激光等离子体相互作用模拟的并行和加速研究

武海鹏[1] 文敏华[1] Simon See[2] 林新华[1, 3]
[1]上海交通大学 高性能计算中心 上海 200240
[2]NVIDIA Technology Center 新加坡
[3]东京工业大学 学术国际情报中心 日本
wenminhua@sjtu.edu.cn

摘要：
随着生成超短激光脉冲的技术不断发展，对这种激光脉冲和等离子体相互作用进行动力学描述也变得越来越重要。Particle-In-Cell是一种在等离子体物理中研究充能粒子在电磁场中的运动轨迹的广泛应用的方法。尽管现在已经有一些在GPU上的PIC方法的实现，但是基于激光等离子体相互作用模拟的特点，仍然有很多重要问题可以尝试一些其它解决思路。这篇论文的三个主要贡献是：(1) 提出了一种把初始的基于CPU的LPI模拟代码完整移植到GPU上的可行方法。(2) 提出了一系列加速初始的GPU版本的方法。(3) 利用并且评估了GPUDirect RDMA技术。实验结果证明，与初始的GPU版本相比，“scatter”阶段加速比为6.1倍，当MPI传输数据大于3KB时，通信过程提速了2.8倍。这些研究证明了针对模拟应用和GPU集群的特点进行特殊的优化能对性能带来显著的提升。

关键词：激光等离子体相互作用，Particle-In-Cell，CUDA程序优化，GPUDirect RDMA，CUDA

1. 问题简介

近20年来，生成超短激光脉冲能力的不断增强也极大的促进了激光等离子体相互作用领域的理论和实验研究[1][2]。激光驱动的电子加速是其中最重要的研究方向之一[3]。

Particle-In-Cell (PIC) 方法已经被广泛应用在激光等离子体相互作用的模拟和其他物理学模拟中[4]。这种模拟方法结果的质量依赖于系统中引入大量的粒子，所以基于PIC方法的应用会包含大量的计算。由于可能存在的大量的数据冲突和不规则内存访问，如何很好的将应用并行化并且利用好集群的特性来加速程序是一个很有挑战性的任务。

我们的研究用到的代码“2DPIC”是一个新研发的电磁学的,基于Particle-In-Cell方法的激光等离子体相互作用模拟的代码[5]。在这个代码中，应用了方向划分方法来求解麦克斯韦方程组。不同于有限差分时间域的方法，这个方法不受限于Courant条件：$\Delta t=\Delta x/c<\Delta y/c$，其中$\Delta t$, Δx, 和Δy分别是时间，纵向和横向分辨率[6]。

在将程序移植到GPU上的时候，我们利用“数组”作为了存储网格和粒子数据的主要的数据结构。为了处理“scatter”阶段中存在的数据冲突，在最开始我们尝试利用了传统的原子操作的方法。然而我们发现大量的原子操作成为了程序性能的瓶颈，所以我们提出了动态冗余算法和混合精度算法来加速这部分计算过程。

另外值得注意的一点是当程序运行在安装了很多GPU的集群中时，MPI通信部分的运行时间会显著增加。在这种情况下，利用GPUDirect RDMA技术可以将数据传输过程加速30%-60%，取决于要传输到其它MPI区域的数据量的多少。

论文剩余部分的安排：我们将在第二章介绍相关的研究工作，在第三部分我们会简单说明Particle-In-Cell算法。在第四部分和第五部分，我们阐述了移

National Key Research and Development Program of China under Grant No. 2016YFB0201400，2016YFB0201800, 国家重点研发计划(2016YFB0201400，2016YFB0201800); Japan JSPS RONPAKU (Thesis PhD) Program, 日本学术振兴会JSPS RONPAKU项目; NVIDIA GLOBAL CENTER OF EXCELLENCE, NVIDIA GPU全球卓越中心

