

密级: _____



中国科学院大学
University of Chinese Academy of Sciences

博士学位论文

强流束流动力学研究与软件开发

作者姓名: _____ 刘志聪

指导教师: _____ 秦庆 研究员

_____ 中国科学院高能物理研究所

学位类别: _____ 理学博士

学科专业: _____ 粒子物理与原子核物理

培养单位: _____ 中国科学院高能物理研究所

2018 年 06 月

Study of High Current Beam Dynamics
and
the Development of Simulation Code

by
Zhicong Liu

A thesis submitted to
The University of Chinese Academy of Sciences
in partial fulfillment of the requirements
for the degree of
Doctor of Nuclear and Particle Physics

Institute of High Energy Physics, Chinese Academy of Sciences

June, 2018

学位论文独创性声明

本人郑重声明：我所呈交的学位论文是本人在导师指导下进行的研究工作及所取得的研究成果。尽我所知，除了文中已经标注引用的内容外，本论文中不含其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中作了明确的说明或致谢。本人知道本声明的法律结果由自己承担。

作者签名：_____ 日期：_____

关于学位论文使用授权的说明

本人完全了解中国科学院高能物理研究所有关保留、使用学位论文的规定，即：中国科学院高能物理研究所有权保留送交论文的复印件，允许论文被查阅和借阅；可以公布论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存论文。

(保密的论文在解密后应遵守此规定)

作者签名：_____ 导师签名：_____ 日期：_____

摘 要

强流束流动力学主要问题是空间电荷效应问题。空间电荷效应会导致束流和粒子的共振和不稳定性，束流的共振和不稳定的一个显著结果就是束晕，从而产生束损。为了探寻束晕产生背后的物理机制，我们使用了非自治的模型对束晕产生的机制进行了分析，得到了一个粗略的物理图像。由于空间电荷效应问题是一个多体耦合问题，我们无法得到解析解，为了更加深入的了解问题背后的机制，我们需要一个软件进行精确的数值模拟。

束流模拟软件在加速器建设中起到了必不可少的作用。随着束流流强的增大，需要模拟的粒子数目增加了几个量级，需要的功能也越来越多。面对新的物理问题，我们需要探索新的算法，优化新的程序结构，编写出一个新的粒子模拟软件，了解并掌握软件背后的计算过程。粒子云网格（Particle-In-Cell）算法被广泛应用于加速器的研究和设计中。基于此，作者在前人的基础上使用C++开发了束流模拟软件P-TOPO。并且对ADS进行了end-to-end模拟，模拟结果和TraceWin进行对照相符很好。作者还编写CPU和GPU两个版本，实现并行计算，来提高模拟软件的效率。

除此之外，作者还编写了基于GPU的求解空间电荷效应的无网格保辛算法。这种算法模型并不利用网格，而是利用高阶分解来求解空间电荷效应。相比PIC算法，这种方法能够显著的降低由于网格数值噪声带来的发射度增长，但是计算量要大得多。然而由于无网格算法很适合并行，有很好的可扩展性，这种算法非常适合使用GPU进行加速计算。通过使用CUDA和GPU，可以显著加快无网格粒子跟踪代码的运行速度。

关键词： 空间电荷效应， PIC， Symplectic， GPU， 模拟软件

Abstract

A main problem of high current beam dynamics is space charge effect, which would lead to the resonance and instability of beam and particles. A direct result from resonance is beam halo and lose. In order to explore the physical mechanism behind the halo formation, we used a non-self-consistent model to analyze the propagation of beam and obtained a rough physical image. Since the problem of space charge effect is a multi-body coupling problem, we can not get the analytic solution. In order to understand the mechanism behind the problem, we need a software to carry on the accurate numerical simulation.

Beam simulation code plays an indispensable role in accelerator construction. As the beam current increases, the number of particles to be simulated increases by several orders of magnitude and more features are required. Faced with new physical problems, we need to explore new algorithms, optimize new program structure, and make a new particle simulation software. It also help us understanding the algorithms behind the calculation process. Particle-In-Cell (PIC) algorithm is widely used in accelerator research and design. Based on PIC algorithm, the author developed a beam simulation software P-TOPO using C++ on the basis of predecessors. An end-to-end simulations of ADS was done, and the simulation results were in good agreement with TraceWin. The author also prepared code in CPU and GPU versions to achieve parallel computing and improve the efficiency of simulation software.

In addition, the author also made a GPU-based the space charge effect solver based on gridless symplectic algorithm. This model uses a gridless spectral method instead of mesh point to calculate the space charge effect. It can effectively reduce the emittance growth associated with numerical grid heating compared with the PIC algorithm. However, this model is much slower compared with PIC method. Fortunately, it is very suitable for parallelism and can achieve very good speeded and scalability, especially by using GPU and CUDA library

Keywords: Space Charge Effect, PIC, Symplectic, GPU, Simulation Code

目 录

摘要	vii
Abstract	ix
目录	xi
图形列表	xiii
表格列表	xv
符号列表	xvii
第一章 引言	1
1.1 课题研究的背景和意义	1
1.2 课题研究的现状	2
1.3 本论文主要工作	4
1.4 论文主要创新之处	4
第二章 强流束流动力学	5
2.1 基本理论	5
2.2 束流横向运动	5
2.3 束流纵向运动	5
2.4 空间电荷效应	5
2.5 国内外强流加速器简介	5
2.6 小结	5
2.7 GPU和CPU架构	5
2.8 束核模型	6
2.9 PIC算法	10
第三章 P-TOPO程序结构与ADS注入器I模拟	21
3.1 程序结构	21
3.2 正确性校验	23

3.3 C-ADS注入器I模拟	25
3.4 小结	30
第四章 PIC算法在GPU上的实现	33
4.1 GPU程序结构	33
4.2 正确性校验	47
4.3 性能	48
4.4 CPU集群	54
4.5 小节	58
第五章 Symplectic算法在GPU上的实现	61
5.1 Symplectic算法	62
5.2 GPU程序结构	64
5.3 性能	68
5.4 模拟	71
5.5 小结	73
第六章 总结	75
附录 A 粒子模拟程序用户界面	77
参考文献	79
作者简历	83
攻读学位期间发表的学术论文及科研成果	85
致 谢	87

图形列表

2.1 GPU与CPU的结构对比	6
2.2 束核包络与单粒子横向位移	8
2.3 粒子的庞加莱截面	8
2.4 不同相位压缩因子下的庞加莱截面	9
2.5 PIC算法块循环	11
2.6 PIC算法分段流程	11
2.7 分配方式示意图	13
2.8 形状因子方程	14
2.9 使用FFT求解泊松方程	16
3.1 P-TOPO程序结构	22
3.2 同一均方根尺寸下的不同种类束团分布	23
3.3 P-TOPO与理论值的点电荷电势对比	24
3.4 零流强和15mA时，P-TOPO结果与包络方程的对比	25
3.5 C-ADS注入器I结构示意图	25
3.6 RFQ中的横向发射度和纵向发射度	27
3.7 RFQ中束团横向均方根尺寸	28
3.8 RFQ中束团纵向均方根尺寸和能散	28
3.9 超导段横向发射度和纵向发射度	29
3.10 超导段束团横纵向尺寸以及能散	30
4.1 权重差值中的线程冲突	34
4.2 网格分块	35
4.3 GPU上的PIC算法分段流程	35
4.4 粒子排序示意图	37
4.5 权重插值示意图	39
4.6 单GPU域分解模式解泊松方程各部分时间占比	43
4.7 64*64*64格点下，跨节点多GPU域分解模式解泊松方程时间随GPU个数的变化	45

4.8	128*128*128格点下, 跨节点多GPU域分解模式解泊松方程时间随GPU个数的变化.....	46
4.9	正确性校验	48
4.10	PIC程序在单GPU上的加速比	50
4.11	程序各部分耗时在不同粒子数时所占百分比	51
4.12	不同的格点数情况下单GPU求解泊松方程的加速比	52
4.13	64*64*64个格点, 160k个粒子时, 复制模式程序耗时随GPU个数的变化.....	52
4.14	64*64*64个格点, 160k个粒子时, 域分解模式程序耗时随GPU个数的变化	53
4.15	64*64*64个格点, 1.6m个粒子时, 程序耗时随GPU个数的变化	54
4.16	64*64*64个格点, 16m个粒子时, 程序耗时随GPU个数的变化	54
4.17	1.6m粒子数下, 不同混合并行配置的耗时与内存占用	55
4.18	160k粒子数下, 不同混合并行配置的耗时与内存占用	56
4.19	16m粒子数下, 不同混合并行配置的耗时与内存占用	56
4.20	PIC程序使用多个CPU节点的耗时	57
4.21	使用64个节点时程序各个部分消耗时间所占的百分比	57
4.22	不同并行配置下的多节点运行情况比较	58
5.1	单GPU加速比	69
5.2	多GPU加速比	71
5.3	不同流强下的发射度增长.....	72
5.4	三阶共振附近的庞加莱截面	73

表格列表

1.1 束流模拟软件	3
3.1 C-ADS注入器I基本参数	26
4.1 单GPU域分解模式解泊松方程	43
4.2 单节点双GPU域分解模式解泊松方程	44
4.3 64*64*64格点下，跨节点多GPU 域分解模式解泊松方程各部分所用时间	45
4.4 128*128*128格点下，跨节点多GPU域分解模式解泊松方程各部分所用时间	46
4.5 PIC程序在单GPU上的加速比	49

符号列表

Characters

Symbol	Description	Unit
T	temperature	K
t	time	s
V	velocity vector	$\text{m} \cdot \text{s}^{-1}$
v_x	x component of velocity	$\text{m} \cdot \text{s}^{-1}$
v_y	y component of velocity	$\text{m} \cdot \text{s}^{-1}$
v_z	z component of velocity	$\text{m} \cdot \text{s}^{-1}$
r	position vector	m

Operators

Symbol	Description
Δ	difference
∇	gradient operator

Abbreviations

Acronym	Description
LHC	Large Hadron Collider
RHIC	Relativistic Heavy Ion Collider
BEPCII	Beijing Electron–Positron Collider II
ALS	Advanced Light Source
ADS	Accelerator Driven Sub-critical system
CSNS	China Spallation Neutron Source
PIC	Particle In Cell
P-TOPO	Paralleled Trace-Of-Particle-Orbit
NGP	Nearest Grid Point

CIC	Cloud In Cell
TSC	Triangular Shaped Cloud
h.o.t.	High Order Term
FD	Focusing Defocusing
FODO	Focusing Drift Defocusing Drift
GPU	Graphics Processing Unit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FFT	Fast Fourier Transformation
BLAS	Basic Linear Algebra Subprograms

第一章 引言

1.1 课题研究的背景和意义

粒子加速器（particle accelerator）是使带电粒子在高真空中受磁场力控制、电场力加速而达到高能量的装置。作为为基础科学提供条件的工具，粒子加速器在理解各类原子核结构、原子核内部的相互作用、天体物理、宇宙学、生命科学、材料科学、核医学等基础学科都有重要的意义。粒子加速器最开始的用途是高能物理研究，在高能物理和粒子物理研究中粒子加速器作为对撞机使用，来研究粒子的微观结构和发现新的物理现象，比如CERN的LHC（Large Hadron Collider），日本KEK的SuperKEKB，美国BNL的RHIC（Relativistic Heavy Ion Collider），中国的北京正负电子对撞机（Beijing Electron–Positron Collider II，BEPCII）。

除了高能物理，加速器在其他方面也有着许多应用。粒子加速器可以用作同步辐射光源，比如美国ALS（Advanced Light Source），上海同步辐射光源，以及北京在建的新光源，同步辐射在物理，材料，化学，生物，医学等方面发挥着越来越重要的作用。加速器还在医疗方面有很多应用，比如质子治疗是目前最有效的治疗癌症的方法。除此之外加速器还在能源问题研究起到重要作用，比如加速器驱动次临界洁净核能系统（Accelerator Driven Sub-critical System，ADS），是目前产生核能、增殖核材料和嬗变核废物的解决方案之一。所有的这些需求，都在推动着加速器往更高流强发展。

粒子加速器的种类繁多，它们的特点各有不同，可以按不同的原则加以分类。按照粒子运动的轨道形状分，可以分为环形加速器和直线加速器。按照加速粒子的种类划分，可以分为电子加速器，离子加速器，其中强流离子加速器为本课题主要的研究范围。由于加速器相关的大部分实验事例与加速器束流流强正相关，所以强流加速器成为了未来的加速器科学重要的发展方向之一。强流加速器在核物理、生命科学、材料科学等基础科学研究，核医学、放射医学等应用研究方面有着越来越重要的作用。在最近二十年来，得益于高能物理、核物理和包括散裂中子源、加速器驱动的次临界核能系统以及其他应用领域的强烈需求，强流粒子加速器也得到了较大的发展[1, 2]，随着加速器朝着高流强的方向发展，其面临着新的问题和挑战。

强流束流动力学研究的对象加速器中是大量带电粒子在自生库伦场和外界电磁场中的演化问题。其重要的特点是离子束流受空间电荷效应的影响很大，束流的集体不稳定性较为突出，导致束流的整体损失或束流品质变坏。空间电荷效应

是指束团内部粒子之间的直接库仑作用，其作用的来源为束团本身，束团中的粒子带相同的电荷，而相互排斥，粒子会收到束团的斥力，导致带电束团的自然发散和粒子的相位改变。而集体不稳定性是束团与其运行的环境有关，来源于外部的电磁元件，比如与弯转磁铁，聚焦磁铁，加速腔等的相互作用。空间电荷效应和束流的集体不稳定性都可能导致束流的整体损失或品质变坏。

空间电荷效应和束流流强，束流能量，以及束团的大小有关。在现有聚焦强度下，流强越强，空间电荷效应带来的问题越严重。在束流能量低，束流流强大的时候，空间电荷效应非常明显，甚至在内部作用和外部作用中占主导地位。特别是对束流损失的严格控制和束流功率提升所需要的功率源的问题。更高的束流功率意味着更加严格的束流损失率控制，即使很小的束流损失也将造成机器维护无法进行。如国际上流行采用平均损失率为 1 W/m 的手动维护标准，对应于停机4小时后距离束流管 0.3 m 处的剩余剂量率为 100 mrem/hr 。这个标准最早是针对 1 GeV 量级的束流能量的，但对较低的束流能量也基本上适合。譬如，对于能量为 1 GeV 、平均流强为 1 mA 的质子束，允许的束流损失率为 1 nA/m 或 $10^{-6}/\text{m}$ 。这是非常严格的要求，需要从物理设计和技术措施上保证其实现。

因此，强流束流动力学研究的主要问题和难点是如何自治的表述束流自身产生的非线性空间电荷效应。空间电荷效应导致的束流和粒子的共振和不稳定性，束晕，发射度交换、束流丢失等问题也正成为了强流束流物理中关注的焦点。如何定性，定量理解强流束流动力学中的由于非线性空间电荷效应导致的束流集体不稳定性、束晕粒子形成机制也成为了近年来加速器物理研究中的重要课题。在束流模拟软件中，随着束流流强的增大，需要模拟的粒子数目上升了若干量级，需要的功能也越来越多。面对新的需求和新的物理问题，我们需要编写一个新的粒子模拟软件，了解并掌握软件背后的计算过程。而且，由于需要模拟的粒子数很大，粒子模拟程序的效率和功能是得到精确物理图像的关键，我们必须针对粒子模拟进行并行优化。

1.2 课题研究的现状

由于多体非线性耦合系统的复杂性，强流束流物理问题的研究可以从较为粗糙的解析近似和精确的数值模拟两个角度出发。其中解析方法基本是在哈密顿力学的框架内完成，加上一些必要的近似，建立起一些物理模型，比如束流包络的稳定性问题，单粒子运动以及共振问题等。针对束晕、束流集体效应等问题，目前国际上流行的做法是使用非自治的模型，比如束核模型，对束晕产生的机制进行分析，给出基本的物理图像。数值模拟方法主要是使用束流模拟软件在计算机

名称	开源	图形界面	并行	数据处理	匹配	速度	精确度	是否收费
Parmila	否	否	否	是	否	慢	低	免费
Impact	否	否	是	否	否	较快	较高	免费
TraceWin	否	是	是	是	是	较快	较高	收费
Dynac	是	否	否	是	否	快	较低	免费
Orbit	否	否	是	否	否	较快	较高	免费
Track	否	否	否	否	否	较快	较高	免费

表 1.1: 束流模拟软件

上进行大规模计算，对带电粒子进行跟踪和模拟，其基本模型有传输矩阵映射，PIC（Particle In Cell）数值模拟等。国际上普遍的做法是使用PIC方法进行粒子跟踪，得到粒子演化的图像，分析其背后的物理机制。

解析的方法在某些特定的情况下可以得到准确的结果，但是更多的还是需要近似，与真实的加速器模型相差较远。解析方法可以定性的给出一些束流性质的预测和加速器设计的准则，但是无法给出某个加速器下的具体结论。而数值方法依赖于具体加速器的设计，可以给出相比解析方法更为准确的结果，在加速器建设中起到了必不可少的作用。

目前在国际加速器界存在着一些束流模拟软件 [3–9]，如表1.1所示。但随着实际加速器的流强上升，我们的需求也越来越多。此外，国外软件大部分都不是开源，不方便我们根据自己的需求进行改进。国际上针对提升程序的效率提升方面也做出了很多探索，比如MPI，OpenMP等等CPU程序并行化，另外，GPU并行计算也是模拟软件的发展方向和研究热点。

国内近年来设计或在建的强流加速器项目有C-ADS，中国散裂中子源（CSNS，China Spallation Neutron Source）等，依靠于这些项目，国内加速器同行业对强流加速器研究也在进行中。但是国内对束流模拟软件的探索和开发较少，目前还没有一套比较成熟完备的加速器束流模拟软件。加速器界同仁大部分依然使用国外的相关软件，并且对软件内部的计算过程并不十分了解。

总之，随着加速器束流流强的增大，需要模拟的粒子数目增加了几个量级，需要的功能也越来越多。面对新的物理问题，我们需要探索新的算法，优化新的程序结构，编写出一个新的粒子模拟软件，了解并掌握软件背后的计算过程。

