**声子计算在GPU上的性能优化技术设计与实现**

摘 要

ShengBTE是计算玻耳兹曼声子输运方程常用的软件之一。虽然该程序可以计算声子散射率的收敛集，并利用它们来获得声子散射率κℓ和许多相关量[1]，但是它的计算过程漫长。我们需要对代码进行GPU移植来实现加速效果。本文深入分析了性能瓶颈产生的原因，提出了相应的代码优化方案。实验结果表明，该方法在不降低精度的前提下，实现了单温度测试下最高4.62倍的加速。

关键词：玻耳兹曼声子输运方程，ShengBTE, 性能分析优化，GPU

**Towards the GPU Acceleration of Phonon Computation in ShengBTE**

**Abstract**

ShengBTE is a software package for computing the Boltzmann transport equation. Though the program can compute converged sets of phonon scattering rates and use them to obtain κℓ and many related quantities, it takes a long time to compute. We need to port the code on GPU to achieve acceleration effect. In this paper, we thoroughly analyze the cause of the performance bottleneck and propose corresponding optimization for performance speedup. The experiment results show that our approach achieves a maximum speedup of 4.62x without degrading the precision.

**Key words:** Boltzmann transport equation, ShengBTE, Performance analysis and optimization，GPU

# Introduction

对于热电[2]、热管理[3]和基于相变材料的非易失性存储器的开发[4]等需要具有特定导热系数材料的重要技术来讲，晶格热导率κℓ是一个重要的参数。晶格热导率是声子对总热容的贡献，而研究固体声子输运的一个重要方法是玻耳兹曼输运方程[5]。

ShengBTE是Wu等人提出的一个新的声子玻耳兹曼输运方程的求解软件包。ShengBTE对于寻找具有目标导热性能的新型材料，以及深入理解固体中的热传导实验测量具有重要的价值[1]。该程序可以计算声子散射率的收敛集，并利用它们来获得声子散射率κℓ和许多相关的量。此外，它还实现了作者中的一些人开发的一种近似，以有效和准确地预测纳米线的导热系数。ShengBTE完全由FORTRAN语言构成，是一个仅能运行在CPU上的程序。通过全面的性能分析，我们发现了几个性能瓶颈，并且提出了一部分代码优化与GPU加速方法，显著提高了ShengBTE计算速度。

具体而言，本文的贡献如下：

-分析了ShengBTE的性能，确定了Ind\_plus与Ind\_minus是计算瓶颈。

-提出了几种提高性能的优化策略，包括并行化，GPU移植与kernel blocksize调整。

-在应用该算法后，我们对ShengBTE的性能进行了评估，程序在不降低迭代次数与计算精度的前提下，实现了最高4.62x的加速。

本文的其余部分组织如下。在第2节中，我们阐述了声子计算软件发展背景以及GPU加速在科学软件的应用。在第3节中，我们提供了ShengBTE的瓶颈分析。第4节给出了热点函数的优化策略。第5节阐述了实验环境，并对实验结果进行了分析。第6节介绍了相关工作，第7节给出了本文的结论。

# Background

## 2.1 Phonon Computation

声子计算主要基于三种方法：Density functional perturbation theory (DFPT)、Frozen phonon method、Finite displacement method。1987年，Baroni、Giannozzi和Testa提出了微扰密度泛函方法(Density Function Perturbation Theory)[5]作为一种新的晶格动力学性质计算方法，没有晶胞大小的限制，因此可以应用到复杂材料性质的计算上，一经提出就被广泛应用到了半导体、金属和合金、超导体等材料的计算上。Frozen phonon method在1982年已经成功地使用[6]，优点是计算简便，缺陷在于它要求声子波矢与原胞边界(super size)正交，或者原胞足够大使得Hellmann-Feynman力在原胞外可以忽略不计。于是对于复杂系统的超晶胞计算量急剧增加。Finite displacement method[7,8]可以节约RAM和CPU时间，但是需要生成supercell来获得比较可靠的力常数。