植到GPU上的一些细节和我们采用的优化方法。优化的效果将在第六章讨论。最后在第七章对我们的工作做出了总结。

2. 相关工作

近些年，尽管一些基于PIC方法的代码已经被移植到了GPU平台，但是我们进行了一些不同的探索。

Viktor K. Decyk 和 Tajendra V. Singh将一个比较简单的PIC代码移植到了GPU，这个代码只应用了Poisson方程来计算出磁场信息[7]，但是我们用到的代码是基于电磁学，并且有更大的计算强度。另一方面，我们所用的粒子排序算法不同，在他们的实现中，粒子在每一步迭代后都会保持有序，而我们参数化了这个过程。对于实验方面，他们利用了一块Tesla C1060和GTX 280，我们在一个GPU集群中进行了实验评估。

Rejith George Joseph等人研究并且将PIC算法中计算强度最大的部分在一个GPU上进行了并行化[8]，但是我们将整个算法都移植到了GPU上。他们描述了一种新的并行的分层的木桶排序算法，其中粒子在每次循环迭代后逐渐变得有序。

G.Chen等人描述了一种基于CPU+GPU的一维PIC算法的移植实现[9]。这种方法的基本特点是，他们从场求解器中分离出了粒子轨迹的计算部分，于此同时也保持了系统自洽。

Xianglong Kong等人研究了一种线程竞争的算法来解决数据冲突的问题，但是这种算法仍然类似于传统的单精度的原子操作方法。并且他们只是在一块GPU上进行了测试和分析[10]。

Heiko Burau等人第一个研究了一种可扩展到集群中的针对等离子体物理模拟的PIC方法的实现[11] 。他们利用了LinkedList作为主要的数据结构，并且在4个GPU上做了测试。对于粒子排序，他们通过实验发现了在进行一定次数的循环之后对粒子进行重排序可以加速程序，但没有详细分析并且给出一个解决方案。

Bei Wang等人开发了基于PIC方法的GTC-P代码并且在一些超级计算机上进行了测试[12]。这个代码的算法和激光等离子体相互作用的模拟算法是不同的，他们所用的一些优化手段不适合应用在我们的模拟代码中。另一方面，他们用了传统的MPISendRecv的模式来在CPU和GPU之间交换数据，没有利用GPUDirect RDMA技术。

因此和以上存在的这些方法相比，我们采用的实现和优化方法有以下的不同：

1）在整体上把模拟算法移植到了GPU上，而不仅仅是其中计算强度高的部分。

2）探索了一系列方法来减少由于原子操作而产生的内存冲突带来的性能下降。同时我们也考虑到了尽量减少内存的使用这一因素。

3）研究了一种参数化的粒子排序算法，不需要每次迭代之后都进行粒子排序，从而在取得较好的性能的同时也不会浪费过多的GPU内存。

4）探索了在激光等离子体相互作用模拟中应用GPUDirect RDMA技术，而不是利用传统的MPISendRecv方法来在CPU和GPU之间进行数据传输。

3. 算法

这个代码同时完成对场信息求解的麦克斯韦方程组和对微粒子的运动求解的方程组的计算。对于没有发生碰撞的等离子体，在每一个时间步Δt内，相对论方程组：

$$\dot{p} = q_s \cdot (E + v \times B), \quad p = m_s \gamma v$$

$$\dot{r} = v, \quad \gamma = \sqrt{1 + (p / m_s c)^2}$$

会作用于每一个粒子。粒子会对电荷和电流密度生成贡献值。在当前网格上会求解以下麦克斯韦方程组：

$$\nabla \times E = -\partial_t B, \quad \nabla \times B = \frac{1}{\varepsilon_0 c^2} j + \frac{1}{c^2} \partial_t E,$$

$$\nabla \cdot B = 0, \quad \nabla \cdot E = \frac{1}{\varepsilon_0} \rho$$

这个过程会不断迭代进行，直到在等离子体场中达到自洽的状态。PIC循环如图1所示，主要包含了四个阶段:

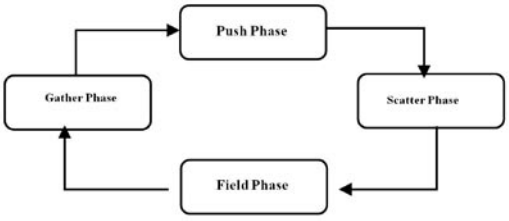

图1 PIC循环逻辑

1) Gather阶段：粒子的电场和磁场信息是由粒子在网格中的相对位置决定的网格点处得到的。每个粒子通过这些贡献值来生成当前位置的力。

2) Push阶段：上一个Gather阶段产生的力推动粒子到新的位置。

3) Scatter阶段：每一个粒子将自己的贡献传播到当前网格中。每一个网格点上的贡献值被累积起来加到局部密度中。

4) Field阶段 : 每一个网格点会从相邻的网格点来取得数据计算出新的电场和磁场值。

4. 基于GPU的并行化

4.1 合并函数

每一个PIC方法的循环都由图1描述的几个主要阶段组成。尽管每次迭代可以看作依赖于前一次迭代的结果（粒子新的位置），每一个粒子的gather，push和scatter这三个阶段可以看作是相互独立的，因此我们把这三个阶段合并为一个kernel，叫做“ParticleKernel”。通过这种方法，我们不仅简化了代码，而且更为重要的是利用了数据的局部性，这意味着对于每一个粒子，在上一个阶段产生的数据将马上被用在下一个阶段的计算中。伪代码如图2所示。

```
for all GPU blocks in parallel do
  for all threads in a block in parallel do
    TID = threadIdx + blockIdx * blockDim
    if(isValidLocation(TID))
      compute_force()
      move_particle()
      charge_deposition()
      if(this particle moves to another domain)
          store this particle to the global to_send
array
        isValidLocation(TID) = false
      Endif
    Endif
  Endfor
Endfor
```

图2 ParticleKernel伪代码

4.2 线程分配策略

我们为全局粒子数组中的每一个位置都分配了一个GPU线程。我们首先判断对应于这个粒子的位置是否“有效”。然后利用上一个阶段的输出作为下一个阶段的输入依次完成Gather，Push 和Scatter阶段。

4.3 数据结构转换

PIC方法主要基于两种类型的数据集合：一种是坐标连续的充电粒子相关的数据，另一种是网格相关的数据。对于每一种数据，我们都额外分配一个全局数组来存储。通过这种从AOS到SOA的转换，可以最大限度地减少数据量，无论是通过CudaMemcpy从CPU传到GPU的数据量，还是通过MPI通信在不同的区域中传输的数据量。

5. 对GPU版本的优化

5.1 “Scatter”阶段优化

如图3所示，“ParticleKerel”一共有三个步骤:计算贡献值,确定新的目标地址，和累加贡献值。

```
for all particles on the grid do
Calculate the contribution of this particle
Determine which cell the particle will move to
Add the contribution to the grid current
  Endfor
```

图3 ParticleKernel的三个步骤

如图4所示，每一个粒子都可能向9个方向移动（8个邻接位置和当前的位置）并且可能会对16个位置（图中的灰色区域）进行写操作。因此第三步“将贡献值加到当前网格”会带来大量的数据冲突，从而将严重的降低程序性能。为了解决这个问题，最简单的方法是利用传统的“原子操作”方法。然而由于CUDA现在并没有提供基于双精度的原子操作的底层实现，我们尝试应用了CAS(Compare & Set)来实现。但是经过分析程序，我们发现这个部分仍然占用了超过80%的时间，所以我们会首先分析原因然后依次介绍我们的优化方法。

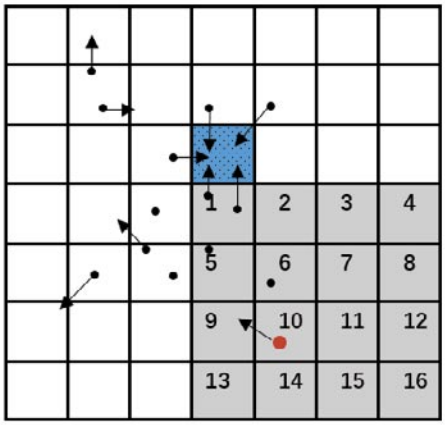

图4 粒子产生数据冲突

5.1.1 动态冗余算法

在尝试分析了产生数据冲突的根本原因之后，我们总结了两方面因素。一是同属一个网格的粒子可能尝试去写同一个内存地址；另一方面，属于不同的网格的粒子也可能产生数据写冲突。比较直接的想法是可以分配一块和原始的网格大小相同的内存区域来存储由于数据冲突而产生的临时数据。