1.3 本论文主要工作

本论文的工作主要分三个方面，首先是探索求解空间电荷效应的算法；其次是多粒子追踪模拟软件的开发，我们将其命名为P-TOPO（Paralleled Trace-Of-Particle-Orbit）；最后是使用开发的软件P-TOPO进行束流动力学研究。

本文中，第二章将会对强流束流动力学的基本理论进行简要介绍，其中包括束流的横向运动与纵向运动、空间电荷效应的基本理论等，并对国内外的强流加速器进行简要介绍。第三章为计算机程序的物理模型及程序设计实现，主要介绍在包括束核模型在内的一些物理模型，并对束流模拟程序中常用的PIC算法，和一种新型的保辛算法进行介绍。第四章介绍模拟程序的设计以及算法的实现，包括程序总体的设计，以及PIC算法在GPU上和CPU集群上的实现，保辛算法在单GPU和GPU集群上的实现，并分别对每一种算法实现进行了正确性校验。第五章介绍模拟程序的性能和结果，利用模拟程序对空间电荷效应和束流集体效应进行研究，并对C-ADS注入器I进行模拟研究。

1.4 论文主要创新之处

本文对不同的物理模型进行了比较，并开发了束流模拟软件P-TOPO，而目前国内还没有成熟的加速器束流模拟软件，本课题的有关工作将填补国内的空白。而且相对于其他束流模拟软件，P-TOPO在很多地方进行了创新。比如大多数模拟软件使用飞行时间法获得加速腔的相位，而P-TOPO使用类似于实际运行中的扫相方法获得加速腔相位，虽然花费时间要更长，但是结果更加精确；再比如我们在不同的子程序中使用了不同的并行策略，更加灵活的分配计算负载，提高了并行的效率。

除此之外，本文还在不同的空间电荷算法上进行了探索，并且在GPU上做了实现。PIC算法和保辛算法在GPU上的实现均有创新，其中保辛算法在GPU上的实现更国际首次。在PIC算法在GPU集群的实现中，我们比较了不同的并行策略下求解泊松方程的效率，并且实现并测试了新的节点间通讯方法；而在保辛算法的实现中，我们针对GPU架构进行了海量优化，尤其是针对GPU的内存读写规则，重新组织了程序，使其数据吞吐速度更高。而且，我们还测试了保辛程序每一部分的加速比和瓶颈，为下一步升级做好了准备。

最后，本文进行了空间电荷的研究，对束晕产生的机制进行了新的探索。在对C-ADS注入器I的模拟中，验证了其设计的合理性。

第二章 强流束流动力学

2.1 基本理论

2.2 束流横向运动

2.3 束流纵向运动

2.4 空间电荷效应

2.5 国内外强流加速器简介

2.6 小结

2.7 GPU和CPU架构

GPU英文全称Graphic Processing Unit，中文为“图形处理器”。GPU一开始是由于在现代的计算机中图形的处理变得越来越重要，需要一个专门的图形的核心处理器，使显卡减少对CPU的依赖，并进行部分原本CPU的工作；CPU是英文全称为Central Processing Unit，中文为“中央处理器”，一般由逻辑运算单元、控制单元和存储单元组成。近年来，GPU高速发展，极大的提高了计算机图形处理的速度和质量，不但促进了图像处理、虚拟现实、计算机仿真等相关应用领域的快速发展，同时也为人们利用GPU进行图形处理以外的通用计算提供了良好的运行平台。

相对于CPU，GPU在并行处理和计算密集型问题方面具有很大优势。目前，GPU已成为普通计算机强大、高效的计算资源。如图2.1所示，一个CPU一般只拥有少数几个核，而一个GPU会有几百甚至几千个核。从系统架构上看，GPU针对向量计算进行了高度并行的数据流优化处理，这种多核架构特别适合进行数据并行。这种以数据流作为处理单元的机制，在对数据流的处理上可以取得很高的速度。GPU 加速计算是指同时利用GPU 和 CPU，加快应用程序的运行速度[10]。GPU厂商为科学计算设计了自己的API，其中CUDA（Compute Unified Device Architecture）就是NVIDIA设计的并行计算平台和编程规范[11]。利用CUDA，我们能够有效的利用GPU，使程序的运行效率大大提高。

根据使用CUDA的测试结果显示，利用GPU实现FFT（Fast Fourier Transformation）、BLAS（Basic Linear Algebra Subprograms）、排序及线性方程组求解等科学计算，与单纯依靠CPU实现的算法相比，平均性能提高了近20倍。

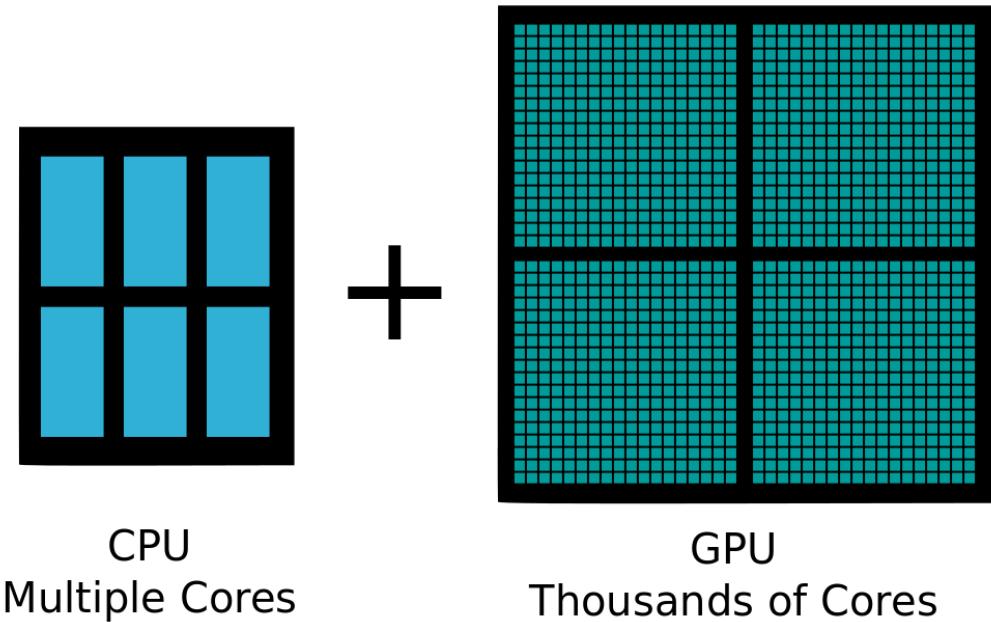


图 2.1: GPU与CPU的结构对比

随着GPU可编程能力、并行处理能力和应用范围方面得到不断提升和扩展，GPU已成为当前计算机系统中性能很高的部件。因此，我们需要充分利用现有计算资源，发挥GPU的高性能计算能力，使程序在GPU与CPU之间进行协作，以提高模拟程序的运行效率。

2.8 束核模型

加速器是加速带电粒子束流的机器。我们面临的主要问题也是在束流中，其中空间电荷效应是其中非常重要的一种。针对空间电荷问题，我们主要有两种描述手段，分别是使用解析的方法和数值的方法，其中解析的方法为求解傅拉梭夫方程（Vlasov Equation），如式2.1。

$$\begin{aligned}
 f(x, y, z, p_x, p_y, p_z) &= f(H) \\
 \frac{df}{dt} &= 0 \\
 \Delta U &= -\frac{\rho}{\epsilon_0}
 \end{aligned} \tag{2.1}$$

但是多体耦合问题是没有解析解的，我们必须采用一些近似，建立一个更简单的模型，以得到这个问题的大致图像。T. P. Wangler在1998年提出了了一种束核模型^{??}以描述空间电荷导致的束晕形成机制。在束核模型中，连续的束流通过一个轴对称的聚焦通道，假设粒子束核为均匀分布，则其包络大小可由包络方程描述。此时一个任意的粒子会受到束核产生的空间电荷力，当粒子处于束核内部时，空间电荷力为线性的；当粒子处于束核外部时，空间电荷力为非线性的。忽略单粒子对束核的影响，则束核的半径可以由式2.2表示。

$$\frac{d^2R}{dz^2} + k_0^2 R - \frac{\epsilon^2}{R^3} - \frac{K}{R} = 0 \quad (2.2)$$

其中

$$K = \frac{qI}{2\pi\epsilon mc^3\beta^3\gamma^3} \quad (2.3)$$

式中 R 为束核包络的大小， q ， m ，和 βc 分别为粒子的电荷，质量，和纵向速度， k_0 为外部聚焦强度， γ 为洛伦兹因子， I 为流强， ϵ 为发射度。

而单粒子的横向运动方程如式2.4：

$$\frac{d^2X}{dz^2} + k_0^2 X - F_{sc} = 0 \quad (2.4)$$

其中 X 为粒子的横向位置， F_{sc} 为由均匀分布束核产生的的空间电荷力，可以表示为：

$$F_{sc} = \begin{cases} KX/R^2, & |X| < R \\ K/X, & |X| \geq R \end{cases} \quad (2.5)$$

对式2.2和式2.4进行无量纲化，我们引入以下变量 $r = R/R_0$ ， $x = X/R_0$ ， $\tau = k_0 z$ ，并将空间电荷导致的相位压缩因子表示为 $\eta = k/k_0 = \sqrt{1+u^2} - u$ 。则无量纲化的束核包络方程可以表示为：

$$\frac{d^2r}{d\tau^2} + r - \frac{\eta^2}{r^3} - \frac{1-\eta^2}{r} = 0 \quad (2.6)$$

无量纲化的单粒子运动方程可以表示为：

$$\frac{d^2x}{d\tau^2} + x - (1-\eta^2) \times \begin{cases} x/r^2, & |x| < r \\ 1/x, & |x| \geq r \end{cases} \quad (2.7)$$

对式2.6和式2.7 仅含有相位压缩因子 η 一个参数。束核的匹配解为 $r = 1$ ，为了描述束核的初始匹配程度，我们将束核的初始的包络半径定位为失配度 $\mu = r_{initial}$ 。

失配度会影响束核包络的变化。当束核完全与聚焦强度完全匹配时，即 $\mu = 1$ 时，束核的包络大小是不变的；而当包络与聚焦强度不匹配的时候，包络半径会周期性变化，如图2.2的蓝色曲线为 $\eta = 0.5$, $\mu = 0.6$ 时的束核半径变化情况。而粒子和束核的相互作用会驱使粒子横向位置发生变化，以表示空间电荷对粒子的影响，如图2.2的红色曲线为粒子初始 $x = 0.8$, $x' = 0$ 时的粒子横向轨迹。

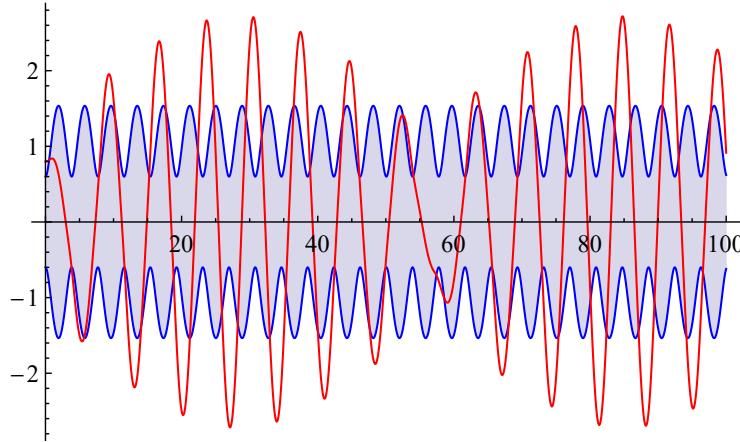


图 2.2: 束核包络与单粒子横向位移

可以看出，束核的半径呈现规律的周期震荡。而当在每个周期的开始处对粒子的横向位置和动量进行观测时，我们可以得到一张粒子的庞加莱截面（Poincare surface of section），如图2.3所示，在粒子的庞加莱截面中可以看到明显的规律，粒子沿着某一条轨道进行运动。

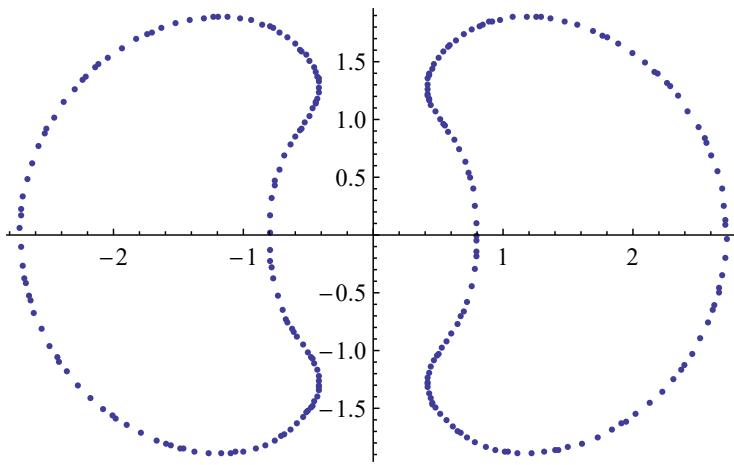


图 2.3: 粒子的庞加莱截面

对粒子从中心到边界均匀排布，我们可以得到一组庞加莱截面，以显示不同初始位置处粒子的行为，如图2.4所示。四副图分别为不同的相位压缩因子 η 下的庞加莱截面，不同的颜色代表不同初始位置的粒子的轨迹。

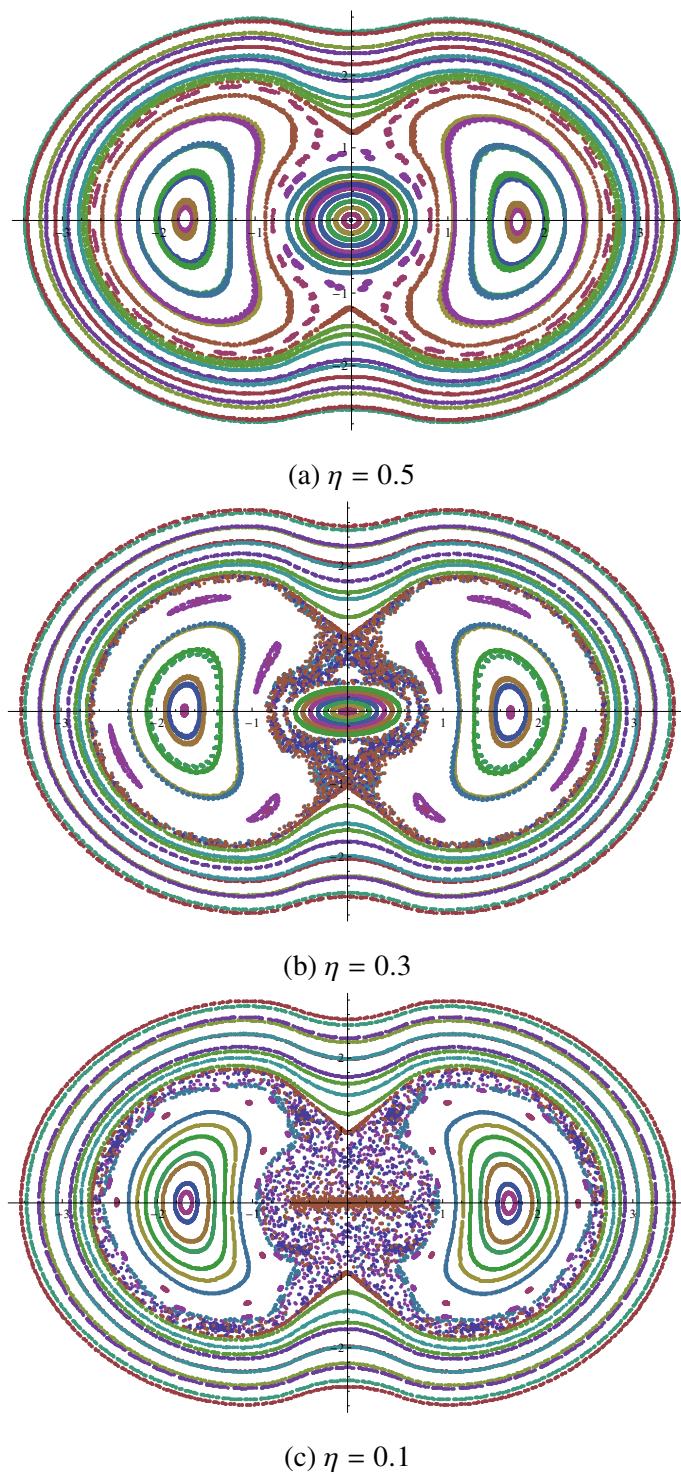


图 2.4: 不同相位压缩因子下的庞加莱截面

庞加莱截面图可以分为三个区域：一，内椭圆区，大小约为束核半径及周边一小部分；二，不动点区，或者叫共振区，包括x轴上的不动点以及周围的环线，显示了参数共振的轨迹；三，外围类椭圆轨迹。在空间电荷效应较弱的情况下，即 η 较大时，三个区的粒子都是做有规律的运动如图2.4a所示。但是当空间电荷主导的时候，比如 $\eta = 0.1$ 和 $\eta = 0.3$ 的时候（图2.4b和图2.4c），在内椭圆区会出现混沌行为，并且随着相位压缩因子变小，混沌行为变得更加明显，参数区的粒子轨迹也开始对初始条件敏感。

由庞加莱截面可以对束晕形成的机制进行分析，束晕中的粒子主要来源于共振区中的粒子，这些粒子最开始分布于内椭圆区与共振区的分界线附近，随着粒子运动，其向外移动到共振区与外椭圆区的边界附近，在两个边界间震荡，形成了束晕。在空间电荷效应主导的情况下，内椭圆区的粒子也有可能形成束晕，如图2.4c所示，内椭圆区也出现了混沌行为，最内部的粒子也会移动到外部形成束晕。

束核模型在一定程度上可以解释束晕形成的机制，但是这个模型并不是自治的，与实际不相符。束核模型假设束核是稳定的，并且外部聚焦力是恒定的常数；而实际的束核有可能是不稳定的，外部的聚焦力也是可能变化的。因此，我们需要一个更加自治的系统来研究束流的行为。

