some were generated during the big bang. Their most easily detectable sources are expected to be binaries of orbiting objects like black holes and/or neutron stars. Their study can thus give information about some important astrophysical objects. A few large-scale laser interferometer gravitational wave observatories have been built, with the goal of directly detecting GWs for the first time. Coherent Wave Burst (cWB) is an important pipeline that looks for gravitational wave in the data from multiple observatories, simultaneously and in real time. It is useful to improve the performance of cWB so as to allow it to perform deeper analyses. Therefore we analyzed a time-critical function from cWB, designed and implemented an efficient acceleration method on GPU. Experimental results show that our method can achieve at least 10x speedup compared with the original CPU implementation with SSE instruction. The results show that our GPU acceleration method is a viable option for improving gravitational wave data processing.

Keywords Gravitational waves, cWB pipeline, GPU, Parallel processing

1 介绍

引力波(Gravitational Wave)^[1-2]是时空的“涟漪”,爱因斯坦广义相对论(1916年)预言了它的存在。通过它可以了解黑洞碰撞和中子星碰撞的过程、超新星爆发的缘由以及宇宙大爆炸是如何发生的。若要探测到引力波,则需提供高达

10^{-21} 的度量精度,需要世界上最精密的仪器,很多年来一直被认为是根本不可能的事情。科学家 M. E. Gertsenshtein 与 V. I. Pustovoit^[3]的贡献彻底改变了人们的这一看法。由于引力波本身具备重大的科学意义,世界众多的一流研究机构与科学家都在围绕这一问题展开研究,力争成为引力波的发现者。西方发达国家先后投资上亿美元,建立了臂长从几百米

到稿日期:2016-10-22 返修日期:2017-02-11 本文受国家自然科学基金(61440057,61272087,61363019,61073008)资助。

都志辉 男,博士,副教授,主要研究领域为高性能计算, E-mail: duzh@tsinghua.edu.cn(通信作者); 林璋熙 男,主要研究领域为并行计算; 顾彦祺 男,主要研究领域为并行计算; Eric O. Lebigot 男,博士,助理研究员,主要研究领域为理论物理; 郭翔宇 男,硕士生,主要研究领域为 GPU 优化加速技术。

到几公里的第一代引力波观测站,为探测引力波提供了硬件设备基础,比如美国的 LIGO^[2]、意大利的 VIRGO^[4]、德国的 GEO600^[5]以及日本的 TAMA^[6]等。为了进一步提高探测到引力波的概率,Advanced VIRGO,Advanced LIGO^[11]等第二代观测站也在建设之中,其分辨率比第一代提高了至少 10 倍以上,同时对引力波流水线数据处理能力的需求提高了几十倍。因此,准确、快速地分析和处理引力波探测信号就成为发现引力波的一种核心技术。快速、实时地处理探测到的数据对于尽快发现引力波、验证引力波以及应用引力波非常必要,因此引力波流水线的高效处理算法以及软件的设计与实现就变得十分重要,它特别需要新型计算技术提供支持,GPU 加速便是其中之一。

随着芯片集成度的提高,处理器多核、众核化的发展趋势^[8]以及以 GPU(Graphics Processing Unit)^[9]为代表的加速器增强型异构体系结构的出现(2016 年 11 月在 TOP500 世界超级计算机排行榜^[10]上位居第一、我国自主研发的“神威·太湖之光”十亿亿次计算系统以及目前世界排名第二的“天河二号”和排名第三的 Titan 系统都采用了加速器增强型体系结构),使得计算机的硬件处理能力可以持续以指数方式快速增长。

GPU 加速的难点在于,必须针对新型硬件体系结构的特点对算法与代码进行全面的优化,才能够充分发挥这些硬件系统的计算能力。已有的 CPU 代码需要向 GPU 移植才能发挥 GPU 的计算能力。直接移植相对简单,但是若要取得比较理想的性能,则需考虑硬件的具体条件来进行计算任务的分配。由此可见,性能优化工作不仅必须,而且十分具有挑战性。

人们设计了不同的数据分析流水线来处理引力波数据,如 PyCBC, GstLAL 和 cWB。虽然已有不少流水线正在处理第一代观测站的数据,但是针对第二代观测站提出的高灵敏度、低延迟和大数据量处理的要求,已有的流水线难以满足。为此,本文在已有的成熟流水线 cWB (Coherent WaveBurst)^[11]的基础上,利用 GPU 加速部件来提高其执行速度,以满足第二代观测站在性能上的需求。cWB 流水线的主要原理是同时对多个观测站的数据进行小波分析,然后对得到的小波系数进行聚类来发现引力波信号。之所以选择 cWB,是因为它是一套成熟的软件系统,有近十年的历史,并且已经成功在全天空(all-sky)和全时(all-time)的引力波瞬态信号查找中^[12]得到应用。