更进一步，我们决定采用两个层次的“数据冗余”方法。一方面，如图5所示，我们为那些由于向不同的网格尝试写操作而产生数据冲突的粒子分配了冗余空间。因为一个粒子可能向16个方向进行写操作，所以我们对整个网格复制了16份，接下来考虑每一个粒子，原来对“原始”数组的写操作现在会转移到另外的冗余数组中，取决于数据会写入16方向中的哪一个网格。通过这种方法，原来由

于写入不同的网格而产生的数据写操作将会被分割开，从而大大减少了这种情况下的数据冲突数量。

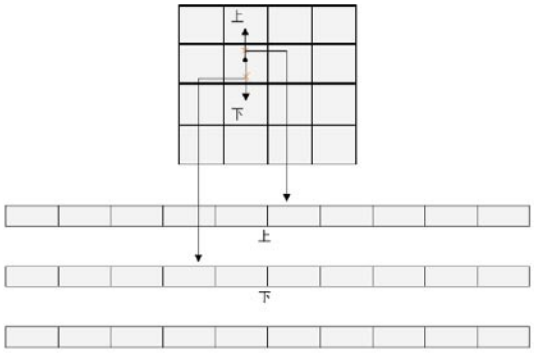

图5 解决属于不同cell的粒子产生的数据冲突

另一方面，那些属于同一个网格中的粒子也可能产生数据冲突，如图6所示，我们对这一个网格进行了冗余化，将当前网格的粒子按照它们的位置进行了划分。具体来说，我们首先将一个网格划分为多份，然后对每一个粒子计算并判断它们属于哪一个子区域。之后对当前网格信息的更新操作会在冗余数组上进行。通过这种方法，我们分割了属于同一个网格可能产生的数据写操作。为了控制在这种情况下为每一个网格分配多少冗余空间，我们引入了一个参数duplication_num。通过仔细的选择这个参数合适的值，可以找到在减少数据冲突和管理数据的开销之间的平衡点。

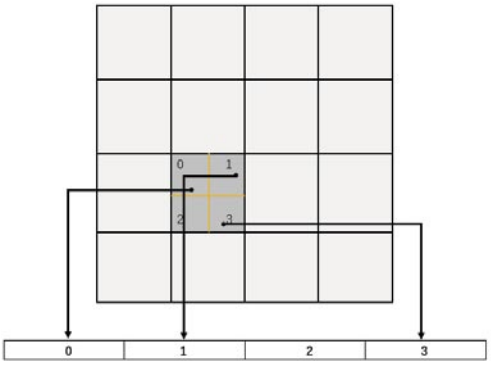

图6 解决属于同一个cell的粒子产生的数据冲突

像上文提到的，当所有的粒子的写操作完成时，我们会从所有的冗余数组中收集这些临时数据，并在另一个Kernel函数中将值更新到原始数组中。然而我们可能会疑惑的一点是这种方法很有可能会占用太多的内存资源。为了解决这个问题，我们引入了“动态冗余”的思想，伪代码如图7中所示。这个算法的基本思想是考虑到粒子在整个网格上其实不是均匀分布的，我们可以把粒子的分布情况考虑在内。不是简单的为每一个网格都分配相同数目的冗余空间，而是分配给那些包含粒子数目多的网格更多的冗余空间，相反，给那些包含粒子较少的网格较少的冗余空间。具体视硬件资源的情况而定。这个方法可以极大的减少内存空间的使用，尤其是当粒子分布很不均匀的情况下。当程序运行在28个GPU的集群中时，与初始的GPU版本相比，我们取得了约1.7倍的加速比。

Calculate the average number of particles each cell has: A

Calculate the max number of particles in a cell: B

For the cells that contains particles fewer than A,

make Base_duplication_num duplications of that cell, for

those contains more particles than A, make Base_duplication_num

* B / A duplications of the cells.