2.9 PIC算法

PIC算法使用自治的系统对粒子进行模拟。PIC算法的基本流程如图2.5和2.6所示。

首先，根据粒子的位置和范围确定所用的空间网格大小，然后执行以下循环：

1. 将网格内的所有粒子的电荷根据粒子的位置分配到网格格点上，即从粒子分布得到获得空间点上的密度分布。
2. 根据网格点上的电荷密度分布，求解网格点上的泊松方程，得到网格点上的电势分布。
3. 根据电势分布，差分得到网格点上的电场分布。
4. 根据粒子位置和网格上的电场分布，反向权重得到静止坐标系下粒子所受到的电场，再通过洛伦兹变换得到实验室坐标系下粒子所处位置的电磁场。
5. 通过牛顿方程，推动粒子，更新粒子位置和动量。

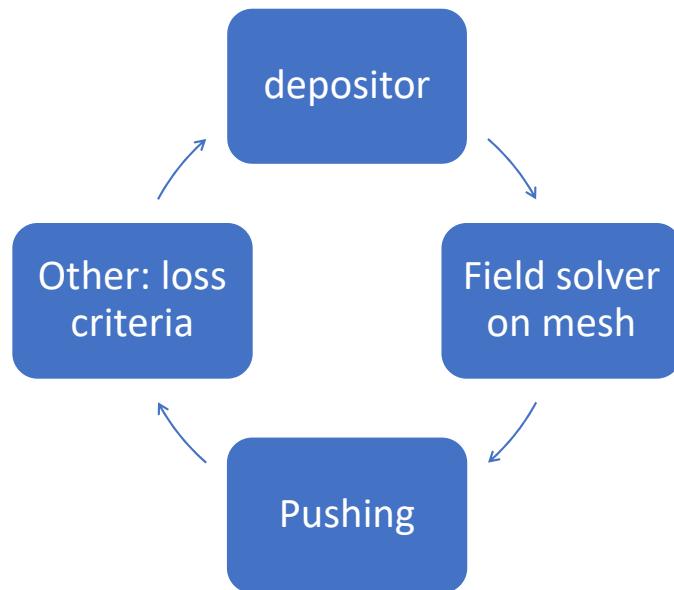


图 2.5: PIC 算法块循环

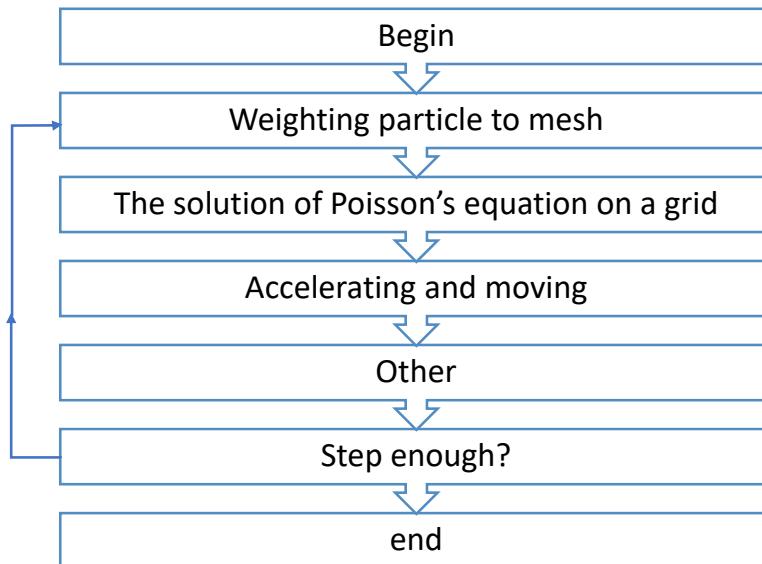


图 2.6: PIC 算法分段流程

2.9.1 权重差值

PIC 算法的第一步是将粒子电荷权重差值到网格上。粒子权重差值主要有以下几个方面考虑：

1. 如何将粒子电荷密度（3D）或者电流密度（2D）权重到空间网格上。
2. 求解泊松方程后，如何将网格上的电场权重差值回粒子上。
3. 如何确定网格点的数目和相对于粒子分布的位置及范围。

其中，对于前两个问题，我们最好采用相同的差值算法的正反两个过程，来处理从粒子到网格和从网格到粒子的差值，从而避免算法上带来的非物理效应。如果采用不同的插值方法，可能会出现粒子自己推动自己的错误。

以一维问题为例，将粒子电荷权重到空间网格上的过程，可以理解为粒子坐标 x_i 到网格点上的电荷密度 ρ_p 的一个映射，其中 p 为网格点的坐标。一些常见的权重方法有最近网格点（Nearest Grid Point, NGP），网格云差分法（Cloud In Cell, CIC），三角云分配法（Triangular Shaped Cloud, TSC），如图4.1所示。图中，粒子处于 x 位置，而 $p - 1, p, p + 1$ 代表不同的网格格点，不同颜色的面积大小代表分配给相应格点的电荷百分比。按照不同的形状分配可以得到不同的结果，NGP方法为零阶差值，只将电荷分配到离粒子位置最近的网格点上，有较大误差。CIC方法为一阶差值，根据粒子位置和格点位置的关系，如图2.7b所示，将粒子按不同比例分配到临近的两个网格点上。TSC方法为二阶差值，如图2.7c所示，分配函数为三角形，将粒子按不同比例分配到临近的三个网格点上。

在数学上，这个分配过程可以表示为：

$$\rho_p = \sum_i^{N_p} q_i W(x_i - x_p) \quad (2.8)$$

其中， q_p 为粒子电荷， $W(x)$ 为形状因子。如图2.8所示，根据插值的阶数不同， $W(x)$ 有很多形式。下式给出了零阶，一阶，和而阶的形状因子：

$$S_0(x) = \begin{cases} 1, & |x| \leq \frac{\Delta x}{2} \\ 0, & |x| > \frac{\Delta x}{2} \end{cases} \quad (2.9)$$

$$S_1(x) = \begin{cases} 1 - \frac{|x|}{\Delta x}, & |x| \leq \Delta x \\ 0, & |x| > \Delta x \end{cases} \quad (2.10)$$

$$S_2(x) = \begin{cases} \frac{1}{\Delta x} \left[\frac{3}{4} - \left(\frac{|x|}{\Delta x} \right)^2 \right], & |x| \leq \frac{\Delta x}{2} \\ \frac{1}{2\Delta x} \left[\frac{3}{2} - \frac{|x|}{\Delta x} \right], & \frac{\Delta x}{2} < |x| \leq \frac{3\Delta x}{2} \\ 0, & |x| > \frac{3\Delta x}{2} \end{cases} \quad (2.11)$$

在数值模拟中，低阶的算法，比如NGP，会带来很大的粒子离散误差。为了避免数值误差，我们需要采取高阶精度的算法，但是高阶的算法会导致计算量的增大，不满足计算的要求，所以实际中往往采用CIC差值方法。在P-TOP0中，所有的差值形式都采用一阶的线性差值（CIC）。

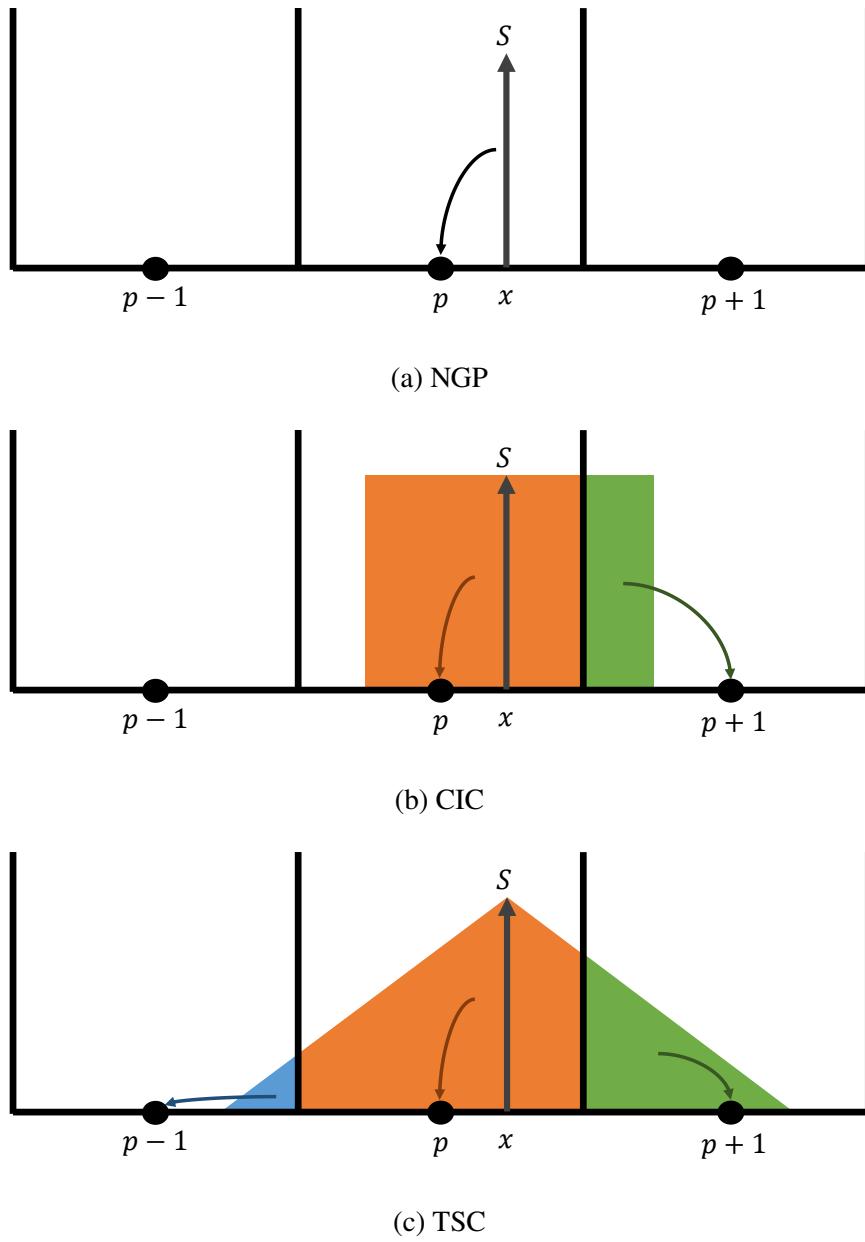


图 2.7: 分配方式示意图

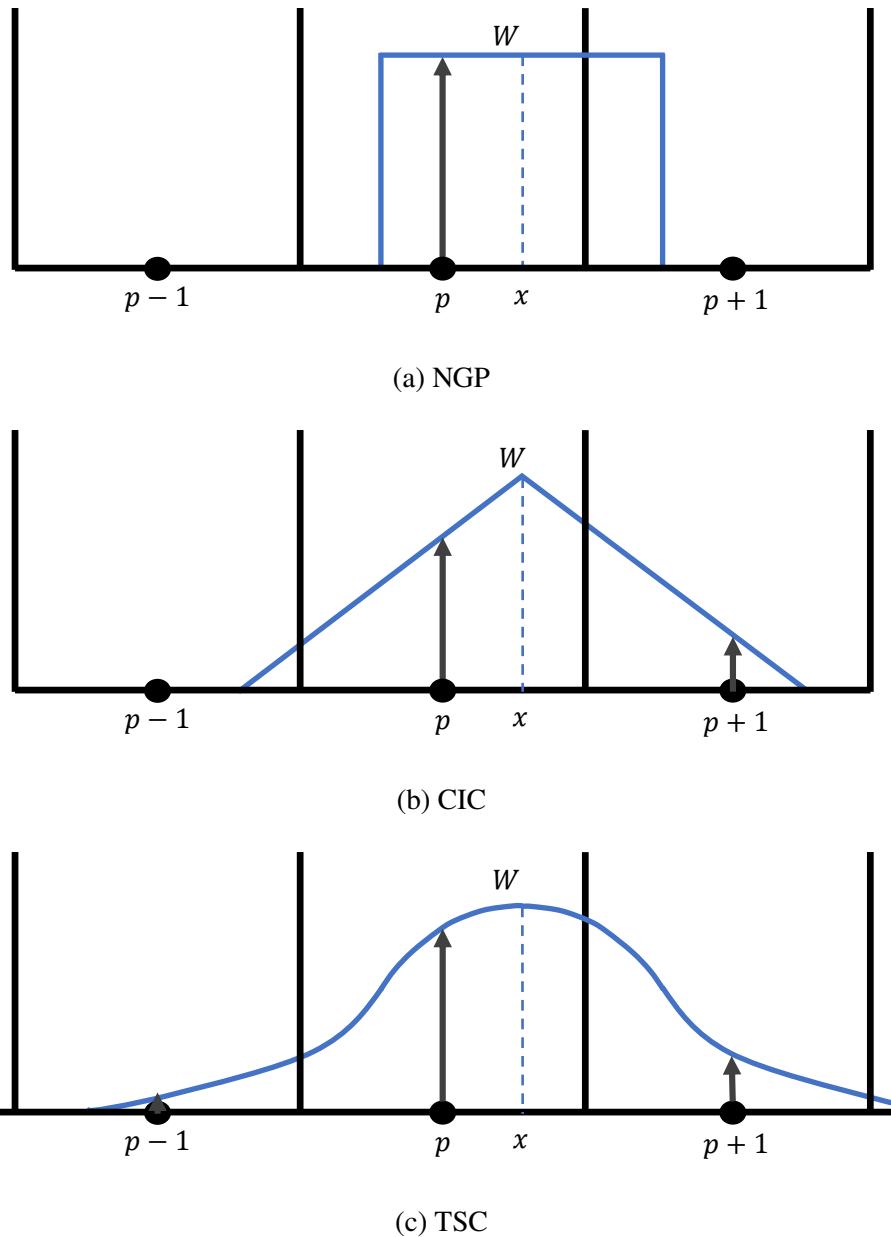


图 2.8: 形状因子方程

对于实际的三维问题，处理方法也是类似，我们采用立方网格，使用体积极权重法来计算网格上的电荷。

2.9.2 使用FFT解泊松方程

经过由粒子到空间网格的权重插值后，我们得到了分布在网格点上的电荷密度分布函数。接下来要做的就是求解网格点上的泊松方程，对于本程序而言，我们使用快速傅里叶变换（FFT）的方法来求解。以一维问题为例，泊松方程：

$$\nabla^2 \phi = -\frac{\rho}{\epsilon_0} \quad (2.12)$$

在网格上可以离散表示为：

$$\frac{\phi_{j-1} - 2\phi_j + \phi_{j+1}}{\Delta_x^2} = -\frac{\rho_j}{\epsilon_0} \quad (2.13)$$

其中， ϕ_j 为第 j 个格点上的电势， ρ_j 为第 j 个格点上的电荷量， Δ_x 为两个格点之间的距离。

将 ϕ_j 和 ρ_j 进行傅里叶展开，我么可以得到：

$$\phi_j = \frac{1}{N_x} \sum_{n=0}^{N_x} \phi_n \exp\left(-\frac{i2\pi nj}{N_x}\right) \quad (2.14)$$

$$\rho_j = \frac{1}{N_x} \sum_{n=0}^{N_x} \rho_n \exp\left(-\frac{i2\pi nj}{N_x}\right) \quad (2.15)$$

其中， N_x 为一个方向上网格点的数目。

将式2.14和式2.15代回式2.13可得：

$$\begin{aligned} \sum_{n=0}^{N_x} \phi_n & \left(\exp\left(-\frac{i2\pi n(j-1)}{N_x}\right) - 2 \exp\left(-\frac{i2\pi nj}{N_x}\right) + \exp\left(-\frac{i2\pi n(j+1)}{N_x}\right) \right) \\ & = \frac{\Delta_x^2}{\epsilon_0} \sum_{n=0}^{N_x} \rho_n \exp\left(-\frac{i2\pi nj}{N_x}\right) \end{aligned} \quad (2.16)$$

对于傅里叶空间的的电势和电荷密度，我们可以得到 ϕ_n 和 ρ_n 的关系：

$$\phi_n \left(\exp\left(\frac{i2\pi n}{N_x}\right) + \exp\left(-\frac{i2\pi n}{N_x}\right) - 2 \right) = \frac{\Delta_x^2}{\epsilon_0} \rho_n \quad (2.17)$$

化简得到

$$\phi_n = \frac{\rho_n}{\epsilon_0 K_n^2} \quad (2.18)$$

其中

$$K_n = \left(\frac{2\pi n}{N_x \Delta_x} \right) \frac{\sin(\pi n / N_x)}{\pi n / N_x} \quad (2.19)$$

可以看出，通过傅里叶变换得到的傅里叶空间中的电荷密度 ρ_n ，我们可以得到傅里叶空间中的电势分布函数 ϕ_n 。然后再经过傅里叶逆变换，我们就可以得到空间中的电势分布 ϕ_j ，最后经过差分我们就可以得到空间电场分布函数。如图2.9所示：

$$\rho_j \xrightarrow{FFT} \rho_n \xrightarrow{\text{Multiply by } \frac{1}{K_n^2}} \phi_n \xrightarrow{IFFT} \phi_j \xrightarrow{\text{Finite Difference}} E_{x,j}$$

图 2.9: 使用FFT求解泊松方程

需要注意的是，这种 \exp 指数的傅里叶变换方式决定了我们使用的是周期性边界条件。如果我们想要第一类边界条件，我们需要采用 \sin 函数变换，其变换过程与FFT类似。在本程序中，我们在横向采用第一类边界条件，用来模拟边界处的电势为0；而在纵向采用周期性边界条件，用以模拟连续束流。

2.9.3 使用蛙跳法推动粒子

我们使用经典的牛顿方程来推动粒子：

$$\frac{d\mathbf{x}}{dt} = \mathbf{v} \quad (2.20)$$

$$m \frac{d\mathbf{v}}{dt} = \mathbf{F} \quad (2.21)$$

粒子推动的方法必须满足以下五个条件：

2.9.3.1 一致性

一致性是指差分方程和微分方程的一致。任何数值算法都是使用有限大小的步长来模拟一个连续的过程。因此，我们应该在时间步长无线小的时候，使其效果与连续推动一致。

例如，我们采用有限步长 ΔT 来推动粒子位置：

$$\frac{\mathbf{x}_{n+1} - \mathbf{x}_n}{\Delta T} = \mathbf{v}_n \quad (2.22)$$

$$m \frac{\mathbf{v}_{n+1} - \mathbf{v}_n}{\Delta T} = \mathbf{F}_n \quad (2.23)$$