## 2.2 GPU Acceleration on Scientific Applications

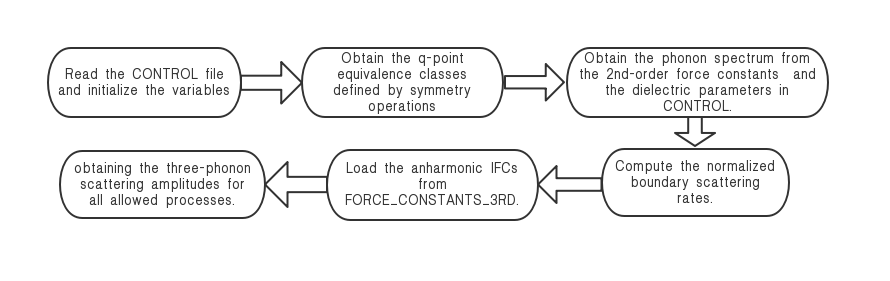
科学应用由于其计算量的巨大性，耗时漫长，迫切需要使用更快速的计算硬件。每年的ASC、SC、ISC世界三大超算比赛都设置了科学应用赛题的优化加速以及固定的计算性能测试程序HPL和HPCG。通过HPL和HPCG测试程序，已经明显的看出GPU在某些计算上可以达到CPU的十几倍，GPU用于优化科学应用也成为了超算比赛的趋势。

事实上，一些科学应用已经加入了GPU计算模块。量子化学领域的Abinit[9]，加入了LOBPCG算法的第一个GPU实现[10]；生命科学领域的relion[11]，使用GPU加速cryo-EM结构后发布了relion2[12]。NVIDIA官网给出了所有拥有GPU加速的应用列表gpu-applications-catalog.pdf[13]。这些应用或多或少都进行了计算密集算法的GPU移植，并且得到了几倍到几十倍的加速比。图【】展示了2014年至今的使用GPU的科学应用数量变化，这一数量还在不断增加。

# Bottleneck Analysis

## 3.1 Execution Flow Analysis

ShengBTE运行需要三个输入文件，CONTROL，FORCE\_CONSTANTS\_2ND和 espresso.ifc2中的一个，以及FORCE\_CONSTANTS\_3RD。ShengBTE的运行不需要额外参数，所有的配置均写入在CONTROL文件里。它的运行流程是大致如图【】。



其中obtaining the three-phonon scattering amplitudes for all allowed processes是计算最昂贵的部分。

3.2 Bottleneck Identification

为了进一步了解ShengBTE执行过程中的行为，我们在本地服务器上运行算例，并且使用Intel的性能分析工具Vtune[14]分析了计算温度为300以及区间为300~900步长100条件下的热点函数。通过观察发现七个算例的热点基本一致，大部分都是Ind\_plus和Ind\_minus，还有少部分的MPI同步与等待时间以及极少量的其他函数运行时间。

以Sn2Bi-F为例，我们给出瓶颈分析图。如图【a】所示，300k条件下，Ind\_\*函数占据了75%的执行时间，热点很突出，多进程执行时产生的同步等待以及规约时间占据了23%，其余的仅占2%。在连温条件下，热点函数如图【b】，相对于单温度测试条件热点函数比例没有太大变化。在ShengBTE中，程序通过Ind\_driver循环调用Ind\_plus和Ind\_minus函数，这部分是计算密集的区域。MPI中有一部分产生在Ind\_driver循环的规约，还有一部分是输入处理的同步等待。移植Ind\_\*函数到GPU，不但可以缩短函数本身执行时间，同时会缩短同步与等待时间。