目前已有的利用 GPU 加速引力波数据处理流水线的工作主要是针对另外一条 SPIIR(The Summed Parallel Infinite Impulse Response)流水线^[13]。SPIIR 是一种运用 IIR 技术的时域引力波搜索方式。虽然它们都是对引力波数据进行处理,但是这两条流水线由于处理算法截然不同,因此在实现上有很大的差异。本文针对 cWB 流水线提出加速处理的方法。

本文的意义在于可以为以引力波探测与发现为核心的重大国际合作科学活动提供强大的数据处理手段,使得应用科学家可以充分利用随时间呈指数增长的大型计算系统的计算能力来快速处理引力波信号,从而提高发现信号的准确度与可靠性,推进发现引力波的进程。

2 cWB 程序性能的分析

2.1 关键代码分析

为显著提高程序的性能,必须首先找到 cWB 程序中耗时最多的部分,然后针对该部分进行优化。

通过对 cWB 程序的主要 5 个计算阶段(read cluster, loadTDampSSE, supercluster, subNetCuts 和 WriteSparseTFmap)进行剖析与测量,得到各个部分的时间开销,如图 1 所示。

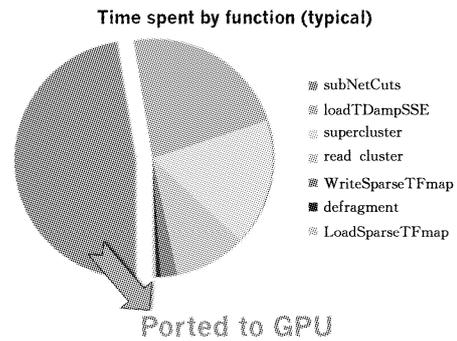


图 1 cWB 各部分的时间开销

如图 1 所示,subNetCuts 阶段的耗时在整个 cWB 程序中占比最大,因此如果首先针对这部分代码进行优化,就可以大幅提升整个流水线的性能。同时通过对这部分代码的深入分析,发现它不仅计算量大,而且逻辑简单,非常适合在 GPU 上并行。因此,本研究首先针对这一函数进行 GPU 加速。

2.2 subNetCuts 函数简介

首先介绍该函数中几个重要的参数。

sky location: 天空中的坐标。cWB 程序旨在探测宇宙中产生的引力波,而引力波来自四面八方。因此在 subNetCuts 函数中,程序将天空分成了许多位置(从 4096 到 196608 不等,由计算精度确定),并且每个位置用一个坐标表示,例如 sky location=0 就代表天空中的位置 0。

detector: LIGO 探测器的数量。现在 LIGO 总共拥有 3 个引力波探测器,因此 detector 参数的值为 3。

pixel: 像素点。LIGO 的 3 个探测器会不断地收集数据,收集到的数据就是这些像素点(pixel)。

cluster: 聚类。聚类包含了所有探测到的像素点(pixel),不同的聚类中 pixel 值可能不同。

v: 决定每个 cluster 中拥有 pixel 的数量,即如果一个聚类的 v 值为 4,那么该聚类拥有 4 个 pixel。v 的取值为几百不等。

lag: 数据包。每个数据包(lag)包含多个聚类,从几十到几百不等,即一个 lag 对应多个聚类。

HEALPIX: 决定计算的精度。计算的精度通过 sky location 的数目来体现。如果 sky location 的数目越多,则说明整个天空被划分得越细,计算的精度会越高,所花的计算时间也会越多。本文使用的 HEALPIX 的参数值为 7,在这个参数值下,sky location 的总数目为 196608。

下面介绍 subNetCuts 函数的功能。首先,cWB 将包围地球的天空分成了多个点,如图 2 所示。

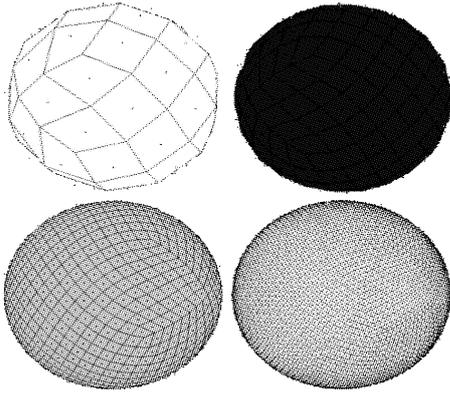


图 2 sky location 分布模型

LIGO 位于地面上的 3 个探测器在不断地收集数据,如果天空中某个位置的方向产生了引力波,那么这个引力波将会依次被 LIGO 的 3 个探测器探测到。如果地面上产生的某些干扰被探测器探测到,那么这些干扰只会被附近的一个探测器探测到而不会被其他两个探测器探测到,由此便可消除地面产生的干扰。subNetCuts 函数将各个探测器探测到的像素点(pixel)依据每个坐标(sky location)进行一定程度的迁移,以判断 3 个探测器是否都探测到了同样的数据。如果数据不同,则将其滤除;如果数据相同,则将其提取出来用于后面的运算。简单的说,subNetCuts 函数内部所做的运算是:循环计算一个 lag 内的多个 cluster,对于 cluster 内的 pixel,subNetCuts 函数需要根据每个 sky location 对 3 个 detector 探测到的 pixel 进行一定的迁移,以判断是否在某个 sky location 探测到了可能的引力波。计算的流程如图 3 所示。