当其步长 ΔT 足够小时，其结果应该与微分方程2.20 和2.21相一致。

2.9.3.2 同时性

同时性又叫做可逆性。微分方程2.20和2.21是可逆的。例如，一个粒子在给定的力场中沿时间正向积分运动，然后如果将其速度取反，并沿时间反向积分，粒子将会沿着正向的轨迹向回运动，并能回到起始点。

这一点在差分近似中并不能得到保证，比如，差分方程2.22和2.23中，粒子在第 $n+1$ 步处取反，粒子位置变化为

$$\frac{\mathbf{x}_n - \mathbf{x}_{n+1}}{\Delta T} = -\mathbf{v}_{n+1} \quad (2.24)$$

其速度为 \mathbf{v}_{n+1} 而不是 \mathbf{v}_n ，也就是说粒子并不能沿其轨迹回到初始位置。这是因为差分方程的左右的时间中心不相同，式2.22左侧，位置改变为 $\mathbf{x}_{n+1} - \mathbf{x}_n$ ，是以 $t_{n+1/2}$ 为时间中心的；而右侧 \mathbf{v}_n 是时间 t_n 时的速度。因此，我们需要将其改变为：

$$\frac{\mathbf{x}_{n+1} - \mathbf{x}_n}{\Delta T} = \mathbf{v}_{n+1/2} \quad (2.25)$$

$$m \frac{\mathbf{v}_{n+1/2} - \mathbf{v}_{n-1/2}}{\Delta T} = \mathbf{F}_n \quad (2.26)$$

这也就是蛙跳法，即位置的时间与速度的时间并不在统一时刻，而是采用相差半个步长，然后交替推动粒子的速度和位置，以满足同时性条件。

2.9.3.3 准确性

准确性与一步计算中的数值解与解析解的误差有关。这个误差主要来源于两方面：第一，来自于计算机的精度，例如，双精度浮点数仅有十四或十五位有效数字；第二，来自于使用有限小的步长代表连续变量的截断误差。通常来讲，第一种来自计算机数值精度的误差非常小，如果算法是稳定的（关于稳定性的论述见下一条），这个误差是可以忽略的。第二种来自于有限小的步长的误差，可以由解析解与数值解的差值来表示，其差值通常与步长 ΔT 的 n 次方 $(\Delta T)^n$ 的成正比，而不同算法之间的稳定性可由阶数 n 来表示。

由式2.25和2.26可知，蛙跳法为解析解2.20和2.21的二阶近似。证明如下：由式2.25和2.26可得：

$$\frac{\mathbf{x}_{n+1} - 2\mathbf{x}_n + \mathbf{x}_{n-1}}{(\Delta T)^2} = \frac{\mathbf{F}(\mathbf{x}_n)}{m} \quad (2.27)$$

如果 \mathbf{X} 为解析解：

$$\frac{d^2\mathbf{X}}{dt^2} = \frac{\mathbf{F}}{m} \quad (2.28)$$

那么有限小步长的误差可以由 δ^n 表示：

$$\frac{\mathbf{X}_{n+1} - 2\mathbf{X}_n + \mathbf{X}_{n-1}}{(\Delta T)^2} = \frac{\mathbf{F}(\mathbf{X}_n)}{m} - \delta^n \quad (2.29)$$

将 \mathbf{X}_{n+1} 和 \mathbf{X}_{n-1} 泰勒在 $\mathbf{X}_n = \mathbf{X}(t_n)$ 处展开，式2.29可以写作：

$$\frac{d^2\mathbf{X}}{dt^2} + \frac{(\Delta T)^2}{12} \frac{d^4\mathbf{X}}{dt^4} + h.o.t. = \frac{\mathbf{F}(\mathbf{X}_n)}{m} - \delta^n \quad (2.30)$$

其中 $h.o.t.$ 为高阶项 (high order term)。式2.29 与式2.30相减可得：

$$\delta^n = -\frac{(\Delta T)^2}{12} \frac{d^4\mathbf{X}}{dt^4} + h.o.t. \quad (2.31)$$

即，蛙跳法为二阶准确度 ($\delta^n \propto (\Delta T)^2$)。

2.9.3.4 稳定性

稳定性与误差随时间的变化有关。即使算法每一步的误差非常小，最终误差也有可能由于累积效应导致变大。在一个数值算法中，如果每一步的误差不会导致一个更大的累积误差，那么这个算法是稳定的。

下面，我们讨论蛙跳法的稳定性。根据式2.27，如果我们给定 $\mathbf{x}_0 = \mathbf{X}_0, \mathbf{x}_1 = \mathbf{X}_1$ 作为初始条件，不考虑误差的情况下，我们可以得到之后的一系列解 $\mathbf{X}_2, \mathbf{X}_3, \mathbf{X}_4, \dots$ ，其中：

$$\mathbf{X}_2 - 2\mathbf{X}_1 + \mathbf{X}_0 = \frac{\mathbf{F}(\mathbf{X}_1)}{m} (\Delta T)^2 \quad (2.32)$$

$$\mathbf{X}_3 - 2\mathbf{X}_2 + \mathbf{X}_1 = \frac{\mathbf{F}(\mathbf{X}_2)}{m} (\Delta T)^2 \quad (2.33)$$

然而，由于误差存在，我们无法得到微分方程中的精确解 $\mathbf{X}_2, \mathbf{X}_3, \mathbf{X}_4, \dots$ ，而是会得到一系列含有误差近似解 $\mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4, \dots$ ：

$$\mathbf{x}_2 - 2\mathbf{X}_1 + \mathbf{X}_0 = \frac{\mathbf{F}(\mathbf{X}_1)}{m} (\Delta T)^2 \quad (2.34)$$

$$\mathbf{x}_3 - 2\mathbf{x}_2 + \mathbf{X}_1 = \frac{\mathbf{F}(\mathbf{x}_2)}{m} (\Delta T)^2 \quad (2.35)$$

如此，在第二步处的误差为：

$$\epsilon_2 = \mathbf{x}_2 - \mathbf{X}_2 \quad (2.36)$$

在随后的每一步计算中，含有误差的近似结果都会被使用，我们需要知道的是 ϵ_2 是如何影响后续计算结果。由式2.34和式2.35可得：

$$(\mathbf{X}_3 + \epsilon_3) - 2(\mathbf{X}_2 + \epsilon_2) + \mathbf{X}_1 = \frac{\mathbf{F}(\mathbf{X}_2 + \epsilon_2)}{m} (\Delta T)^2 \quad (2.37)$$

与式2.33相比较，可得：

$$\epsilon_3 - 2\epsilon_2 = (\mathbf{F}(\mathbf{X}_2 + \epsilon_2) - \mathbf{F}(\mathbf{X}_2)) \frac{(\Delta T)^2}{m} \quad (2.38)$$

将上式右侧在 $\mathbf{x} = \mathbf{X}^2$ 处进行泰勒展开:

$$\epsilon_3 - 2\epsilon_2 \approx \epsilon_2 \left. \frac{\partial \mathbf{F}}{\partial \mathbf{x}} \right|_{\mathbf{x}=\mathbf{X}^2} \frac{(\Delta T)^2}{m} \quad (2.39)$$

类似可得:

$$\epsilon_4 - 2\epsilon_3 + \epsilon_2 = \epsilon_3 \left. \frac{\partial \mathbf{F}}{\partial \mathbf{x}} \right|_{\mathbf{x}=\mathbf{X}^3} \frac{(\Delta T)^2}{m} \quad (2.40)$$

则第n步的误差为:

$$\epsilon_{n+1} - 2\epsilon_n + \epsilon_{n-1} = \epsilon_n \left. \frac{\partial \mathbf{F}}{\partial \mathbf{x}} \right|_{\mathbf{x}=\mathbf{X}^n} \frac{(\Delta T)^2}{m} \quad (2.41)$$

上式并不能得到一个准确的误差递进关系,除非 $\partial \mathbf{F} / \partial \mathbf{x}$ 为常量。由于我们的目标是判断算法是否稳定,而不是算法有多稳定,所以我们考虑最坏的情况,使用 $-\|\partial \mathbf{F} / \partial \mathbf{x}\|_{max}$ 代替 $\partial \mathbf{F} / \partial \mathbf{x}$,其中的负号是因为我们假设所计算的问题是有限边界,而不是会延伸到无穷远。可得误差随时间的变化方程:

$$\epsilon_{n+1} - 2\epsilon_n + \epsilon_{n-1} = - \left. \frac{\partial \mathbf{F}}{\partial \mathbf{x}} \right|_{max} \frac{(\Delta T)^2}{m} \epsilon_n \quad (2.42)$$

$$\epsilon_{n+1} - 2\epsilon_n + \epsilon_{n-1} = -(\Omega \Delta T)^2 \epsilon_n \quad (2.43)$$

其解的形式与震荡方程形式类似 $\epsilon_n = \lambda_n = \exp(i\omega n \Delta T)$ 。两个特征解 λ_+ 和 λ_- 为:

$$\lambda_{\pm} = 1 - \frac{(\Omega \Delta T)^2}{2} \pm \left[\frac{(\Omega \Delta T)^2}{2} \right] \left[1 - \frac{4}{(\Omega \Delta T)^2} \right]^{1/2} \quad (2.44)$$

对于算法而言,想要保证数值上稳定,其误差不能随着时间而变大,即误差推动方程的特征解必须位于单位圆内,即 $|\lambda| \leq 1$ 。

因此,我们必须使时间步长足够小,才能保证算法稳定,对于蛙跳法而言,我们必须使:

$$\Omega \Delta T \leq 2 \quad (2.45)$$

第三章 P-TOPO程序结构与ADS注入器I模拟

粒子云网格(Particle-In-Cell, PIC)方法是研究空间电荷效应非常常用的一种数值模拟算法[12, 13]。PIC发展于上世纪七十年代，被广泛的应用于等离子体模拟和加速器设计。随着加速器的流强越来越高，束流模拟代码在束流动力学研究中的作用也越来越重要，尤其是在例如C-ADS, SNS, ESS等加速器中[14–16]，其非线性空间电荷效应占据主导地位。一个拥有更强大的计算能力以及更高的精确度的程序，是我们研究强流束流动力学所必须的。我们开发了一个新的粒子模拟程序，命名为Parallel-Trace of Particle Orbits (P-TOPO)，致力于研究强流加速器中的空间电荷效应问题[17–20]。

在本章，我们首先介绍P-TOPO代码情况，之后使用P-TOPO对C-ADS进行模拟研究。

3.1 程序结构

P-TOPO使用C++语言开发，并且使用OpenMP进行并行化，目标是在普通多核PC机上快速模拟粒子在加速器中的行为。P-TOPO使用时间t，而不是使用位置z，作为基本的独立变量，这是研究粒子的空间电荷效应非常自然的选择。即时使用位置z作为基本变量，在求解空间电荷力的过程中，也需要将粒子坐标转换为同一时间t下的时空间坐标。在P-TOPO中，产生外场的一些加速器元件，例如各种磁铁，螺线管，RFQ，的外场使用元件的解析模型得到；而对于另外一些元件，例如超导腔，我们首先读取场文件[21]，然后采用二阶插值的方法获取粒子所受到的场的大小和方向。对于束团内部的空间电荷效应，我们使用经典的PIC方法进行求解[22]。程序的结构如图3.1所示，图中每一个条目为一个类：

- 主类MAIN调用其他各个具体的类，比如从外部Lattice结构中获取外场，求解内部空间电荷力，以及根据粒子所受的场推动粒子的位置和动量。
- Lattice类根据外部输入文件，解析或者数值地构建加速器的结构，以及根据粒子坐标，给出粒子所受的外场力。
- Distribution类产生粒子的初始分布，目前可以产生 KV, Waterbag, Parabolic, 和Gaussian四种粒子分布，如图3.2所示。

- Beam类统计粒子的信息，并且计算出如rms大小，发射度等束团参数，将其保存以供进一步分析和输出。
- Internal Field类通过PIC方法计算空间电荷效应，在PIC方法中，我们使用FFT来求解Poisson方程。
- Leapfrog类和Runge-Kutta 4类都是继承Pusher类，是不同的推动粒子位置和动量的方法。

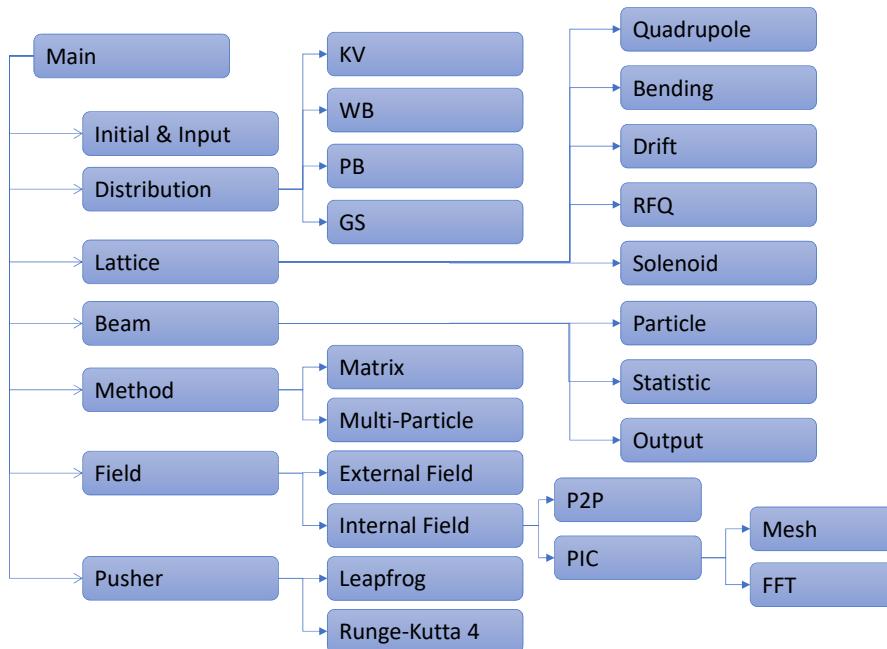


图 3.1: P-TOPO程序结构

程序主循环中的大部分都是并行化的，例如求解内场，推动粒子等。最基本的，程序采用粒子并行策略，即每个线程处理不同的粒子。以粒子推动部分为例，如果粒子总数为160000，线程数为8，那么每个线程推动20000个粒子。程序对可能发生线程冲突的部分，例如权重粒子，采用了其他的并行策略。除此之外，程序也使用了FFTW库的内部并行方法。

我们使用了一个4核的普通PC机进行性能测试，4线程运行的程序的速度是串行运行程序的3.6倍。关于程序的更大规模的并行问题，我们会在下一章做进一步讨论。

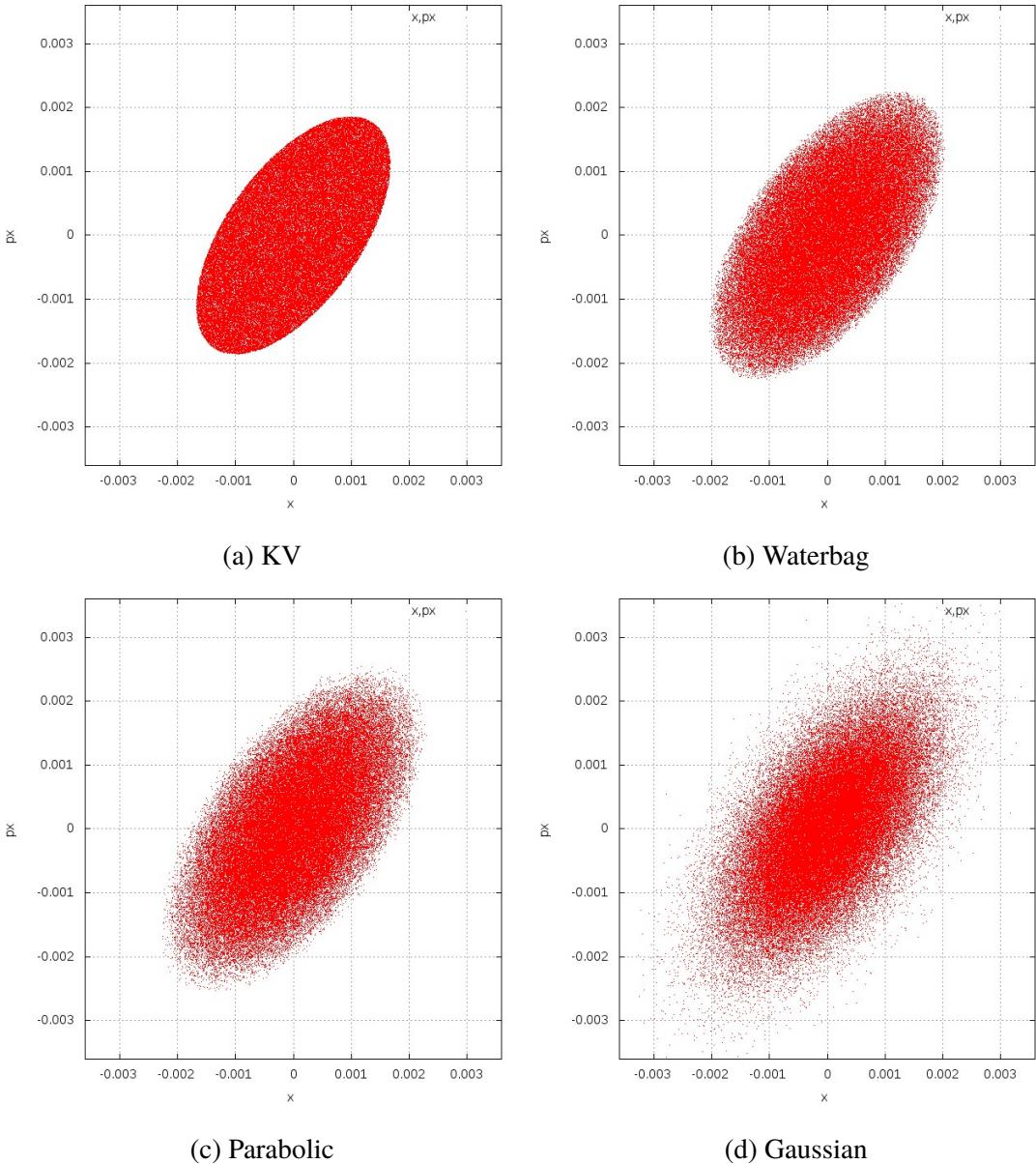


图 3.2: 同一均方根尺寸下的不同种类束团分布

3.2 正确性校验

3.2.1 内场计算

我们使用一个点电荷来测试PIC部分的正确性。使用的格点数目为 $128 \times 128 \times 64$ 。横向边界为狄利克雷边界条件（第一类边界条件），纵向为周期性边界条件。图3.3是P-TOPO的结果与理论值的对比，其中左图为横向电势（X方向），右图为纵向电势（Z方向）。图中红色实线为程序通过PIC方法计算得到的电势场的大小，绿色虚线为理论值，可以看出，程序的结果与理论值非常吻合。