图7 动态冗余算法

现在我们可以从整体上来分析一下“动态冗余算法”。从图8可以看出，我们从两个维度减少了数据冲突:一个是从各个方向生成冗余空间，另一个是为每一个网格生成复制。“动态”意味着每一个子网格中的冗余空间的数量是不同的，取决于粒子在整个网格上的分布的“密度”。

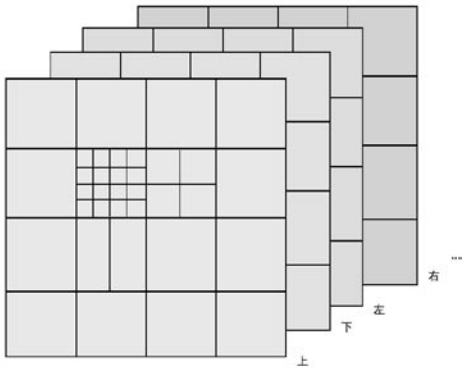

图8 在两个维度上通过冗余解决数据冲突

5.1.2 混合精度算法

另外一个来优化“scatter”阶段的方法是加速原子操作。尽管我们已经用传统的CAS(Compare & Set)方法来实现了双精度的原子操作，但是这种方法仍然会比CUDA运行时环境提供的单精度原子方法慢的多。为了能拥有单精度运算的快速和双精度运算的精确，我们将二者结合了起来。

具体来说就是在ParticleKernel之前和之后利用了另外两个GPU Kernel来做单精度和双精度之间的转换。代码的其余部分，如数据的初始化，Field阶段，和诊断阶段等，我们仍然用双精度来进行计算，尽量减少由于精度损失带来的影响。我们应用这个方法在结果精度可以接受的情况下取得了约2倍的加速比。

5.2 粒子排序算法

粒子可能逃逸出整个网格区域或者移动到相邻的MPI域中。在这两种情况中，全局数组中存储这个粒子的位置都会变成“空”或者“无效”的位置。如果我们不去处理，那么这些位置就会变的越来越多。这样不仅会浪费内存资源，也会浪费GPU计算资源，因为我们这里采用的GPU线程分配策略是为每一个全局粒子数组的位置分配一个GPU线程。进行粒子排序的另一个目的是使得在物理位置上邻近的粒子在内存中也能连续存储。尽管排序粒子能带给我们性能上的好处，但是同时排序也会耗费一定的时间和计算资源，在有些情况下，过于频繁的进行粒子排序并不能有效提升程序性能。

图9 粒子排序

为了在排序带来的好处和产生的额外开销之间取得平衡，我们引入另外一个可调节的参数，empty_ratio，这个参数表示了当前粒子数组的“空余程度”。如果数组中空余位置数量和数组的总的位置数量的比值大于我们设置的empty_ratio,那么粒子就会根据它们在整个网格中的位置进行排序，如图9所示。通过调节这个参数值，我们可以回收“无效”的空间，从而在得到较好的程序性能的同时，也能最大程度的节省内存空间。

5.3 CUDA-Aware MPI 和 GPUDirect RDMA

MPI能很好的和CUDA编程模型相兼容[13]。对于一般的MPI实现来说，只有指向host空间的指针可以被当作参数传递给MPI函数。然而如果把MPI和CUDA相结合，我们就可以发送GPU的缓存，而不仅仅是CPU端的数据了。如果不利用CUDA-Aware的MPI实现，我们就需要利用cudaMemcpy函数来把GPU内存中的数据传输到CPU端的内存。然而如果利用了这个技术，GPU端的缓存可以直接被MPI传输到另一端。从图10和图11可以看出使用了CUDA-Aware的MPI实现后，代码会变得更简洁，编程也更加容易。

```
//MPI rank 0
CudaMemcpy(s_buf_h,s_buf_d,size,cudaMemcpyDeviceToHost);
MPI_Send(s_buf_h,size,MPI_CHAR,1,100,MPI_COMM_WORLD);
//MPI rank 1
MPI_Recv(r_buf_h,size,MPI_CHAR,0,100,MPI_COMM_WORLD,&status);
CudaMemcpy(r_buf_d,r_buf_h,size,cudaMemcpyHostToDevice);
```