# GPU Acceleration Strategies

## 基于第3节的瓶颈分析结果，我们提出了几种优化策略，包括并行化，GPU移植以及kernel blocksize的调整来缓解计算瓶颈。

## 4.1 Eliminating the Loop-carried Dependency

如算法【】所示，在热点函数Ind\_plus里计算密集的部分是一个三重循环，但是在每次循环里N\_plus\_count这个量都有一定几率加一，即循环间存在依赖关系。我们要把函数做到GPU并行化就必须消除这个依赖。为了使每次循环之间没有依赖，我们认为N\_plus\_count在每次循环都会加一，直接取值为k+Nbands\*(ii+j\*nptk)。同时，扩大ndof2ndPhonon\_plus， Indof3rdPhonon\_plus，Gamma\_plus数组空间为Nbands\*nptk\*Nbands。在满足计算条件时，Indof2ndPhonon\_plus[N\_plus\_count]得到的是一个非零值，故我们初始化Indof2ndPhonon\_plus数组为零值，在循环结束后以非零值为依据将三个稀疏数组稠密化。

|  |  |
| --- | --- |
| **算法 A1** Ind\_plus | |
| **1:**  **2:**  **3:**  **4:**  **5:**  **6:**  **7:**  **8:**  **9:**  **10:**  **11:**  **12:**  **13:**  **14:**  **15:**  **16:**  **17:**  **18:**  **19:** | *do j=1,Nbands*  *do ii=1,nptk*  *...*  *do k=1,Nbands*  *...*  *if ((omegap.ne.0).and.(omegadp.ne.0)) then*  *...*  *if(abs(omega+omegap-omegadp).le.(2.d0\*sigma)) then*  *N\_plus\_count=N\_plus\_count+1*  *Indof2ndPhonon\_plus(N\_plus\_count)=(ii-1)\*Nbands+j*  *Indof3rdPhonon\_plus(N\_plus\_count)=(ss-1)\*Nbands+k*  *...*  *Gamma\_plus(N\_plus\_count)=hbarp\*pi/4.d0\*WP3\*abs(Vp)\*\*2*  *Gamma\_plus(N\_plus\_count)=Gamma\_plus(N\_plus\_count)\*5.60626442\*1.d8/nptk*  *end if*  *end if*  *end do*  *end do*  *end do* |

唯一产生的缺陷是调用时传输的数据量增加，使用GPU时显存占用会有少量增加。但是经过我们测试发现数据传输在ShengBTE中没有成为性能限制的瓶颈，采用增加数据传输达到整体更快的加速效果是值得的。

## 4.2 Accelerating the Kenerls

经过4.1小节消除依赖以后，我们得以将Ind\_plus与Ind\_minus函数设置为cuda的内核调用。我们通过在Ind\_driver函数里调用C层代码，然后调用cuda来实现GPU的利用。在GPU上，我们将三层循环实现为并行，并且取得了不错的加速效果。

(待写)

## 4.3 Scaling to Multiple GPUs

（介绍多GPU的实现方案，如果没有太多内容要写，可以合并到3.2小节）

## 4.4 Adjusting the Block Size

一个CUDA代码包含的函数主要可以分为两部分:CPU(主机)运行的函数和GPU(设备)运行的函数。在设备上运行的函数也称为内核，都是以\_global\_开头。当CPU上的CUDA程序调用内核时，将生成一个线程网格。这个网格由块组成，每个块由线程组成。

所以在内核调用前必须指定在设备执行的函数的栅格和块的维数，栅格所有维数大小乘积等于被发送的块的数量，而块的所有维数大小乘积等于每个块的线程数量。针对4.1节所述三层循环，假设块大小为(block.x,block.y,block.z)，我们设定了栅格大小为((Nbands+block.x-1)/block.x, (nptk+block.y-1)/block.y, (Nbands+block.z-1)/block.z )，达到自动最优化使用GPU资源。可能是由于三层循环每层大小存在巨大差异，此时我们发现block三个维度不会在相同时达到较优效果，而是调整到极性差距时得到了最快运行速度，这个调整提升了大概20%的加速效果。

kernelName <<< gridSize , blockSize >>> (inputParameters);