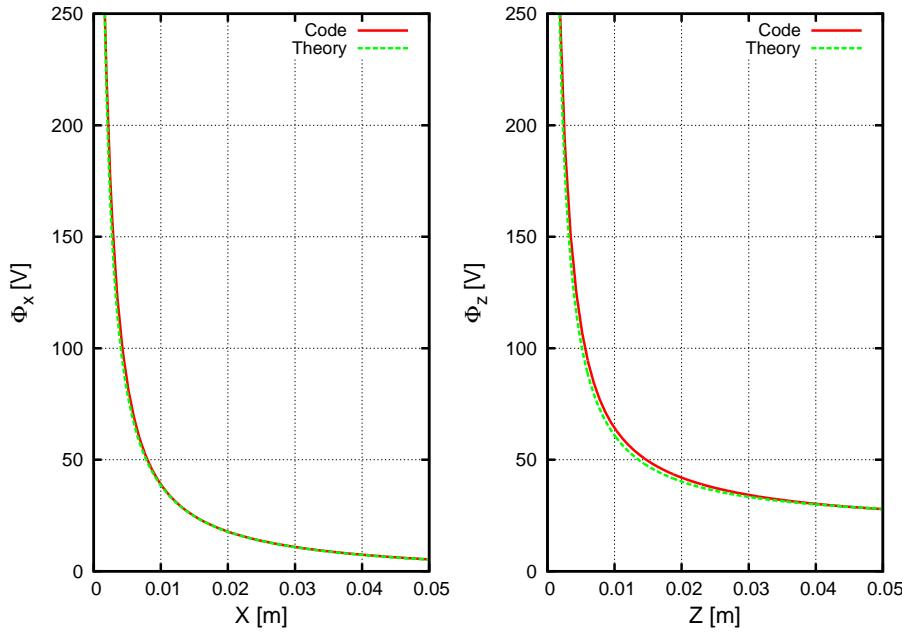


图 3.3: P-TOPO 与理论值的点电荷电势对比

3.2.2 FD结构中与解析解的对比

我们以束流在FD结构中束流的演化为例，来验证P-TOPO程序整体的正确性。连续束在周期聚焦结构中的束流包络方程可以表示为式3.1[23, 24]

$$\frac{d^2R}{dz^2} + k_0^2 R - \frac{\varepsilon^2}{R^3} - \frac{K}{R} = 0 \quad (3.1)$$

其中 R 是束流横向均方根尺寸， k_0 是聚焦强度，与外场力相关， ε 为发射度， K 是空间电荷力强度。束流的横向均方根尺寸与初始匹配和相移有关。在这个验证中，计算空间电荷力的网格数目为 $64*64$ ，我们使用 10000 个宏粒子在 KV 初始分布下，分别以 0mA 和 15mA 在周期性 FD 结构中前进，每个 FD 结构中推动 400 步。图 3.4 显示了由 P-TOPO 给出的束流均方根尺寸的演变和均方根包络方程的理论预期，其中图片下部浅蓝色线表示外部四级铁的聚焦强度，红色实线和绿色虚线分别为零流强下理论结果和 P-TOPO 程序计算结果，而蓝色实线和紫色虚线分别代表 15mA 流强下理论预期和 P-TOPO 计算结果。可以看出，无论是在零流强时还是 15mA 流强时，计算结果与理论值都保持高度一致，其中在 15mA 下束团演化的微小区别是因为包络方程计算中假定发射度为常数，而实际的 PIC 计算中发射度会发生变化。

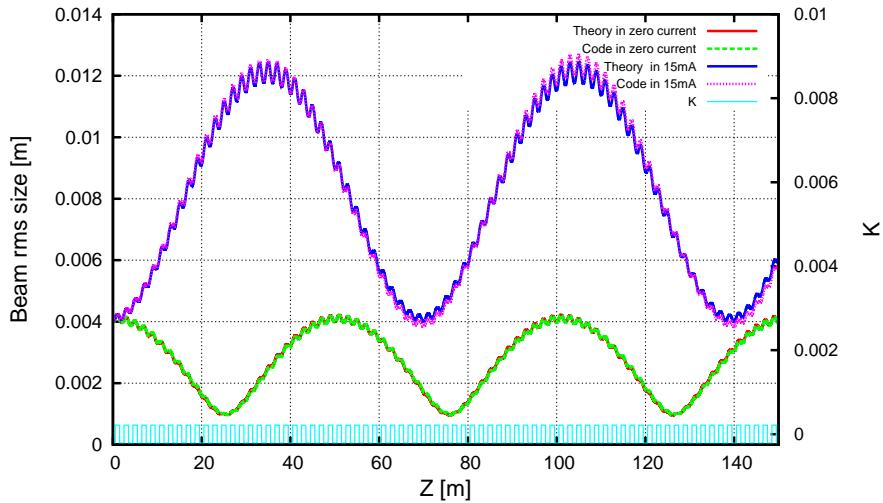


图 3.4: 零流强和15mA时, P-TOPO结果与包络方程的对比

3.3 C-ADS注入器I模拟

我们使用P-TOPO代码模拟了C-ADS的注入器I。加速器驱动次临界洁净核能系统（Accelerator Driven Sub-critical System, ADS），是一种新型的能源装置，其基本原理是利用经过加速器加速的高能质子，撞击重靶核发生反应，一个质子引起的散裂反应可产生几十个中子，再使用散裂产生的中子作为中子源来驱动次临界包层系统，使次临界包层系统维持链式反应以便得到能量和利用多余的中子增殖核材料和嬗变核废物，其中中国ADS（C-ADS）的加速器部分由中国科学院高能物理研究所和近代物理研究所共同研究。C-ADS的注入器I由离子源（electron cyclotron resonance ion-source, ECRIS），低能传输线，RFQ，中能传输线，两个超导加速模块，以及最后的垃圾桶组成，其结构如图3.5所示

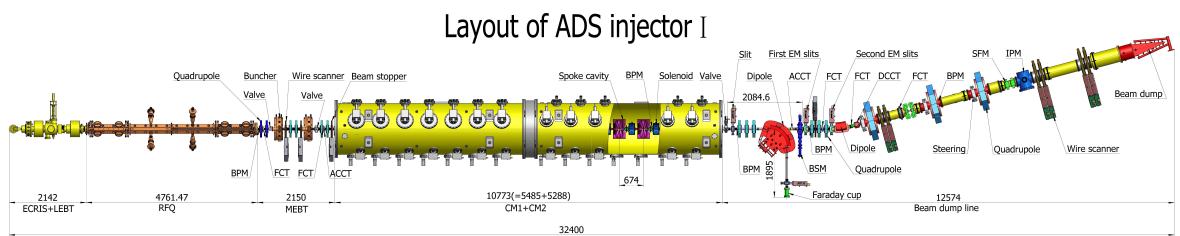


图 3.5: C-ADS注入器I结构示意图

C-ADS注入器I的基本参数如表3.1所示。其中RFQ是使用PARMTEQM[25]设计的，频率为325MHz，将10mA的质子束流从35keV加速到3.2MeV，并且以后有可能升级到15mA。因为注入能量较低，只有35keV，RFQ的参数和聚束节使用绝热设计以降低空间电荷效应的影响，这导致了最终较小的纵向发射度。RFQ之后是超

导加速腔，经过14个Spoke腔将束流加速到最终的10MeV。

Particle	Proton
Rf frequency (MHz)	325
Injection energy (MeV)	0.035
Output energy (MeV)	10
Pulsed beam current (mA)	10
Beam duty factor	100%
Input normalized rms emittance X (π mm mrad)	0.2
Input normalized rms emittance Y (π mm mrad)	0.2

表 3.1: C-ADS注入器I基本参数

我们对RFQ和后面的超导段分别进行模拟，宏粒子数目为20000个，横向初始分布使用KV分布，纵向初始分布为均匀分布。使用实际运行的加速器结构，并且当粒子的横向位置触碰元件的孔径后标记为丢失，其中RFQ稍微严格对待，其孔径为当前cell的最小半径。空间电荷效应的网格数为64*64*64 (x/y/z)。在RFQ段，我们不但使用P-TOPO程序，同时也使用Track程序[9]，以相同的初始条件进行模拟；同样的，在超导段，除了P-TOPO之外，我们还使用TraceWin[6]来进行研究和对比。下面，我们对RFQ和超导段分别进行讨论。

3.3.1 RFQ模拟

在RFQ模拟中，RFQ中的场可由傅里叶贝塞尔方程的八项式得到，如式3.2:

$$\begin{aligned}
 U_{ex}(r, \theta, z) = & \frac{V}{2} [A_{01}(r/r_0)^2 \cos(2\theta) + A_{10}I_0(kr) \cos(kz) \\
 & + A_{03}(r/r_0)^6 \cos(6\theta) + A_{21}I_2(2kr) \cos(2\theta) \cos(2kz) \\
 & + A_{12}I_4(kr) \cos(4\theta) \cos(kz) + A_{03}I_0(3kr) \cos(3kz) \\
 & + A_{23}I_6(2kr) \cos(6\theta) \cos(2kz) + A_{32}I_4(3kr) \cos(4\theta) \cos(3kz)]
 \end{aligned} \tag{3.2}$$

其中 I_n 为第n阶修正贝塞尔方程， $k = \pi/2\beta\gamma$, $A_m n$ 系数由RFQ设计程序PARMTEQM给出。

图3.6是P-TOPO和TRACK在0mA下和15mA下对RFQ模拟得到的横向和纵向发射度演化，其中红色实线为P-TOPO的结果而绿色虚线为TRACK的结果。在横向和纵向两个方向上，P-TOPO的发射度演变结果都更加光滑，尤其是在RFQ的前端。

在此处，束流开始丝化形成束团，并且相空间开始剧烈旋转。P-TOPO与TRACK的结果误差在合理范围内。

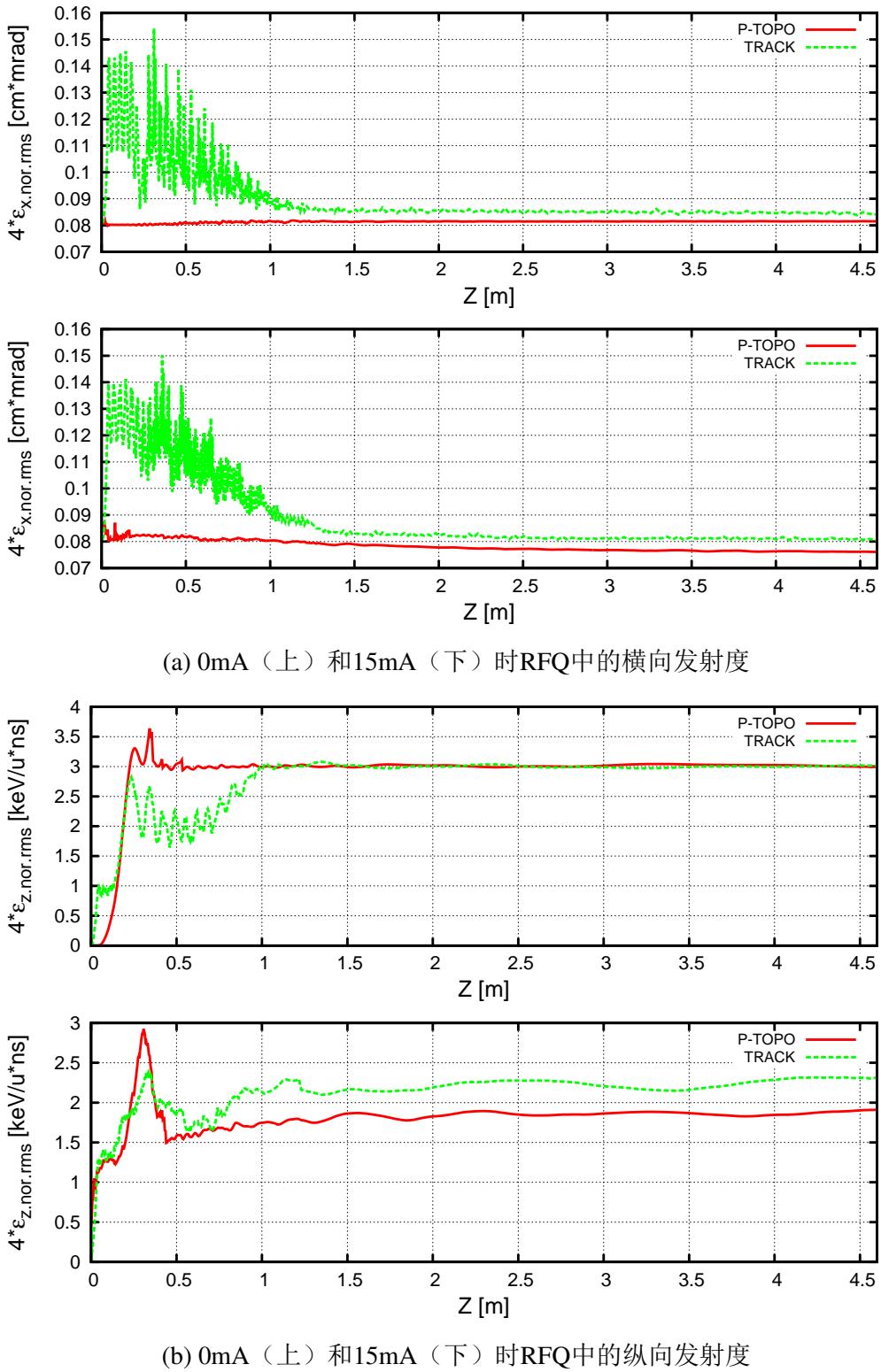


图 3.6: RFQ 中的横向发射度和纵向发射度

图3.7是P-TOPO和TRACK在15mA下对RFQ模拟得到的横向束团尺寸演化，图3.8是P-TOPO和TRACK在15mA下对RFQ模拟得到的纵向束团尺寸和能散演化，其中红色实线为P-TOPO的结果而绿色虚线为TRACK的结果。两个程序在束团均方根尺寸上吻合得很好，其差别在合理范围内。同时验证了C-ADS注入器I的RFQ设计能够有效的控制束团的发射度和尺寸。

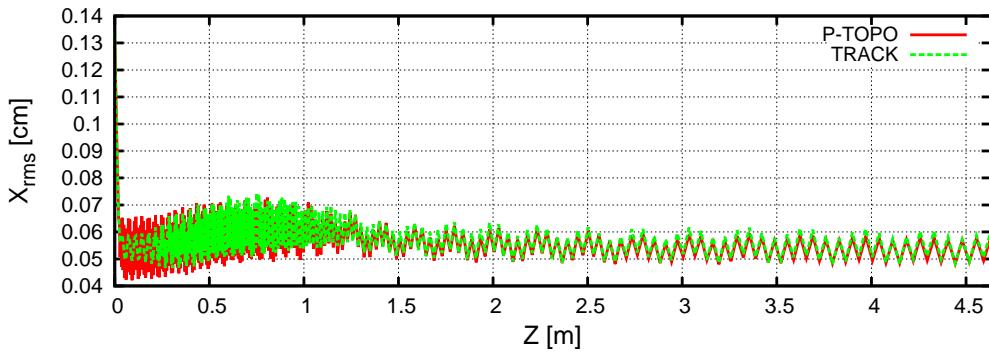


图 3.7: RFQ中束团横向均方根尺寸

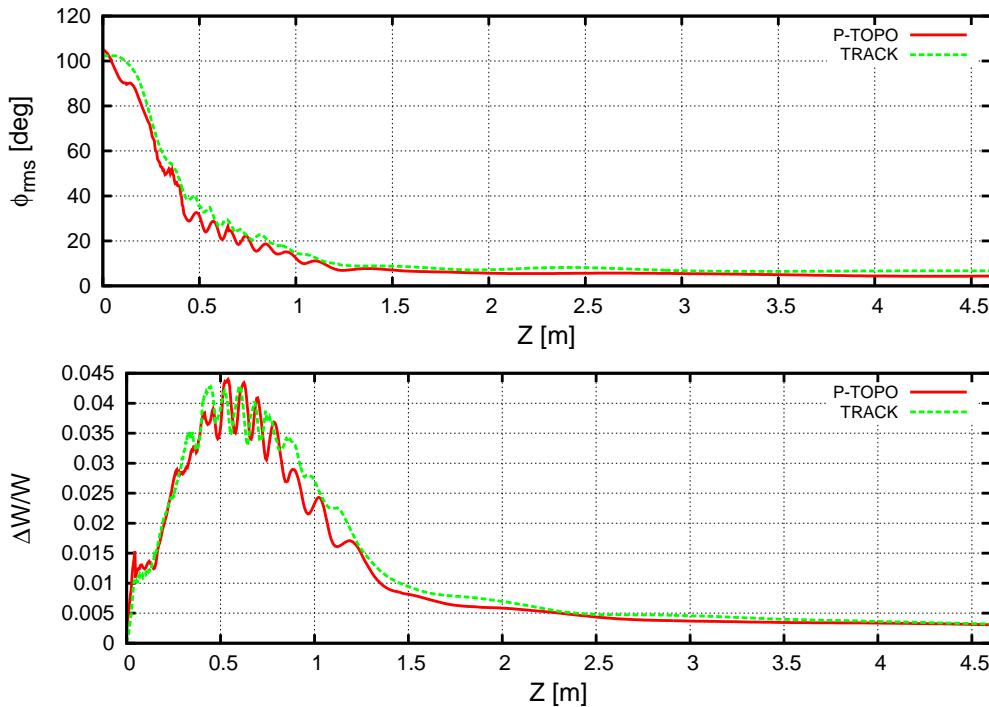


图 3.8: RFQ中束团纵向均方根尺寸和能散

3.3.2 超导段模拟

在超导段中，聚束腔和加速腔的场由文件导入，P-TOPO使用数值差值来得到粒子所受到的场强。在设计的15mA流强下，RFQ出口处3.2MeV的质子束流经过聚

束，通过MEBT进入超导段，逐渐加速到10MeV。在超导段中我们使用P-TOPO和TraceWin来进行模拟。

图3.9为P-TOPO和TraceWin在15mA下对RFQ模拟得到的横向和纵向发射度演化，其中红色实线为P-TOPO的结果而绿色虚线为TraceWin的结果。两个程序得到的结果相一致。其中横向发射度除了在螺线管处有一个峰值外，基本保持不变。螺线管处的峰值是因为P-TOPO和TraceWin都是以时间t为基本变量，即发射度是由同一时刻的粒子信息统计得到，而不是由同一纵向位置的粒子得到，因此，在束团进入螺线管的时候，部分粒子先进入，部分粒子还没有进入，导致相空间的扭曲，从而导致统计发射度的突变。纵向发射度有略微增长，增长幅度在20%以内。横纵向的发射度增长是由空间电荷力，磁铁边缘场的非线性效应，以及超导腔内的非线性纵向力导致，这几个驱动发射度增长的来源共同作用，使粒子相空间扭曲，最终会导致束晕的产生。P-TOPO的模拟中，出口能量为10.01MeV，而TraceWin的模拟出口能量为10.06MeV。