图10 传统的MPI send 和 receive模式

```
//MPI rank 0
MPI_Send(s_buf_d,MPI_CHAR,1,100,MPI_COMM_WORLD);
//MPI rank 1
MPI_Recv(r_buf_d,size,MPI_CHAR,0,100,MPI_COMM_WORLD,&status);
```

图11 利用了CUDA-Aware 的MPI 实现

除了更高的可用性之外，CUDA-Aware MPI还有哪些优点呢？它不仅使得MPI+CUDA更容易使用，而且也使得应用程序取得更好的运行效率，这基于以下两点原因[14]:

1）所有进行数据传输的操作都会被流水化

2）MPI库可以透明化的使用一些如GPUDirect的加速技术

远程DMA(RDMA)技术是在CUDA5.0中引入的GPUDirect技术中重要的一部分[15]，它连通了GPU和第三方的应用了PCI-E标准的硬件。GPU缓冲区中的数据可以不通过CPU端而直接被送到网卡进行传输，从而消除了CPU到GPU和其它PCI-E设备的内存带宽占用。这样也就显著增大了GPU和其他节点的MPISendRecv的效率。在一个由许多GPU加速节点组成的集群中，我们尝试应用了这个技术来加速数据传输过程。通过实验发现，当传输的数据量比较大时会有比较好的加速效果。随着将来越来越多的加速卡和节点的投入使用，更好的利用这项技术来减少数据传输的消耗将会变得越来越重要。

6. 实验结果和分析

表1所示的表格展示了我们的实验环境。在这个模拟中，我们设置了$\Delta x=\lambda/200$，$\Delta y=\lambda/100$和 $\Delta t=\Delta x/c$,在每一个非真空的网格中，对于电子，氘核和氚核，我们初始分别分配了64，32和32个粒子。这个模拟从$t=-\lambda/c$开始，与此同时激光脉冲从左边界$x=-\lambda$处发射。

表1 实验的软硬件环境

CPU	Intel(R) Xeon(R) CPU E5-2670 @ 2.60GHz
内存	62GB DDR3
GPU	NVIDIA Kepler K20
编译器	NVCC 7.0 + Open MPI 1.10 + GCC 4.9.1

从图12可以看出，总体来说，如果我们把使用单个GPU的初始的CUDA版本作为对比，在使用这些优化方法后，我们与初始版本相比取得了更好的可扩展性。我们下面分析每一种优化方法。

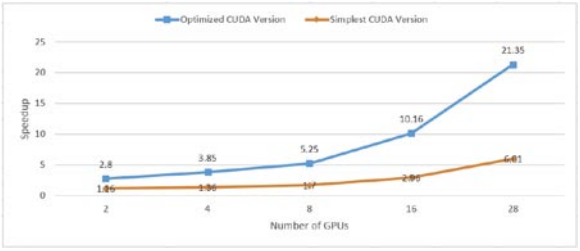

图12 利用了所有的优化方法之后的加速比(以使用单个GPU的初始的CUDA版本作为基线)

6.1 动态冗余算法

我们测试了利用了动态冗余算法之后与初始的GPU版本使用相同数量GPU相比的加速情况。从图13可以看出，当我们用两个GPU时候加速比是2.18.随着使用的GPU数量的增多，加速比会缓慢下降。当使用28块GPU时，加速比为1.69。

我们这里不考虑MPI通信时间随着GPU数量增多而变化带来的影响，因为使用冗余优化方法并不会改变通过MPI传输的数据的数量。因此由于总的粒子数目是固定的，随着使用GPU数的增加，每个GPU上的粒子数目会减少，这样数据冲突的数量就会降低，冗余优化的效果不突出，反而从冗余空间收集数据和更新数组的额外开销变得明显，导致了加速比会缓慢下降。