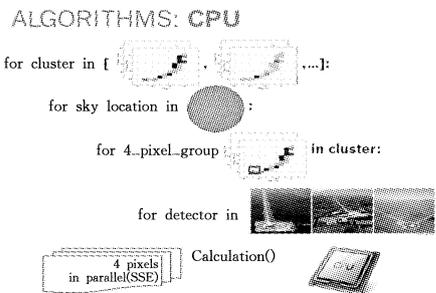


图 3 串行 CPU 版本的 subNetCuts 算法的流程

由于串行 CPU 版本的程序使用 SSE 指令集进行优化,因此在原版本中的计算是一次并行计算 4 个 pixel。此外,cWB 程序会对每个 lag 调用 subNetCuts 函数,因此 subNetCuts 函数会在整个 cWB 程序中被循环执行多次。

在明确加速的重点之后,下面进一步深入到 subNetCuts 内部进行分析。为简化分析,将 subNetCuts 函数算法划分为以下几个步骤。

- 1)DataPreparation:重复在各个 cluster 间读取 pixel 数据,并进行一些初步的计算,为 skyloop 准备数据;
- 2)skyloop:重复在每个 sky location 中对 3 个 detector 进行 pixel 的迁移计算;
- 3)PostSkyloop:根据 skyloop 的计算结果进行一些后续计算。

sky location 的数目 $l_e=196608$,cluster 的数量 K 在 $500\sim$

700取值,每个 cluster pixel 的数量为 $3\sim 400$,得到的各部分所占时间比例如图 4 所示。

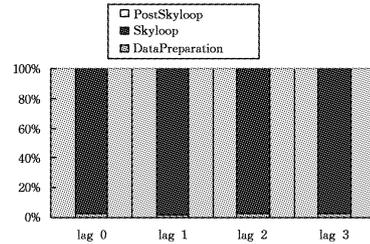


图 4 subNetCuts 函数在各运算阶段的耗时比例

从测试结果中可以看出,绝大多数的时间花费在 skyloop 这段计算中,这也是本文研究加速的重点。

3 GPU 加速方法的设计与实现

我们的算法在设计与优化上有如下特色:1)对于众多可以移植的模块,结合 GPU 的特点仅选取有显著加速效果的模块进行移植与性能优化,这样既减少了工作量,又可以取得显著的效果;2)对 CUDA 线程的计算粒度进行合理的选取,以取得优化的加速效果;3)结合应用的特点对 CUDA 块的维度进行优化,以提高资源利用率,特别地,结合 GPU 的特点对分支代码进行特殊处理以提高性能。

3.1 CPU 与 GPU 的任务划分

首先将 subNetCuts 算法划分为若干个连续的步骤,然后依照各步骤的特点决定由 GPU 或 CPU 来执行。在给 CPU 和 GPU 分配任务时,一般需要考虑如下因素:

- 1)任务的计算量和数据量。一般而言,计算量和数据量大的任务适合由 GPU 来完成,计算量和数据量小则无法充分发挥 GPU 的并行能力,因此这类任务应该交给 CPU 来执行。
- 2)尽可能地减少主机(CPU)和设备(GPU)之间传输的数据量和传输次数,并尽可能让每次数据传输做最多的计算。
- 3)考虑任务的逻辑复杂性。为提升性能,应安排 GPU 处理计算量大且逻辑简单的任务。相较于 CPU,GPU 拥有大量的计算单元,但其逻辑运算单元比 CPU 的少,因此 GPU 缺乏 CPU 处理复杂的逻辑判断与预测的技术手段。过多的逻辑判断会使程序由于 GPU 的判断机制被串行化,无法充分发挥 GPU 的并行计算能力。在 GPU 上如果遇到逻辑复杂的任务,应尽量用其他方式(如算数表达式)来减少判断语句的使用,或者尽量安排判断结果相同的线程在同一组 warp 内。如果同一 warp 中的线程的判断结果都为真或都为假,那么该 warp 内的线程会共同执行相应的分支语句而没有线程需要等待。