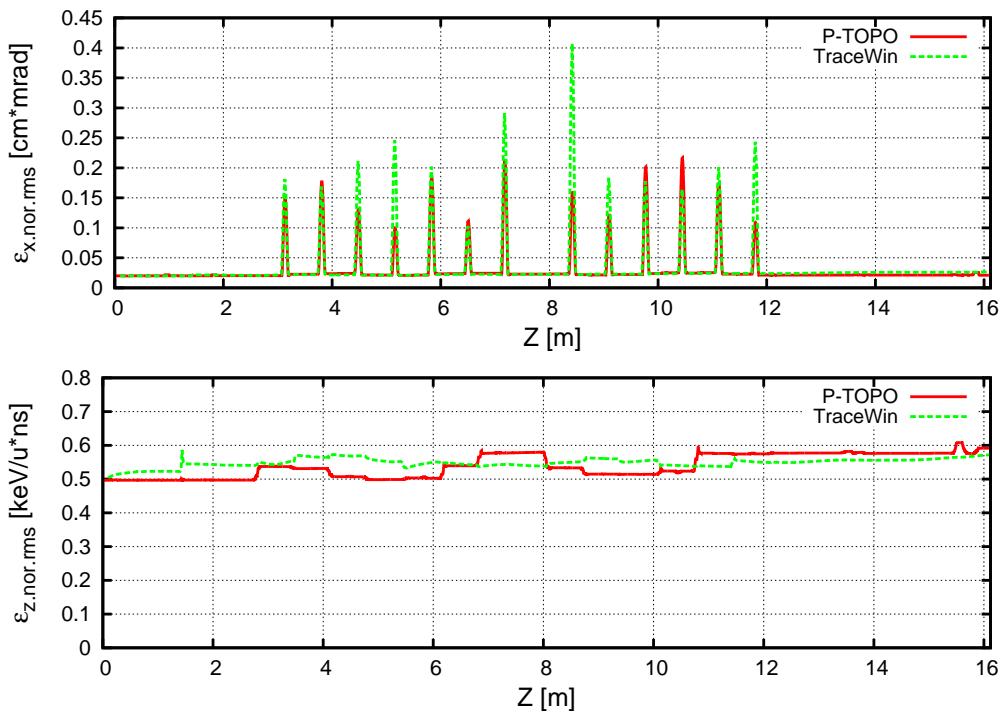


图 3.9: 超导段横向发射度和纵向发射度

图3.10是P-TOPO和TraceWin在15mA下对超导段模拟得到的横纵向束团尺寸演化和能散演化，其中红色实线为P-TOPO的结果而绿色虚线为TRACK的结果。两个程序模拟结果的细微差别主要是因为两个程序获取同步相位的方法不同。TraceWin使用时间漂移法获得同步相位，而P-TOPO使用扫相获得同步相位。束团的横向尺寸

相比管道半径都很小，而纵向尺寸也被聚束腔和接下来的各个加速腔有效地压缩，一般不会有束损出现。

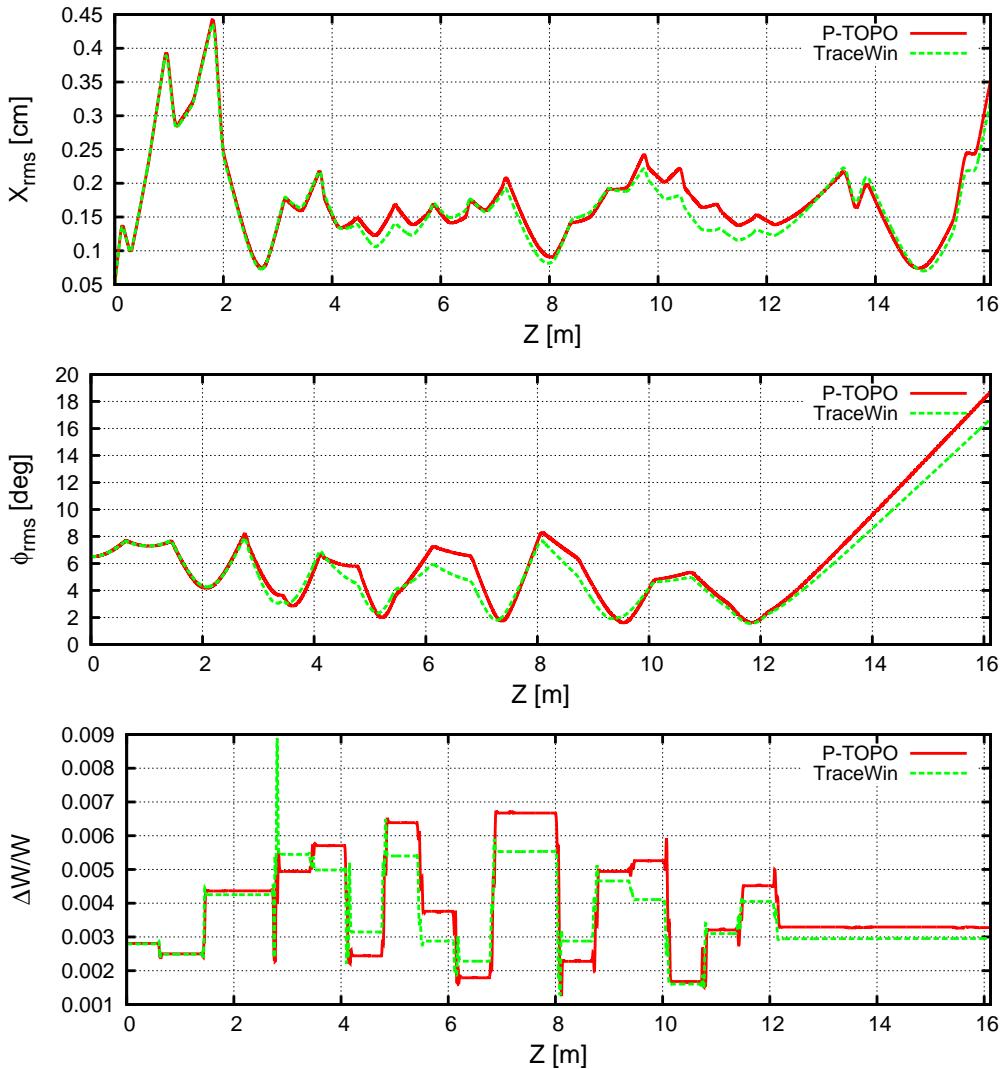


图 3.10: 超导段束团横纵向尺寸以及能散

在RFQ和超导段两段模拟中，传输效率都在99.9%以上，束团尺寸和束损都得到了有效控制，横向和纵向的发射度都保持良好。不同代码的模拟结果差别主要是因为初始的束团分布不同，以及统计数据的方式不同，特别是P-TOPO是以时间t为基本变量，而TRACK是以位置z为基本变量，因此粒子信息收集，空间电荷力的计算，以及粒子推动方式都有所不同。

3.4 小结

我们在介绍了根据PIC算法编写的粒子模拟程序P-TOPO，主要处理强流加速

器中的非线性效应。P-TOPO的PIC部分和整体正确性都得到了验证，并应用在C-ADS注入器I的RFQ和超导段，分别对其进行了模拟并且与其他程序进行了比较。模拟结果证明了现有设计的合理性，束团的尺寸和发射度都得到了有效控制，束损以及能散也在合理范围之内，完全满足需求。之后，我们将继续对P-TOPO进行拓展并加入更多新的功能，以满足强流加速器的各种需求。

第四章 PIC算法在GPU上的实现

我们已经在2.9中介绍了基本的PIC算法。在本章中，我们致力于在GPU上实现基于PIC算法的多粒子跟踪程序的并行化，提高程序的运行效率。

在本章中，首先，我们会介绍GPU程序的基本结构和并行策略。之后，我们做了一个GPU程序和CPU程序结果的对比，以验证程序的正确性。再后，我们展示了GPU程序的效率和在不同规模的问题下的加速比。最后，我们也实现了此程序在新的CPU架构，Knight Landing，上的并行化运行，并与GPU的运行效率进行了对比。

4.1 GPU程序结构

在2.9中我们介绍了，PIC算法主要包括四部分：权重差值，解泊松方程，将场反权重到粒子，推动粒子。在GPU上实现PIC算法，主要难点在第一步权重差值上。第一步权重插值是将粒子逐个权重到网格上的过程，此步骤多线程并行运行时，会遇到不同线程相互竞争的问题（race condition），从而导致错误的结果。

线程竞争是指在程序中，协作的线程可能共享一些彼此都能读写的公用存储区，而竞争出现在当两个或多个线程同时访问同一内存地址，并且尝试写入数据的时候。由于线程调度器会以任意顺序运行多个线程，而我们无法得知线程尝试访问共享内存的顺序。因此，数据的最终结果会取决于线程调度器的算法，即多个线程“竞争”访问和改变数据。

图4.1显示了PIC算法中权重差值部分的运算流程和多线程下可能会出现的竞争和冲突结果，其中蓝色点代表粒子，而绿色星状点代表与粒子相近的网格。使用多个线程运行权重插值时，通常使不同的线程处理不同的粒子。假设我们使用五个时钟锁步线程，如图4.1中所示，线程A-E分别处理五个粒子。A，B，C三个线程并没有发生冲突，但是D线程和E线程会产生竞争，造成错误的运算结果。在第一个时钟周期内，线程D和E分别从同一内存地址M，读取格点原本的电荷量V；在第二个时钟周期，线程D读取粒子D的电荷并计算相加到格点的电荷V+D，而线程E读取粒子E并计算得到V+E；在第三个时钟周期，线程D将V+D写入内存地址M，同时线程E将V+E写入同一内存地址M。但是两个写入都是错误，正确的结果应该是V+D+E。

为了避免线程之间的竞争和冲突，我们采取先将网格分块，再随后将网格上的电荷合并的做法。如图4.2所示，绿色部分为一个区块，而一个线程单独处理一

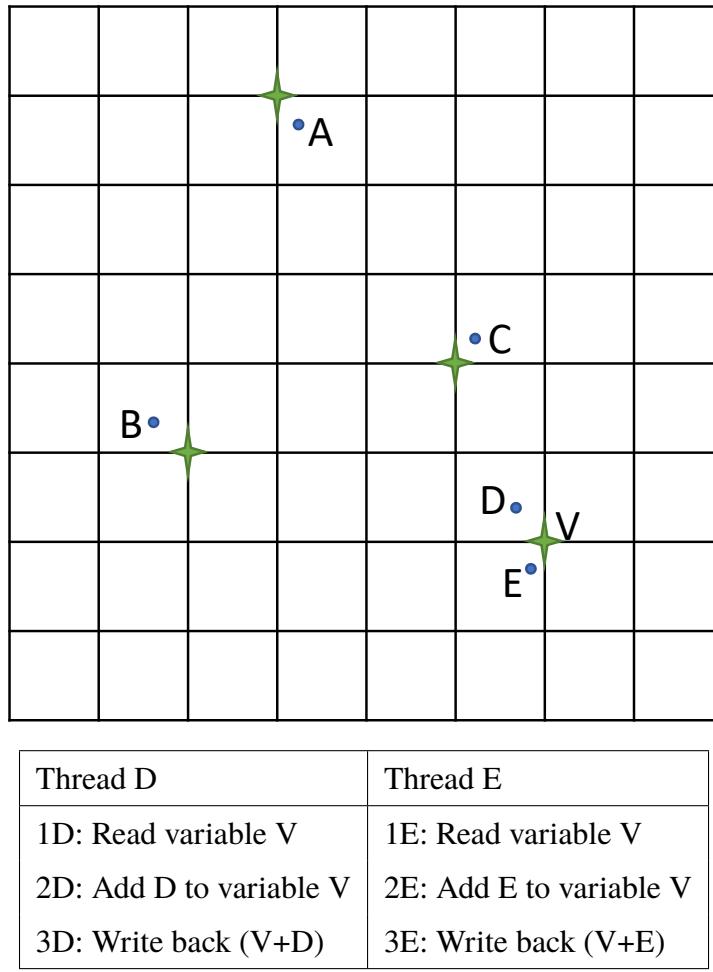


图 4.1: 权重差值中的线程冲突

个区块。想要分块处理，粒子必须是有序的，这样才能使相应的线程找到所对应区域的粒子。所以在插值权重之前，我们需要对粒子进行排序（reorder）。假设区块数目为N，我们需要声明N个数组，以分别对应N个区块中的粒子。在程序初始阶段，我们将粒子数据拷贝到不同的区块数组中，并且在随后的每一步中，粒子可能改变位置，所以我们要对粒子重新排序，以确保粒子处于所对应的区块数组。用这种方法，每一个线程能够单独处理自己所分配的区块，从而避免了线程之间的竞争和冲突。加入了排序之后的PIC算法流程如图4.3所示，其中，标识为红色部分为粒子排序，是PIC算法在GPU上并行运行所必须的前置步骤；而黄色部分的GPU算法和普通CPU算法有较大区别，我们在下面，对流程图种的黄色部分，即GPU上的PIC算法中的粒子排序，权重插值，解泊松方程，以及粒子推动做进一步讨论。

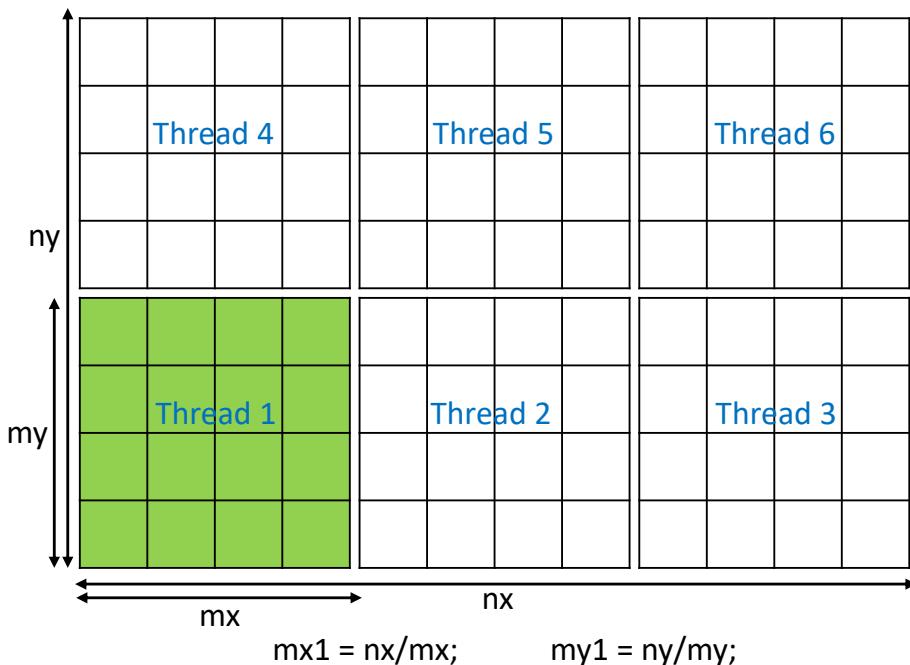


图 4.2: 网格分块

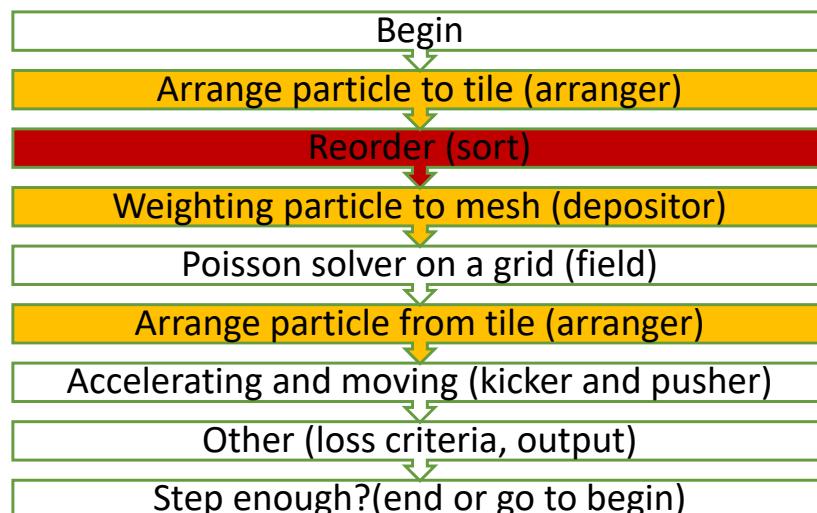


图 4.3: GPU上的PIC算法分段流程

4.1.1 粒子排序

如上所述，GPU上PIC算法中，我们必须要将粒子在每一步进行重新排序，以避免线程竞争。排序在GPU中的实现并不直接，因为其高度不规则的并且难以并行执行。在这里，我们使用临时缓冲区作为中继，以实现粒子排序算法在GPU上的并行运行。我们将整个网格划分为不同的区块，使每个线程只需处理对应的区块即可。例如，在一个网格数为 $64 \times 64 \times 64$ 的模拟中，我们将其分割成大小为 $4 \times 4 \times 4$ 的