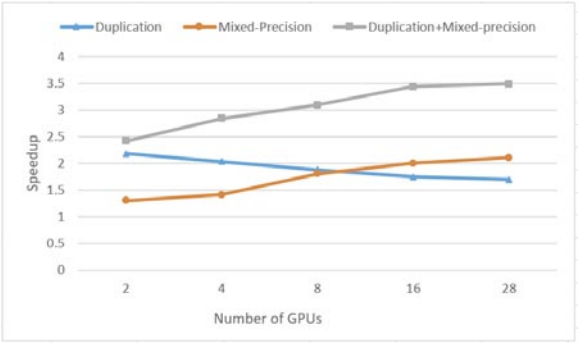

图13 使用各个优化方法之后的加速比
(使用相同数量GPU)

6.2 混合精度算法

总体上来说，随着GPU数目的增多，混合精度计算方法带来的加速收益会变得越来越明显，当GPU数量超过8后，加速效果会超过动态冗余算法。这是因为虽然总的计算量是固定的，但是当GPU数目较少时，每个GPU上的计算量就会较多，这样进行数据精度转换带来的额外开销会比较大，当每个GPU上的粒子数变少后，由于各个GPU上精度转换函数是并行执行的，精度转换的总开销也会相应降低，加速效果就会变得越来越明显。

6.3 RDMA

对于GPUDirect RDMA，我们通过改变MPI线程的数目来改变需要传输的数据量。我们发现使用GPUDirect RDMA可以显著减少数据传输的时间。如图14所示，当传输的数据量大于3KB时，我们可以获得2.8倍的加速比。但是当传输的数据量为0.9KB时，加速比只有1.37。这是因为当需要传输的数据量比较少时，RDMA的优势不明显。待传输的数据量越大，越多的数据要传输到另一个MPI域中。对于传统的MPISendRecv方法来说就需要更多的时间来将数据从CPU端传输到GPU端。

我们这里需要指出的一点是，对于非RDMA版本，MPI通信时间的计算不仅仅包括纯粹的数据传输时间，还包括收集数据，将数据放到数组中和把接

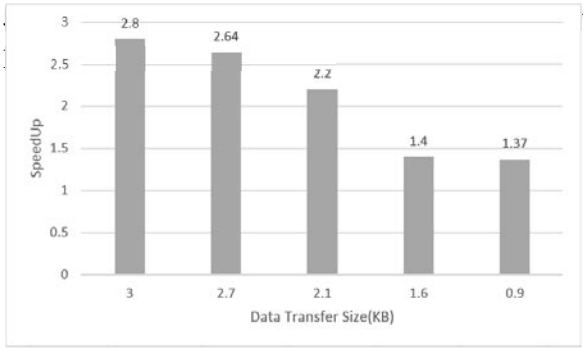

图14 MPI通信时间的加速比

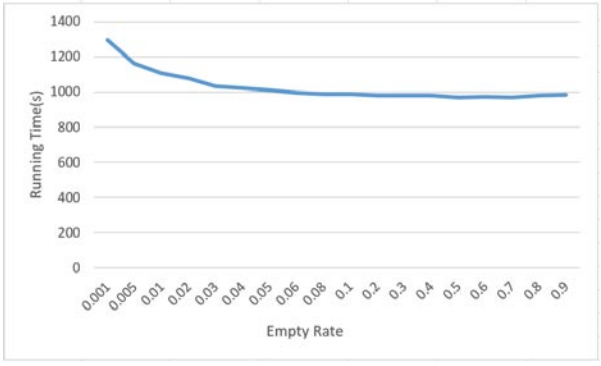

图15 不同的空余率对程序性能的影响

6.4 粒子排序

我们的粒子排序算法的中心思想是引入了一个参数empty_ratio，目的是在性能达到或者接近最优的情况下找到这个参数最小的值。我们用八个MPI进程来测试这个参数不同的值的情况下的程序性能，结果如图15所示。实验发现当empty_ratio小于0.05的时

候，性能受这个参数的影响很大。然而当大于0.05的时候，性能接近于稳定。由于这个参数越小代表着内存空间的使用越少，所以我们可以通过设置这个参数为0.05来达到性能和内存使用之间的平衡。