从前面对 subNetCuts 的时间分析可知,skyloop 前的数据准备部分和 skyloop 后的计算部分所需的计算量很少,并不能充分发挥 GPU 的性能,而且涉及复杂的数据结构,因此将这个部分安排在 CPU 上执行。skyloop 部分所需的计算量最大,可并行性也最高,因此被安排在 GPU 上执行。

为了尽可能提高计算性能,达到实时处理的效果,设计 skyloop 在 GPU 上的加速步骤如下:

1) 提取出每次计算中都不变的数据 FP, FX, ml, mm, V 和 tsize 进行处理。首先,将它们的数据充分合并,以减少数据传输次数和函数调用的开销(overhead);然后,一次性地将它们传输到 GPU 上,用尽量少的数据传输做最多的运算。另外,将本次数据传输放在程序的起始阶段,使得在传输数据的同时 CPU 也可以进行下一步的计算,从而有效地隐藏数据传输的时间。

2) GPU 设备会被隐式地进行初始化,由于传输的数据量较大,因此将数据存储到 GPU 的显存中,能有效减少传输的时间。

3) CPU 获取前 n 个 cluster 的参数,以此计算 skyloop 中所需的数据,得到 vtd, vTD, eTD 和 nr 等数据,并将其合并存储到预先分配的内存上。

4) 将前 n 个 cluster 的 vtd, vTD, eTD, nr 等数据传输到 GPU 的显存, GPU 执行这 n 个 cluster 的 skyloop 计算。

5) GPU 执行完本次的 n 个 cluster 的 skyloop 计算后,对结果执行并行的归约计算。

6) GPU 执行完本次的 n 个 cluster 的归约计算后,将 GPU 显存的部分数据清零。

7) GPU 执行完全部计算后,将结果传回到 CPU, CPU 以中断的形式执行本次 n 个 cluster 的后续计算。

8) CPU 获取下 n 个 cluster 的参数,并循环步骤 3)–7), 直到所有 cluster 计算完毕。

与原来的 CPU 处理相比,该过程节省了大量的时间(详细对比见 4.3 节)。

3.2 在 GPU 上的设计与实现

3.2.1 并行 subNetCuts 算法的设计

算法设计的关键在于确定 GPU 的并行粒度,即确定是基于每个 sky location 中的 pixel 的并行,还是基于 sky location 的并行。考虑每个 cluster 中 pixel 数量的不定性以及访存合并问题后,我们选择性能更高的一种,即每个线程计算一个 sky location 的并行粒度。基本算法描述如下。

算法 1 Alogrithm of subNetCuts

全局内存: vtd[nIFO], etd[nIFO], tmp, output

寄存器: pe[nIFO]

GridSize = blockDim. x * gridDim. x

ThreadId = blockIdx. x * blockDim. x + threadIdx. x

////////////////////////////////////

for (c=0; c<ClusterNumber; c++){

 得到该 cluster 的 vtd[nIFO]位置

 for (l=ThreadId; l<sky_location; l+=GridSize){

 得到该 sky location 的 etd[nIFO]的位置 pe[nIFO]

 通过 pe 计算 aa

 if ((aa-m)/(aa+m)<0.33)

 return;

 通过 vtd 和 pe 计算结果,并将其存入 tmp 数组中

 }

}

归约 tmp 数组中的结果,并将结果保存到 output 数组中

将 tmp 数组清零

3.2.2 CUDA 的 Block 和 Thread 维度设计

Block 和 Thread 维度设计是 GPU 加速的重点,直接决定了最终计算性能的提高程度。根据 CUDA 编程模型, grid 中的每个 block 都会被分配到 GPU 的一个 SM 上执行,分配到同一 SM 上的多个 block 按照随机的顺序执行。SM 中有多种执行单元:加载/存储单元、CUDA 计算核和特殊功能单元。因此,当一个线程块进行访存等非计算操作时(占用加载/存储单元),另一个线程块可以在该 SM 上执行运算(占用 CUDA 计算核),将多个这样的线程块称为活动线程块(active block)。为了形成各种形式的混合指令,令每个 SM 上至少拥有 2 个活动线程块,因此在一个线程块执行非计算操作时另一个线程块执行计算操作,可以更加充分地利用 SM 上的资源。但活动线程块的数量受限于每个线程块中线程的数量、每个线程占用的寄存器数量和每个线程块占用的共享存储器大小等 GPU 硬件条件。本文的硬件条件计算具体如下:

(1) 在基于 Tesla 架构的 GPU 中,每个 SM 至少需要 6 个活动线程束(active warp)才能有效隐藏内存或指令延迟。

(2) 在编译内核程序时添加-ptxas-options=-v 选项,可以在编译结束后观察到每个内核函数使用的 GPU 硬件资源的数量。

(3) 使用 CUDA SDK 中提供的 cudaDeviceProp 类来查询当前 GPU 的硬件资源^[10]。

(4) 计算每个 SM 上的活跃线程块与活跃线程束数量。由于 GPU 实际上是以线程束 warp 为单位调度线程的,而线程束的大小为 32,因此线程块内线程的数量最好是 32 的倍数才能保证每次调度都能充分调度 32 个线程,以免浪费 GPU 的资源。故线程的数量一般为 64~256。本文通过计算选择每个线程块执行 128 个线程。此时,每个线程块所需的资源如下。