区块，总共 $16*16*16=4096$ 个区块，这样我们可以避免线程的竞争。

在程序初始化的时候，我们首先需要为每一个区块声明数组并为其分配内存空间，如以下代码所示。其中，我们根据实际使用的GPU的可用内存大小来确定每个区块的最大粒子数目。如果某一区块的粒子数目超过了最大粒子数，程序会报错并停止。在这种情况下，使用更少的粒子数目或者使用内存更大的显卡可以解决问题。因为在实际模拟中，每个区块的粒子数一般并不相同，使用内存预分配会导致浪费很多内存空间。但是不同于CPU内存，在GPU上分配内存耗时很长，所以无法在每一步计算中根据实际情况来进行内存的分配，只能采用预分配的方法，以提高程序运行效率。

```

const int lth =BLOCKSIZE;
const int dim =_dimension;
int mth      = (mx1*my1*mz1-1)/lth+1;      //number of tiles

double *ptc;        //tiled particle
int    *kpic;       //number of particles in each tile
int    *nhole;       //number of hole left
int    *ncl;         //ndirec[k*26 + i]: number of particle going
                     //to destination i[0-26] from tile k
double *pbuf;       //buffer for transfer particles

//get the maximum number of particle according to the GPU memory size
size_t freeMem,totalMem;
cuda_SafeCall(cudaMemGetInfo(&freeMem,&totalMem));
int npm = freeMem / 2 / sizeof(double) / dim / (mx1*my1*mz1);
if(npm>numberOfParticle) npm = numberOfParticle;
int MaxPtcTransf = npm / 2;

//allocate memory for particle array
size_t ptc_mem_size = sizeof(double)*lth*dim*npm*mth;
cuda_SafeCall(cudaMalloc((void**) &ptc,      ptc_mem_size));

//allocate memory for particle counting array
size_t kpic_mem_size = sizeof(int)*mx1*my1*mz1;
cuda_SafeCall(cudaMalloc((void**) &kpic,      kpic_mem_size));

//allocate memory for holes at tiles array
size_t nhole_mem_size = sizeof(double)*lth*2*(MaxPtcTransf+1)*mth;

```

```

cuda_SafeCall((cudaMalloc((void**) &nhole,  nhole_mem_size));

//allocate memory for particle buffer array
size_t pbuff_mem_size = sizeof(double)*lth*dim*MaxPtcTransf*mth;
cuda_SafeCall((cudaMalloc((void**) &pbuff,  pbuff_mem_size));

//allocate memory for transferred particle counting array
size_t ndirec_mem_size = sizeof(int)*mx1*my1*mz1*(3*3*3-1);
cuda_SafeCall((cudaMalloc((void**) &ndirec,  ndirec_mem_size));

```

在之后的每一步计算中，在空间电荷效应之前，我们都需要根据粒子空间位置对粒子进行排序，排序流程如图4.4所示：

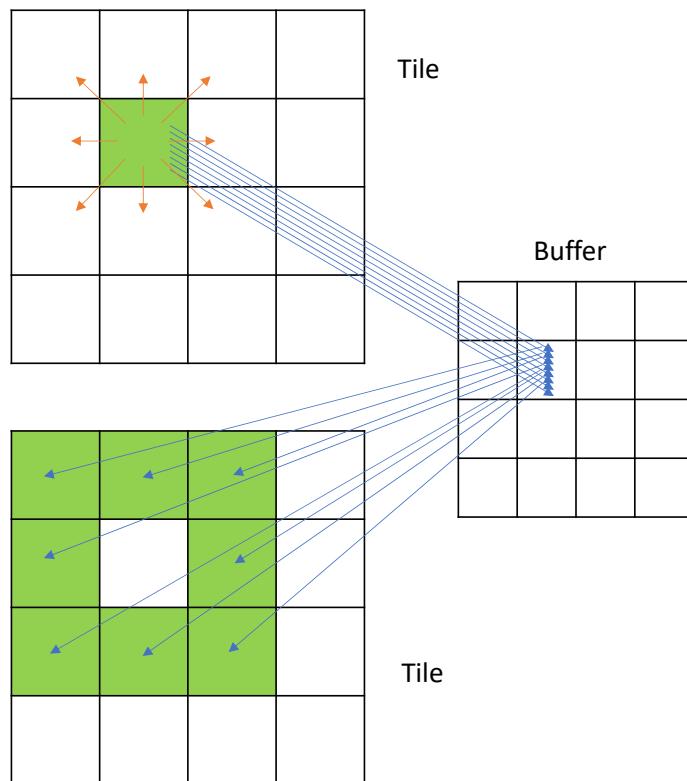


图 4.4: 粒子排序示意图

首先，每个线程统计对应区块中将要离开当前区块的粒子信息，并使用数组nhole和ndirec记录粒子的序号和每个方向的数目。如图4.4中黄色箭头所示。如上面代码所示，我们使用最大粒子数目的一半作为最大穿越数目，而nhole已经根据最大穿越数目进行了内存预分配，同样的，如果离开当前区块的粒子数目超过了最大穿越数目，程序会报错停止，这时候需要使用更少的粒子数目或者更大的GPU内存。

之后，我们将离开当前区块的粒子信息拷贝到一个缓冲区**pbuf**。作为全局内存，**pbuf**同样根据最大穿越数目进行了内存预分配。**ndirec**包括了每个方向上的粒子数目，对**ndirec**进行迭代求和（running sum），我们可以得到每个方向上的内存初始位置，以便于将前往某个方向的粒子在**pbuf**中连续排布。

最后，对于每个区块，经过第一步的统计，我们可以知道会有多少粒子进入，以及它们在**pbuf**中的位置。在将粒子信息从**pbuf**拷贝到当前区块**ptc**的过程中，我们先填满因为粒子离开造成的空洞。如果如果进入的粒子数目大于空洞数目，即在所有空洞都被填满后，仍有粒子进入，我们将其写入在当前区块**ptc**的最后；但如果进入的粒子数目小于空洞数目，即写入所有进入粒子后仍有空洞存在，则从本区块粒子数组**ptc**尾部开始，依次移动粒子数据到空洞处。通过这种方式，粒子信息在内存中的连续性得到了保证。

我们对上述步骤做了一个简要总结，通过这些排序流程，使每个线程仅处理对应的区块，来确保不会出现线程竞争。

- 首先，对本区块内的粒子进行遍历，将因为位置改变而不再属于本区块的粒子序号，以及其离开的方向记录下来，写入**nhole** 和**ndirec**。
- 其次，根据其序号和方向，将离开的粒子拷贝进入全局缓存区**pbuf**。
- 最后，再根据相邻区块离开粒子的序号和方向，确定将要进入本区块的粒子，将缓存区内的数据拷贝到本区块。

4.1.2 权重插值

通过以上粒子重排，我们使得每一个区块的粒子相对独立，这样，我们可以使每个线程处理一个区块，从而避免了线程冲突的情况。一个直接的权重插值流程如图所示4.5：

- 首先，每个线程将自己所处理的粒子权重到一个小网格上，得到本地的网格密度分布。
- 然后，我们将所有本地的密度分布结合到一起，得到全局密度分布。

上述为单GPU的权重差值方法，在使用多个GPU的时候，我们有两种方法：一种是使不同的GPU处理不同的空间域，在最后进行通讯，这种方法在CPU上的PIC程序中很常用，被称为“域分解”；另外一种是简单直接的使所有GPU做同样的工作，我们称这种模式为“复制模式”。下面，我们使用一个例子对这

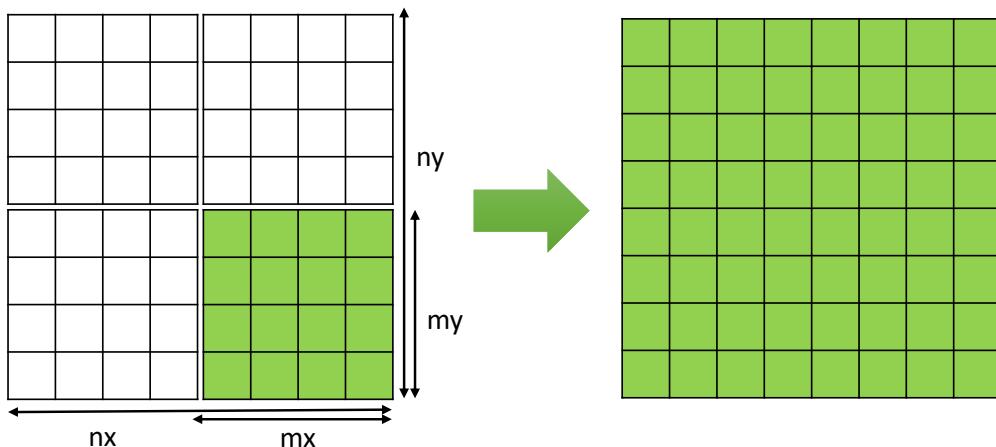


图 4.5: 权重插值示意图

两种方法进行比较，假设使用 $64*64*64$ 个格点，区块大小为 $4*4*4$ ，区块的数目为 $16*16*16$ ，在TITAN上使用4个GPU运行。

4.1.2.1 域分解模式

在域分解模式中，每个GPU只需要处理相对应的域。假如区块数目为 $16*16*16$ ，在4个GPU上运行的话，每个GPU需要处理的域的大小为 $4*16*16$ 。但是，这种模式需要预先将粒子排序，使每个GPU只处理对应空间域内的粒子，因此需要额外的通讯和计算，具体步骤如下：

1. GPU之间的粒子排序

(a) 挑选粒子，每个线程处理一个区块，因此我们共有 $4*16*16=1024$ 个线程。然而，在TITAN上每个GPU有2688个核，线程数小于核数，我们无法充分使用GPU。

(b) GPU之间信息交换

- i. 从GPU内存拷贝到CPU内存，共需要拷贝 $4*16*16*(nGPU-1) * nPtcMax$ 个粒子，其中nPtcMax个粒子为最大传输粒子数。
- ii. CPU和CPU之间的信息交换，我们使用MPI_send/receive，共需要交换 $4*16*16*(nGPU-1) * nPtcMax$ 个粒子。
- iii. 从CPU内存拷贝到GPU内存，共需要拷贝 $4*16*16*(nGPU-1) * nPtcMax$ 个粒子。

2. GPU内部的粒子排序，每个GPU的区块数目为 $4*16*16=1024$ ，小于在每个GPU上的核数2688。
3. GPU内部的权重差值，由于第一步进行了GPU之间的粒子排序，所以每个GPU上的粒子数目不同，格点数目为 $16*64*64$ 。
4. GPU之间的格点信息归约。
 - (a) 从GPU内存拷贝到CPU内存，共需要拷贝 $16*64*64$ 个格点。
 - (b) CPU和CPU之间的信息交换，我们使用MPI_Allgather，共需要交换 $64*64*64$ 个格点。
 - (c) 从CPU内存拷贝到GPU内存，共需要拷贝 $64*64*64$ 个格点。

4.1.2.2 复制模式

在复制模式中，每个GPU进行同样的工作，同域分解模式相比，复制模式无法使用多GPU来节省计算时间，但是也无需在GPU之间进行粒子信息的交换。其流程如下所示，其中每一项和上面域分解模式相对。

1. 无需GPU之间排序
2. GPU内部的粒子排序，每个GPU的区块数目为 $16*16*16=4096$ ，大于在每个GPU上的核数2688。
3. GPU内部的权重差值，由于第一步进行了GPU之间的粒子排序，所以每个GPU上的粒子数目相同，格点数目为 $64*64*64$ 。
4. GPU之间的格点信息归约。
 - (a) 从GPU内存拷贝到CPU内存，共需要拷贝 $64*64*64$ 个格点。
 - (b) CPU和CPU之间的信息交换，我们使用MPI_Allreduce，共需要交换 $64*64*64$ 个格点。
 - (c) 从CPU内存拷贝到GPU内存，共需要拷贝 $64*64*64$ 个格点。

可以看出，两个模式相比较的话，域分解模式在第一步中有了很多额外的通讯，在第二步与第三步中能够使用更少的格点数。然后，使用通讯时间来换取更少的计算量并不能带来效率的提升，而且，在域分解模式下，我们并不能充分利用GPU。因此，在我们的代码中以及接下来的性能测试中，我们均采用“复制模式”。

4.1.3 泊松方程

在将粒子权重到网格上之后，下一步即解网格上的泊松方程。在单GPU上运行的版本我们在此不再赘述，而在多GPU上的并行版本和权重差值相似，在也存在两种方法：一种是参考经典的CPU上的PIC程序，使用不同的GPU处理不同的空间域，即“域分解模式”；另外一种是简单直接的使所有GPU做同样的工作，即“复制模式”。由于解泊松方程的运算量较大，是PIC方法中很重要的一步，我们对两种模式都进行了实现并进行了比较。

4.1.3.1 域分解模式与复制模式

域分解的优点在于，通过使用不同的GPU处理不同的空间域，每个GPU可以降低计算量，从而提高程序运行速度；缺点也很明显，域分解模式需要不同GPU之间的信息交换，而在目前GPU与GPU之间，特别是在不同节点之间，并不能直接进行信息交换，而是必须通过CPU来进行，即信息交换需要三步：

1. 从GPU内存拷贝到CPU内存。
2. CPU和CPU之间的通过MPI进行信息交换。
3. 从CPU内存拷贝到GPU内存。

如小节2.9.2所示，求解泊松方程中最主要的部分是使用傅里叶变换（FFT）。在GPU代码中，我们使用NVIDIA公司的CUDA快速傅里叶变换库（cuFFT）[26]在进行变换。在求解多GPU上的3维傅里叶变换时，最好是在三个方向分别做1维的傅里叶变换，并在每次变换之间对数据进行转置。

假设在X, Y, Z三个方向上的格点数分别为 N_x , N_y , N_z ，那么当执行X方向的傅里叶变换的时候，变换的数组长度即为 N_x ，变换的次数为 $N_y \times N_z$ 。以使用四个GPU为例，那么1号GPU需要处理的数据为 $\text{rho}[N_x][0 \rightarrow \frac{N_y}{4}][N_z]$ ，2号GPU需要处理的数据为 $\text{rho}[N_x][\frac{N_y}{4} \rightarrow 2\frac{N_y}{4}][N_z]$ ，3号GPU需要处理的数据为 $\text{rho}[N_x][2\frac{N_y}{4} \rightarrow 3\frac{N_y}{4}][N_z]$ ，4号GPU需要处理的数据为 $\text{rho}[N_x][3\frac{N_y}{4} \rightarrow N_y][N_z]$ 。每个GPU只需要进行 $\frac{N_y}{4} \times N_z$ 次变换即可。理想情况下，和单GPU运行相比，使用4个GPU运行只需要花费1/4的时间。

在X方向的傅里叶变换结束后，我们需要其他的数据来进行Y方向的运算，目前每个GPU上的数据为 $\text{rho}[N_x][(n-1)\frac{N_y}{4} \rightarrow (n)\frac{N_y}{4}][N_z]$ ，而进行Y方向的傅里叶变换需要的数据为 $\text{rho}[(n-1)\frac{N_x}{4} \rightarrow (n)\frac{N_x}{4}][N_y][N_z]$ 。我们必须对数据求转置并在GPU之

间进行数据交换。而由于目前GPU与GPU之间并不能直接进行数据交换，我们不需将GPU中的数据拷贝回CPU内存中，并在CPU端进行通讯，这将需要额外的时间。所以域分解模式相比复制模式的效率高低将取决于额外的通信时间与降低的运算时间的比较。

以 $64*64*64$ 个格点数，使用4个GPU为例，域分解模式求解泊松方程的具体步骤可以表示如下：

1. X方向的FFT

- (a) 拷贝到GPU ($64*16*64$)
- (b) FFT ($64*16*64$)
- (c) 拷贝到CPU ($64*16*64$)

2. 转置及CPU与CPU之间的信息通讯

3. Y方向的FFT

- (a) 拷贝到GPU ($64*16*64$)
- (b) FFT ($64*16*64$)
- (c) 拷贝到CPU ($64*16*64$)

4. 转置及CPU与CPU之间的信息通讯

5. Z方向的FFT

- (a) 拷贝到GPU ($64*16*64$)
- (b) FFT ($64*16*64$)
- (c) 拷贝到CPU ($64*16*64$)

相比较而言，复制模式的步骤为：

- 1. X方向的FFT ($64*64*64$)
- 2. Y方向的FFT ($64*64*64$)
- 3. Z方向的FFT ($64*64*64$)

4.1.3.2 单GPU测试

我们在单GPU上对域分解模式进行了测试，并统计了每一部分所花费的时间。测试的格点数为 $64*64*64$ ，每部分所花费的时间如表4.1，表中的单位为秒，每部分占比如图4.6。

从CPU拷贝到GPU	1.306
从GPU拷贝到CPU	2.979
FFT计算	0.487
转置和通讯	3.772
总计	8.544

表 4.1: 单GPU域分解模式解泊松方程

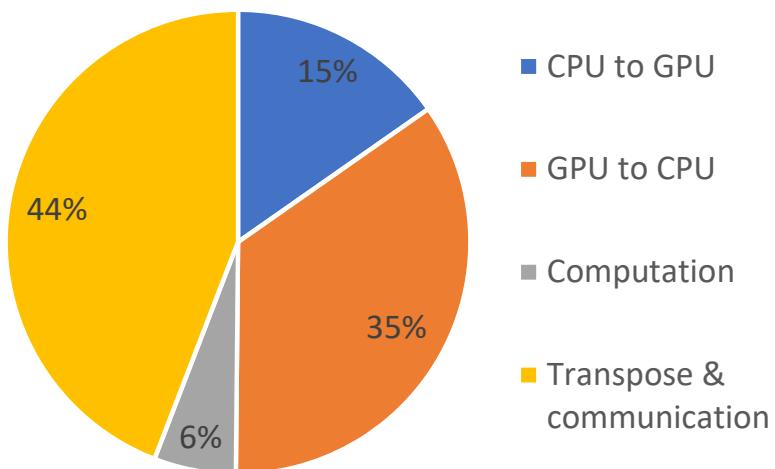


图 4.6: 单GPU域分解模式解泊松方程各部分时间占比

可以看出，FFT计算所花的时间为0.487秒，相比于额外增加的数据拷贝时间 $1.306 + 4.627 = 4.933$ 秒，其占比非常小，仅仅占总时间的6%。