7. 结论

我们研究了一种把整体的基于PIC方法的激光等离子体相互作用模拟的代码移植到GPU端的可行的方法并且对比原始的CPU代码取得了可观的加速比。基于这个初始的GPU版本，我们介绍了一系列的优化方法来加速基于原子操作的包含大量数据冲突的scatter阶段，包括动态冗余算法，混合精度计算方法和一种参数化的粒子排序方法。我们也尝试利用并且评估了GPUDirect RDMA方法在集群中加速MPI通信时间的效果。我们发现当数据传输的量大于一定的阈值的情况下，这种方法能显著减少MPI通信时间。我们相信这些优化方法也能应用于其他基于PIC方法的物理学模拟代码，并且也对激光等离子体相互作用的研究发展有着非常重要的意义。

致谢

该论文受国家重点研发计划(2016YFB0201400，2016YFB0201800)资助。林新华特别致谢日本学术振兴会JSPS的RONPAKU项目资助。感谢NVIDIA GCOE的支持。感谢上海交通大学物理与天文系的翁苏明教授的指导和帮助。

参考文献：

[1] G. Mourou, T. Tajima, and S. Bulanov. Optics in the relativistic regime[J]. Rev.Mod.Phys:vol.78, no. 2, pp.309 - 371, 2006.

[2] A. Korzhimanov, A. Gonoskov, E. Khazanov, et al. Horizons of petawatt laser technology[J]. Phys - Usp+: vol. 54, no. 1, pp. 9 - 28, 2011.

[3] I. Kostyukov and A. Pukhov. Plasma - based methods for electron acceleration: current status and prospects[J]. Phys - Usp+: vol. 58, no. 1, pp. 81 - 88, 2015.

[4] Wen Minhua, Yu Zhanpeng, Simon See, et al. A NVIDIA Kepler Based Acceleration of PIC Method[J]. Procedia Engineering: vol. 61, p. 398 - 401, 2013.

[5] S. M. Weng, M. Murakami, H. Azechi, et al. Quasi - monoenergetic ion generation by hole - boring radiation pressure acceleration in inhomogeneous plasmas using tailored laser pulses[J]. Physics of Plasmas:vol. 21, p. 012705, 2014.

[6] V. K. Decyk and T. V. Singh. Adaptable particle - in - cell algorithms for graphical processing units[J]. Comput.Phys Comm:vol. 182, p. 641, 2011.

[7] L. R. Pfund, R.E.W., and M. ter Vehn J. Lpic++ a parallel one - dimensional relativistic electromagnetic particle - in - cell code for simulating laser - plasma - interaction[J]. MPQ:vol. 225, 1997.

[8] R. Joseph, G. Ravunnikutty, S. Ranka, et al. Efficient gpu implementation for particle in cell algorithm[C]. 2011 IEEE International, Intl. Parallel and Distributed Symposium (IPDPS), 2011.

[9] G. Chen, L. Chacon, and D. Barnes. An efficient mixed - precision, hybrid cpu - gpu implementation of a non - linearly implicit one - dimensional particle - in - cell algorithm[J]. J.Comput. Phys: vol. 231, p. 5374, 2012.

[10] X. Kong, M. C. Huang, C. Ren, et al. Particle - in - cell simulations with charge - conserving current deposition on graphical processing units[J]. J.Comput. Phys:vol. 230, p. 1676, 2011.

[11] H. Burau, R. Widera, W. Honig, et al. Picongpu: a fully relativistic particle - in - cell code for a gpu cluster[C]. IEEE Trans. Plasma Sci:vol. 38, p. 1676, 2010.

[12] K. Madduri, E. - J. Im, K. Ibrahim, et al. Gyrokinetic particle - in - cell optimization on emerging multi - and manycore platforms[J]. Parallel Comput:vol. 37, p. 501, 2011.

[13]J. Kraus. An introduction to cuda - aware - mpi[OL]. 2013.Available:https://devblogs.nvidia.com/parallelforall/introductioncuda - aware - mpi/

[14] D. Rossetti. Benchmarking gpu direct rdma on modern server platforms[OL]. 2014. Available : https://devblogs.nvidia.com/paralle - for - all/benchmarking - gpudirect - rdma - on - modern - server - platforms/

[15] N. Corporation. Developing a linux kernel module using gpudirect rdma[OL]. 2015. Available : http://docs.nvidia.com/cuda/gpudirect - rdma