1) 对于 kernel_skyloop

① 每个线程块所需共享存储器大小: $140+12=152$ Byte。

② 每个线程所需寄存器数量为: 32。

③ 每个线程块中 warp 的数量为: $128/32=4$ 。

再根据硬件资源($GT \times 480$)计算活跃线程块与活跃线程束的数量。

2) 对于 kernel_skyloop

① 由共享存储器所限,得到的活跃线程块数量为: $16384/152=107$ 。

② 由寄存器所限,得到的活跃线程块数量为: $(32768/128)/32=8$ 。

③ 由每个 SM 上可运行的 warp 所限,得到的活跃线程块数量为: $32/4=8$ 。

④ 由每个 SM 上可运行的线程块所限,得到的活跃线程块数量为: 8。

对于每个内核函数,取上述计算结果中最小的一个,因此对于本文设计的 kernel 函数,活跃线程块数量为 8。但一般情况下 grid 中线程块的数量应为 GPU 的 SM 数量与活动线程块数量乘积的数倍,这样才能充分利用 GPU 的并行能力。此外,程序执行的任务的特点也是决定开启线程块的数量的重要参考因素,因此本文选择的启用线程块和线程数量如下。

kernel_skyloop: 启用 64 个线程块, 每个线程块启用 128 个线程。

以上对 Block 和 Thread 的调度能充分发挥出硬件的计算能力, 使得整个加速流程有了底层的保障。

3.2.3 实现与优化方法

如前文所述, 在考虑了每个 cluster 的 pixel 数量的不定性以及访存合并的问题后, 选择每个线程计算一个 sky location 的并行粒度。该实现能够满足线程对全局内存的合并存取条件。为了进一步改进 kernel_skyloop, 本文还在数据访问、算法和 GPU 利用率 3 个方面做了详细的优化。

(1) 数据访问

1) 满足对全局存储器的合并访问条件是提高 CUDA 程序效率的最明显的因素之一。如前文所述, 本文在程序中每次读取全局内存时都尽可能地让线程能够访问连续的存储空间, 从而有效满足合并访问条件, 提高程序性能。

2) 为实现合并访问, 需将首地址对齐。通过在程序中将地址末尾补零, 使其长度满足 32 的倍数, 再在计算中滤去这些数据的影响, 使程序的内存读取符合首地址对齐条件。

3) 根据 CUDA 存储器模型中各个存储器的特点合理使用各个存储器, 以提高程序性能。例如将每个线程都相同的只读数据(En, Es, Lsky 等)放入常量存储器中, 以提高访问效率。另外, 因为大部分输入数据所占空间过大, 而且数据使用的重复率很低, 所以不适合采用共享存储器。

(2) 算法

1) CPU 和 GPU 之间的数据传输是影响 CUDA 程序效率的另一关键因素, 而传输的数据量是决定数据传输时间的重要因素。要使 CPU 和 GPU 之间的数据传输所占时间最少, 意味着需要找到程序中数据的发散点和约束点。由于数据的发散点和约束点之间一般进行了大量的运算, 因此这意味着从 CPU 迁移到 GPU 的代码量会增加, 此外还需考虑新增加的代码是否适合在 GPU 上运算, 权衡后找出最适合通过 GPU 优化的代码。本文最初仅迁移了瓶颈部分的算法到 GPU 上进行运算, 结果发现优化后数据传输所占的时间甚至还多于计算的时间。因此后来通过迁移足够多的代码到 GPU 上, 使 GPU 仅输出少量的最终结果而不是大量的中间结果, 大大减少了数据传输所占用的时间。

2) 影响数据传输时间的因素包括传输的数据大小以及调用数据传输函数所产生的开销。在传输的数据大小已经最小化的情况下, 通过减少数据传输的次数来减少所需的函数调用开销成为了减少数据传输所占时间的唯一手段。因此, 本文预先在 CPU 中分配了一段足够存储多个 cluster 输入的内存空间, 一次性地将多个 cluster 的输入传输到 GPU 上, 再在 GPU 上循环计算多个 cluster, 最后将多个 cluster 的结果一并传输到 CPU 上。通过这个方法, 程序有效地减少了 80% 的函数调用开销。