# Evaluation

## 5.1 Experimental Setup

所有的实验都在本地服务器上进行，使用14进程，机器型号为【？浪潮M4，软硬件配置如下：

|  |  |  |
| --- | --- | --- |
| 操作系统 | 硬件 | 软件 |
| CentOS7.6  (内核3.10.0 x86 64) | CPU 2xIntel E5-2680v4  GPU 2xNvidia P100  Mem 24x16G | GCC v4.8.5  ICC v2018.5.274  CUDA v10.0  IFORT v2018.5.274  Intel VTune v2018.4.0.573462  Intel MPI v2018.4.274 |

## 5.2 Performance Improvement on Kernels

我们在4.2小节进行了Ind\_\*函数GPU kernel化移植。Ind\_plus和Ind\_minus函数在Ind\_driver函数里循环被调用，于是我们统计了整个循环的执行时间，即kernels在单块GPU上执行的总时间，然后与这个循环在单块CPU上的执行时间进行对比。如图【？】所示，在大多数算例上，kernel的加速效果都很明显。

## 5.3单温计算

我们测试了所有优化后ShengBTE运行性能的提升。我们在CPU，CPU+GPU和CPU+2\*GPUs模式下进行了对比试验，图【时间，加速比a】展示了在温度为300条件下使用不同的七个算例计算得到的时间与加速比。单块GPU对比单块CPU最高得到了4.62x的加速比，但是一些算例表现仅有1.44x。同时GPU之间的扩展带来了额外的1.5~1.8倍的加速效果。这些算例部分来自ASC19，部分来自almaBTE的算例，所有算例已经上传到【github？】。对于加速比不高的算例，分析源代码与Vtune收集的数据发现penta-graphene不是一个计算密集的算例，在Ind\_\*函数里调用Vp\_\*热点函数的次数远远不如其他算例。这一点在5.2小节的kernel加速比中可以得到验证。总体来看，平均加速比已经达到了3.x。

## 5.4 连温计算

同时，ShengBTE支持连续温度下的计算，我们使用Sn2Bi-F算例进行了以300K开始的步长为100K的多次运行测试。图【】是连温条件下在单块CPU和单块GPU上的运行时间与加速比。从图中可以看出，随着运行温度区间越来越长，GPU得到的加速比越来越高并趋于平稳。这是由于输入处理占据了一定时间，但是无论是单温度测试还是多温度测试输入处理只有一次，故温度区间越大输入处理时间百分比越少，加速效果会更明显。

## 5.5 Roofline Model Analysis

（CPU、单卡，以及kernel和整体分析，roofline参考游心论文）

## 5.6 Parameter Sensitivity Analysis

我们移植的Ind\_\*函数三层循环值分别为Nbands、nptk、Nbands，故我们使用了一个三维的block并且保证了x和z维度值相同。如图【】，我们在三个维度的值都选用了对所有算例都比较友好的2的幂，最终调整至1x64x1取得了最佳计算效率。

# Related Work

almaBTE[15]是ShengBTE的继承者。almaBTE编译产生了一组可执行文件，允许用户灵活使用。同时almaBTE保留了一个ShengBTE的模拟器，可以处理与ShengBTE相同的输入。almaBTE产生的模拟器由FORTRAN与C混合生成，而ShengBTE完全是FORTRAN生成。我们尝试运行了多组温度下的almaBTE，发现单温度测试时alamBTE并没有比ShengBTE有运行时间上的缩短，甚至运行时间更长了。但是在连续温度下的计算时，almaBTE可以更有效地缓存与温度无关的量，每个温度计算可以有3倍的加速比。

# Conclusion and Future Work

随着材料技术发展，新的特定性能材料的寻求对现有软件计算性能的优化起到了重要的推动作用。本文对ShengBTE进行了综合性能分析，找出了性能优化的瓶颈。此外，我们还提出了一些提高ShengBTE性能的优化策略，包括并行化，GPU加速与kernel blocksize调整。实验结果表明，在不降低精度的前提下，我们的优化方法实现了单温度测试下最高4.62x的加速。在未来的工作中，我们希望进行knl重核CPU的移植以及次要热点函数的GPU移植。