3) 相比于 CPU, GPU 拥有更多的计算单元, 但是它的逻辑控制单元非常少, 因此 GPU 处理分支语句的表现与 CPU 相差甚远, 除非对一个 warp 内的所有线程执行某次条件判断语句产生的结果都相同, 否则会影响 CUDA 程序的性能。因此在 CUDA 程序中, 本文尽量避免使用条件判断语句。

例如: 原分支判断语句:

```
if(AA > stat) {
    stat=AA; Lm=L0; Em=E0; Am=aa; lm=l; Vm=m; suball=
    ee; EE=em;
}
```

改为:

```
msh=(AA > stat);
stat=stat-msh+AA*msh;
Lm=Lm-Lm*msh+L0*msh;
Em=Em-Em*msh+E0*msh;
Am=Am-Am*msh+aa*msh;
lm=lm-lm*msh+l*msh;
Vm=Vm-Vm*msh+m*msh;
suball=suball-suball*msh+ee*msh;
EE=EE-EE*msh+em*msh;
```

以此, 程序便能充分发挥出 GPU 计算的能力, 避开其在逻辑运算上的不足。

(3) GPU 利用率

通过增加 GPU 上活跃线程块的数量, 可以使得 GPU 上并行执行的线程数量增加, 提高对 GPU 的利用率, 从而可以相应地提升程序的性能。活跃线程块数量与 GPU 上的硬件资源限制和程序所需硬件资源有关。通过上文的分析发现 kernel_skyloop 函数部分对寄存器的需求很大, 而这限制了它的活跃线程块的数量。因此, 本文采用编译命令——maxrregcount 来限制 kernel_skyloop 使用的寄存器个数。经过测试, 当 kernel_skyloop 函数的每个线程使用 32 个寄存器(原先为 64 个)时, 程序效率达到最高。这使得程序的运行效率在原基础上提升了 7%。

4 实验结果

4.1 测试环境与测试数据

实验中使用的 CPU 为 Intel® Core i3-2100, 主频为 3.10GHz, 主机内存为 2GB, 操作系统为 Scientific Linux 6; 使用的 GPU 为 Geforce GTX 480; 使用的 CUDA 版本为 CUDA 6.0。CPU 版本的 subNetCuts 函数算法基于 C++ 实现, 实现过程中使用了 SSE 指令集, 因此具有一定程度的并行性。CPU 版本使用的编译器为 g++, CUDA 程序使用的编译器为 nvcc。

测试使用的参数为 HEALPIX=7, setTDFilter=4。测试的数据为 LIGO 提供的 coherence_938027208_904_S6B_BKG_LF_L1H1_2G_SUPERCLUSTER_run2a_job10.root。基本的参数如表 1 所列。

表 1 测试参数

参数名称	参数值
sky location	196608
Cluster	500~700
V	3~300
Lag	10

其中, sky location 的数值非常大, 可以很好地体现出 GPU 强大的计算能力。程序将计算每个数据集(lag), 每个数据集集中有数个聚集(cluster), 每个聚集集中有 v 个像素点(pi-

xel),每个像素点要计算多个位置(sky location)。

4.2 GPU 与 CPU 计算结果的对比

对每个优化程序来说,首先要保证结果的正确性。本文是对原算法进行改进,因此判断程序正确性的方法就是对比改进后的算法和原算法的输出。输出结果如表 2 所列。

表 2 CPU 与 GPU 计算结果的对比

变量名	数值相同数目/总数目	相同率/%
Stat	387/387	100
Lm	365/387	94.3
Em	387/387	100
Am	387/387	100
Suball	387/387	100
EE	387/387	100
Lm	387/387	100
Vm	387/387	100

从表 2 可知,基于 CPU 的算法和基于 GPU 的算法的结果非常接近,不同之处主要在于 Lm 这个变量。Lm 变量代表的意思是 sky location of max statistics,即 stat 值最大的那个 sky location 点。这个数值的误差率为 $500/196608 * 100\% = 0.002\%$,且这种误差并不会对后续的计算产生影响,因此本文程序结果的正确性是有保证的。

4.3 各步骤效率的对比

4.3.1 数据准备阶段 DataPreparation 的效率对比

算法在进入 skyloop 计算前,需要为 skyloop 计算准备数据。本文测试了两种算法进入 skyloop 计算前的计算时间,结果如图 6 所示,从中可以看出,使用 GPU 计算时也会产生额外开销。

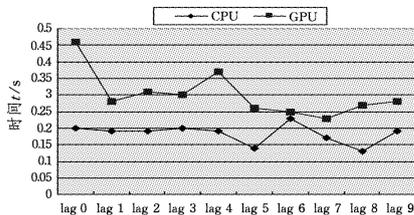


图 6 GPU 与 CPU DataPreparation 计算时间的对比

可以看出,在数据准备阶段,GPU 版本的运行时间大于 CPU 版本的运行时间,造成这个情况的因素主要有:

1)GPU 初始化所占用的时间。GPU 在程序第一次调用 GPU 上操作的相关函数时会进行隐式的初始化,该过程花费的时间大约为 0.1s。但这个初始化只会在计算第一个数据集时进行。

2)程序在 CPU 和 GPU 分配内存空间时所需的开销。在程序将 CPU 上的数据传输到 GPU 之前,需要在 CPU 和 GPU 上分配足够的内存空间。该空间在程序执行的过程中一直存在,直到计算完成后由程序显式地释放。

3)为了减少数据传输次数且使数据符合 GPU 合并访问模式而对数据进行预处理产生的开销。例如将数据进行压缩、末尾补零使数据对齐等。

由此可见,在利用 GPU 强大的计算能力之前,程序必须做一些额外的工作,这些额外的工作会消耗更多的时间。如果程序加速后的效果不能弥补程序做这些额外工作所产生的开销,那么这个程序就不适合用 GPU 进行加速。这也是计算量小的程序并不适合用 GPU 进行加速的原因。

4.3.2 skyloop 计算阶段的效率对比

本小节主要比较 CPU 进行 skyloop 计算所需的时间和 GPU 进行数据传输、skyloop 计算所需的时间,这也是本文最为关键的部分。测试结果如图 7 所示。

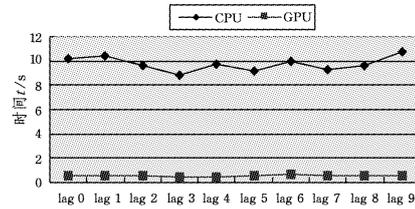


图 7 GPU 与 CPU skyloop 计算效率的对比

从图 7 可以看出,基于 GPU 的 skyloop 计算的性能提升明显,加速比达到 15~21 倍左右。由于本文采用一个线程处理一个 sky location 的模式,并且在整个程序中都充分考虑到 GPU 全局内存的合并访问规则并避免了分支语句,因此充分发挥了 GPU 的计算性能。另外,数量庞大的 sky location 令 subNetCuts 函数的计算量大大增加,这是 CUDA 程序能取得可观的加速比的关键。

4.3.3 后续计算阶段 PostSkyloop 的效率对比

本小节主要比较 CPU 进行 skyloop 后续计算阶段 PostSkyloop 所需的时间和 GPU 计算所需的时间。测试结果如图 8 所示。

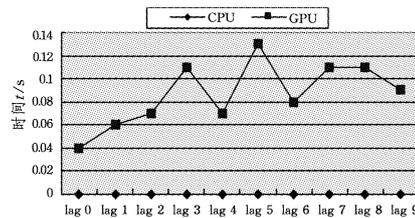


图 8 GPU 与 CPU PostSkyloop 计算时间的对比

从图 8 可以看出,CPU 的 PostSkyloop 部分几乎不花费时间,因此在图中看不到 CPU 的曲线。而 GPU 却耗费了一些时间,造成这个结果的原因有两点:

1)从 GPU 传回的数据是经过压缩存储的数据,为了从中取回原数据,CPU 需要做一些冗余的计算,这部分会多花一定的时间。

2)在 subNetCuts 函数每次调用完成之后,需要调用 CUDA API 来释放分配在 GPU 和 CPU 上的内存,这部分的调用也存在一定的开销。

4.4 算法总体效率的对比

将 GPU 上的 subNetCuts 算法命名为 GPU-subNetCuts,将 CPU 上的 subNetCuts 算法命名为 CPU-subNetCuts,令 sky location=196608,分别在 10 个数据集上测试算法的运行总时间,结果如图 9 所示。

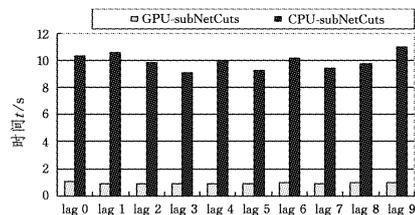


图 9 GPU 与 CPU subNetCuts 计算时间的对比

subNetCuts 函数的加速比如图 10 所示。

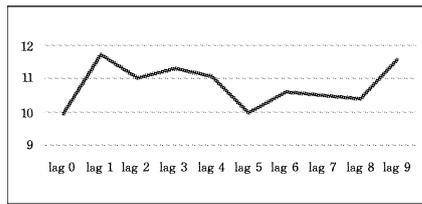


图 10 subNetCuts 函数的加速比

从图 9 中可以看到, GPU-subNetCuts 的运行时间明显减少。从图 10 中可以看到, GPU-subNetCuts 在 10 个数据集上均取得了 10 倍以上的加速效果。

结束语 本文基于 GPU 实现了 cWB 程序的优化,并在真实环境下通过性能优化达到了 10 倍以上的加速。本文主要的贡献有:

1) 分析了 cWB 程序的各部分耗时,找出了效率瓶颈 subNetCuts 函数,并对函数的算法进行了充分的分析。以实际的数据集测试了原算法中各步骤的运行时间,为算法优化找准了方向。

2) 根据 subNetCuts 函数中各计算步骤的特点和耗时,结合 CPU 和 GPU 的计算特点,合理分配计算任务到 CPU 和 GPU 上;并且基于 GPU 的并行模式为 subNetCuts 函数中的 skyloop 计算步骤确定了并行粒度。

3) 找出所编写的 CUDA 程序的瓶颈,对程序进行进一步的优化。在减少 GPU 与 CPU 数据传输量方面,本文找到了数据的发散点和约束点,迁移足够多的代码到 GPU 上,减少了传输的数据量。在减少函数开销方面,通过数据压缩减少数据传输次数以及内存分配次数。在存储器访问方面,合理安排线程和函数以符合内存合并访问规则,合理利用 GPU 提供的各种存储器。在减少分支语句方面,使用布尔运算避开逻辑判断,保证程序的并行程度。

本文方法还存在以下可以被进一步改进之处:

1) 虽然本文最后设计的算法是在 GPU 上并行计算、由 CPU 和 GPU 协同完成的算法,但是该算法在 CPU 和 GPU 之间的计算却是串行的。因此如果能让任务的执行在 CPU 和 GPU 之间并行,即在 GPU 计算本次的 n 个 cluster 的数据的同时 CPU 能够为 GPU 的下 n 个 cluster 的计算准备数据,则能一定程度地提升程序的性能。在理想情况下,这样的算法可以达到隐藏 CPU 计算时间的效果,提升 20% 的程序效率。

2) 依据本文的算法,在每次调用 subNetCuts 函数的最开始,程序要在 CPU 和 GPU 上分配足够的内存空间,且在每次 subNetCuts 函数调用结束时程序也要释放 CPU 和 GPU 上的内存空间。然而 subNetCuts 函数会被调用上百次,而每次程序分配的内存空间的大小几乎相同,因此程序在该过程中

做了上百次的重复操作。所以可以考虑将分配内存空间的函数放在 subNetCuts 函数的调用处,而非 subNetCuts 函数内。这是下一步的优化工作。

3) 在优化了 subNetCuts 后,我们将进一步深入考查其他计算部分在 GPU 上实现的开销,通过综合平衡选择下一个在 GPU 上优化实现的功能。

参考文献

- [1] FINLEY, DAVE. Einstein's gravity theory passes toughest test yet; Bizarre binary star system pushes study of relativity to new limits[N]. Phys. Org., 2013.
- [2] THORNE K. Gravitational Radiation in 300 Years of Gravitation[M]. HAWKING S W, ISRAEL W, eds. Cambridge University Press, 1987.
- [3] GERTSENSHTEIN M E, PUSTOVOIT V I. On the detection of low-frequency gravitational waves[J]. Soviet Journal of Experimental and Theoretical Physics, 1963, 16(2): 433.
- [4] CARON B, DOMINJON A, DREZEN C D, et al. The virgo interferometer[J]. Classical and Quantum Gravity, 1997, 14(6): 1461.
- [5] LUCK H, et al. The geo600 project[J]. Classical and Quantum Gravity, 1997, 14(6): 1471.
- [6] ANDO M, ARAI K, TAKAHASHI R, et al. Stable operation of a 300-m laser interferometer with sufficient sensitivity to detect gravitational-wave events within our galaxy[J]. Phys. Rev. Lett., 2001, 86: 3950-3954.
- [7] HARRY G M, et al. Advanced ligo; the next generation of gravitational wave detectors [J]. Classical and Quantum Gravity, 2010, 27(8): 084006.
- [8] ASSNOVIC K, BODIK R, DEMMEL J, et al. A view of the parallel computing landscape [J]. Communications of the ACM, 2009, 52(10): 56-67.
- [9] OWENS J D, et al. A survey of general-purpose computation on graphics hardware[J]. Computer Graphics Forum, 2010, 26(1): 80-113.
- [10] TOP500 list of the world's most powerful supercomputers [OL]. <https://www.top500.org>.
- [11] KLIMENKO S, et al. Coherent method for detection of gravitational wave bursts[J]. Class. Quantum Grav., 2008, 25: 114029.
- [12] ABADIE J, et al. All-sky search for gravitational-wave bursts in the second joint LIGO-Virgo run[J]. Phys. Rev. D, 2012, 85: 12200.
- [13] LIU Y, DU Z H, CHUNG S K, et al. GPU-accelerated low-latency real-time searches for gravitational waves from compact binary coalescence[J]. Classical and Quantum Gravity (CQG), 2012, 29(23): 235018.