多GPU的一种方式是使用同一节点内使用多个GPU，因为节点内的总线宽度限制，数据拷贝时间并不会随着使用GPU数目的增加而明显减少。而复制模式只需要花费FFT计算的时间，因为FFT计算花费的时间0.487秒远小于数据拷贝的时间4.933秒，所以即使在理想情况下使用N个GPU时FFT计算的时间减少到原来的 $1/N$ ，域分解模式的总时间依然是比复制模式要大。

多GPU的另一种使用跨节点的多个GPU，即每个节点有1个GPU，通过使用多个节点来使用多个GPU，这也是目前超级计算机TITAN的运行模式。总线宽度

会随着节点数增加而增加，所以数据拷贝时间会下降，假设其下降为现行的，即在N个节点上运行的话拷贝数据时间只有原来的 $1/N$ ，则总时间将会是 $4.933/N+0.487/N$ ，如果我们想要取得优于复制模式的速度，即 $4.933/N+0.487/N < 0.487$ ，GPU数目必须大于11。类似的，如果我们想要取得两倍的速度，GPU数目必须大于22。然而，这个计算忽略了CPU和CPU之间的通讯时间，一般来说，节点间的通讯时间随着节点数目增加而增加，因此我们很难取得理想情况的加速比。

下面，我们分别对节点内多GPU和跨节点GPU进行测试。

4.1.3.3 单节点多GPU测试

在同一节点中，我们进行了格点数为 $64*64*64$ 的多GPU测试，表4.2是各个部分所花费的时间，以及和单GPU时的对比。可以看出，双GPU的总时间更大了，由8.544秒增加到了9.537秒。其中，FFT计算的时间有0.487秒减少到了0.249秒，几乎减少了一半，证明了多GPU对于减少纯FFT计算时间是确实有效的；对于数据拷贝时间，双GPU略有减少，大概减少了三分之一；总时间中增加的部分来源于转置和通讯的时间，几乎增加了一倍。

	1GPU	2GPU
从CPU拷贝到GPU	1.306	0.817
从GPU拷贝到CPU	2.979	1.991
FFT计算	0.487	0.249
转置和通讯	3.772	6.480
总计	8.544	9.537

表 4.2: 单节点双GPU域分解模式解泊松方程

4.1.3.4 跨节点多GPU测试

我们使用超级计算机泰坦（Titan）来进行域分解模式下的解泊松方程在跨节点多GPU上的效率。泰坦是一台由克雷公司承建的超级电脑，置放于美国能源部下属的橡树岭国家实验室中，使用由超微半导体提供的皓龙（Opteron）处理器链接英伟达提供的Tesla运算用GPU以进行协同运算。

首先，我们测试了常用的 $64*64*64$ 个格点下，域分解模式下解泊松方程所花费的时间随GPU个数的变化，如图4.7所示，图中蓝线为总时间，各个柱代表各个部分所花费的时间，在GPU数目很大时，图中不清楚部分可以参考表4.3。

GPU数目	1	2	4	8	16	32	64
CPU to GPU	2.111	0.922	0.390	0.158	0.061	0.031	0.0155
GPU to CPU	1.618	0.694	0.314	0.167	0.066	0.033	0.0158
FFT计算	0.266	0.136	0.069	0.039	0.024	0.017	0.0150
转置和通讯	4.02	5.521	2.415	1.146	0.691	0.572	0.6320
总计	8.015	7.273	3.188	1.51	0.842	0.653	0.6783

表 4.3: 64*64*64格点下，跨节点多GPU 域分解模式解泊松方程各部分所用时间

可以看出，花费的总时间随着GPU数目增加而逐渐减少，并在32个GPU的时候到达最小值，随后开始逐渐增加。其中，拷贝数据所花费的时间基本随着GPU的数目线性减少；而通讯所花费的时间先随着GPU数目增加而逐渐减少，到达极值后又开始逐渐增加，此时通讯时间占程序所花费的时间的绝大部分；FFT计算所花费的时间随着GPU数目的增加而逐渐减少，但是FFT计算占总时间的百分比很小。

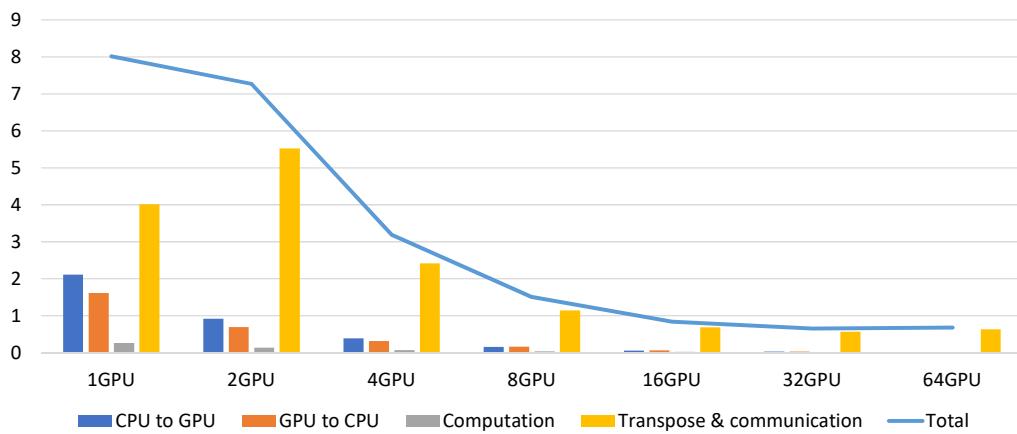


图 4.7: 64*64*64格点下，跨节点多GPU 域分解模式解泊松方程时间随GPU个数的变化

我们也测试了更多格点数目时的情况，格点数目越多，所需要的计算量越大。当格点数为128*128*128时，程序在不同的GPU数目下消耗的时间如图4.8和表4.4所示，其整体的趋势和64*64*64时类似，因为更大的计算量，其总时间在使用128个GPU时到达最小值。

在64*64*64和128*128*128两种情况下，总时间开始都随着GPU数目增加而减小，然后到达最小值，随后由于通讯所需要的时间变大，总时间也随之变大。

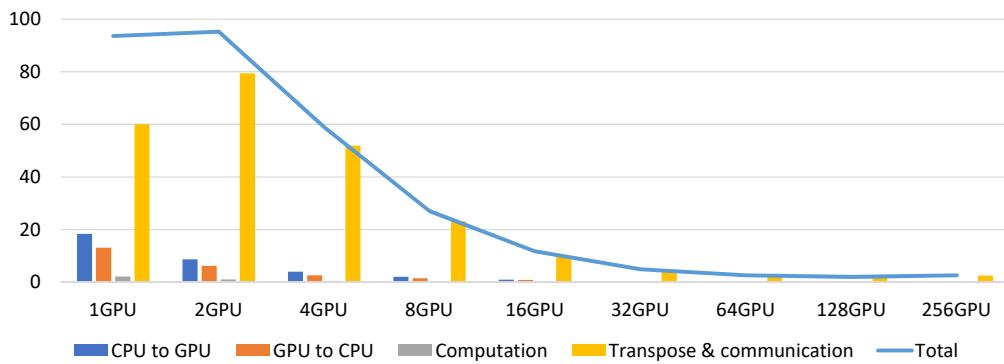


图 4.8: 128*128*128 格点下，跨节点多 GPU 域分解模式解泊松方程时间随 GPU 个数的变化

GPU数目	1	2	4	8	16	32	64	128	256
CPU to GPU	18.29	8.687	3.946	2.023	0.953	0.404	0.148	0.064	0.033
GPU to CPU	13.11	6.15	2.618	1.463	0.795	0.368	0.139	0.07	0.036
FFT计算	2.185	1.034	0.522	0.266	0.138	0.071	0.036	0.021	0.017
转置和通讯	59.99	79.34	51.85	23.30	9.797	4.041	2.237	1.859	2.506
总计	93.58	95.21	58.94	27.05	11.68	4.884	2.56	2.014	2.592

表 4.4: 128*128*128 格点下，跨节点多 GPU 域分解模式解泊松方程各部分所用时间

4.1.3.5 小结

通过上面对于“域分解模式”的实现和测试，可以看出其主要受限于CPU和GPU之间数据拷贝的速度与CPU和CPU之间的通讯速度。如果GPU带宽足够大，或者以后能够实现GPU与GPU之间的直接通讯的话，“域分解模式”可以作为一个可行的方案。但是在目前，与“复制模式”的比较，“域分解模式”由于其额外的数据拷贝和通讯开销，并不能通过减少运算量来达到提高速度的目的。所以我们在程序中依然使用“复制模式”，即使所有的GPU都同时运行同样的程序。

4.1.4 粒子推动

在小节4.1.2粒子权重过程中，粒子是按照区块排列的。因此在推动粒子的时候，直接的方法是像权重过程中一样，由每个线程处理一个区块中的粒子。然而这种方法的缺陷是每个线程上的负载不均匀，可能出现某些区块中粒子数很多从而其线程运算量很大，然而另外某些区块中的粒子数较少，其线程运算量很小的情况。为了保证负载的均衡，我们采取另外一种方法，先把区块中的粒子信息重

组，拷贝到一个典型的 $6 \times N$ 的连续数组中，其中 N 是粒子数，6是维度；然后再并行推动。这种方法虽然需要额外的时间进行数据拷贝，但是相比第一种区块推动的负载更均匀，所以整体时间比区块推动更短，其流程如下所示：

1. 将粒子从区块格式`dev_ray_tile[lth*dim*npm*mth]`重新排列，并拷贝到经典的 $6 \times N$ 数组中`dev_ray[N][6]`。
2. 对`dev_ray[N][6]`进行并行推动。
3. 推动完成后，将粒子拷贝回原区块`dev_ray_tile[lth*dim*npm*mth]`中，以用来下一步的粒子排序和权重插值。

4.2 正确性校验

模拟程序最重要的确保模拟结果的正确性，我们通过将GPU程序的结果与成熟的CPU程序的结果进行比较来验证GPU程序的正确性。图4.9是CPU程序和GPU程序的发射度比较，从图中可以看出，两个程序得到的结果完全相同。通过输出数据得到，两者的区别在十的负十三次方量级，这可能是由于双精度浮点数的精度所限，双精度浮点数的精确度在十的负十四左右。而一般我们输出只取九位有效数字，所以可以认为两个程序的输出结果完全相同，GPU程序的正确性得到了验证。

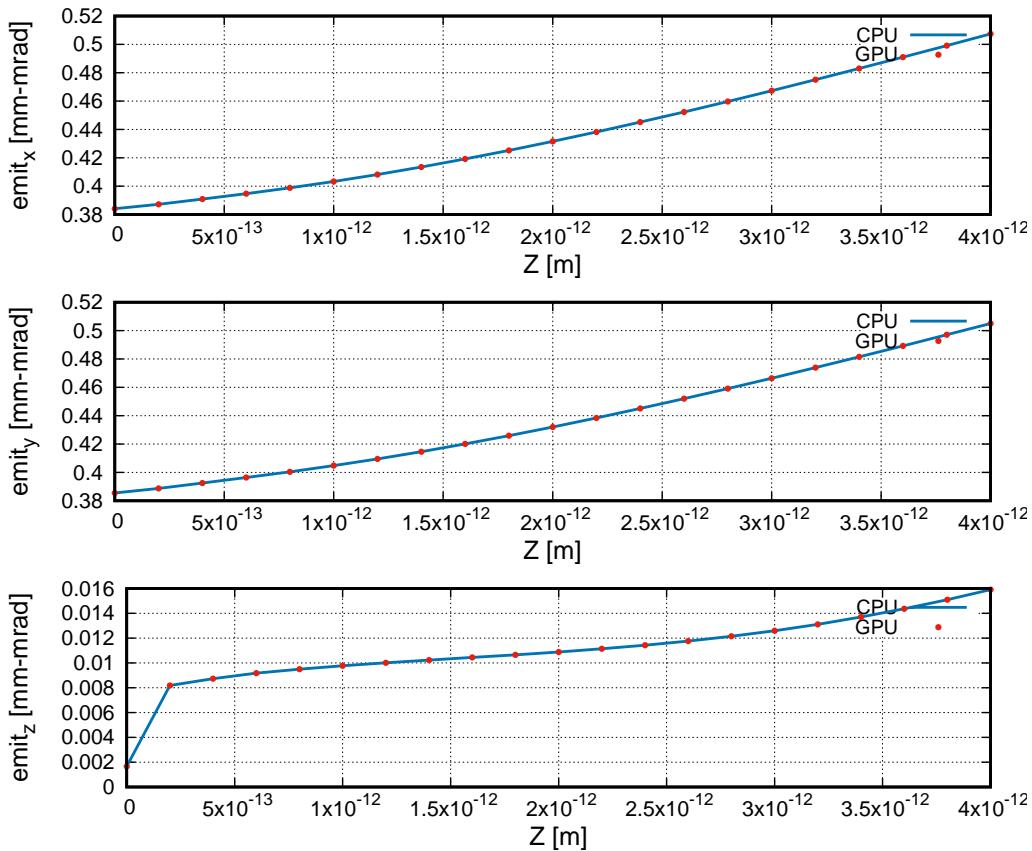


图 4.9: 正确性校验

4.3 性能

4.3.1 单GPU

首先，我们通过对比CPU程序和GPU程序的运行速度，来得到GPU程序的的加速比，加速比等于CPU程序耗时除以GPU程序耗时。测试使用的格点数为 $64*64*64$ ，图4.10和表4.5是粒子数从16k到1.6m时CPU程序和GPU程序各部分耗时以及加速比。程序消耗的总时间的加速比大约有40，然而每一部分的加速比有很大不同。

其中，橙色柱行代表求解泊松方程，其加速比大约是64，而且并不随粒子数目变化而变化。事实上，求解泊松方程的计算量主要和格点数相关，而此次测试中我们都是用同样的格点数，因此其耗时基本不变，之后我们会比较在不同格点数情况下求解泊松方程的加速比变化。灰色柱行代表推动粒子的加速比，其随着粒子数目增加而变大，在粒子数较大时取得超过了70的加速比。浅蓝色和深蓝色柱行分别是权重差值和粒子信息输出的加速比，其中GPU的权重差值部分包括了粒子排序运算，而且由于其运算的不规则性，加速比较低；而粒子信息输出部分也包含了束团参数的计算，这部分加速比较低是因为输出带宽的限制。

16k	CPU(s)	GPU(s)	Speedup
depositor (include sort)	0.16888	0.05874	2.875043
Poisson solver	26.13173	0.42511	61.47051
pusher kicker	0.53748	0.01781	30.17855
output	0.01977	0.01149	1.720627
total loop	27.78216	0.51199	54.26309
64k	CPU(s)	GPU(s)	Speedup
depositor (include sort)	0.3897	0.10847	3.592698
Poisson solver	26.08269	0.41554	62.76818
pusher kicker	2.00422	0.04032	49.70784
output	0.0641	0.01433	4.473133
total loop	29.54123	0.57598	51.28864
160k	CPU(s)	GPU(s)	Speedup
depositor (include sort)	0.82705	0.20731	3.989436
Poisson solver	25.9208	0.40208	64.46677
pusher kicker	4.88644	0.08343	58.56934
output	0.15477	0.02316	6.682642
total loop	32.94187	0.71	46.397
640k	CPU(s)	GPU(s)	Speedup
depositor (include sort)	2.79413	0.71529	3.90629
Poisson solver	25.72129	0.40045	64.23097
pusher kicker	22.48269	0.31207	72.04374
output	0.62289	0.06931	8.987015
total loop	53.51193	1.46745	36.46593
1.6m	CPU(s)	GPU(s)	Speedup
depositor (include sort)	6.7528	1.73779	3.885855
Poisson solver	26.03071	0.39894	65.24969
pusher kicker	56.05512	0.76562	73.21533
output	1.56528	0.16288	9.61002
total loop	90.40391	2.99053	30.23006

表 4.5: PIC程序在单GPU上的加速比

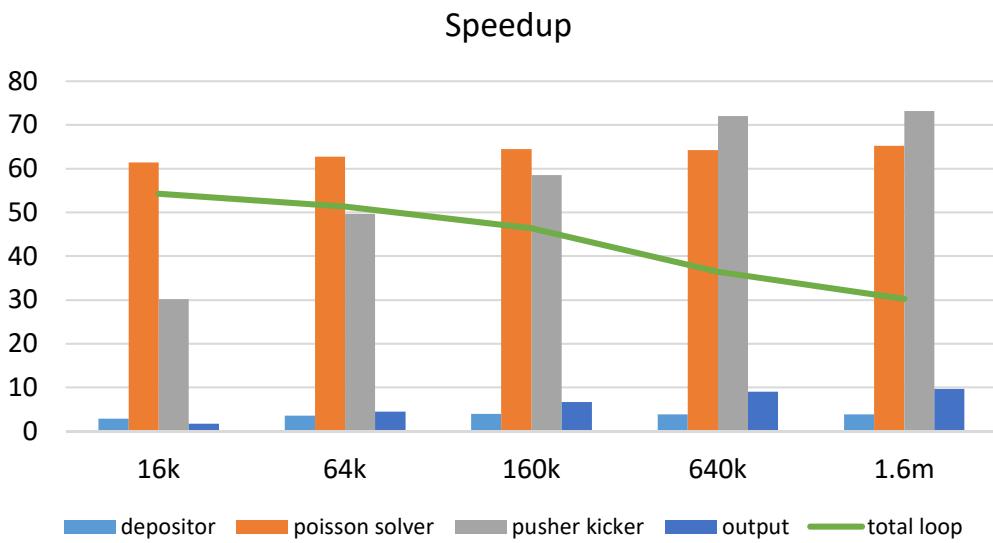


图 4.10: PIC 程序在单 GPU 上的加速比

权重差值和粒子信息输出的的加速比较低，从而拉低了整体的加速比。整体的加速比随着粒子数的增加而逐渐变小，其原因是当粒子数目较小时的时候，求解泊松方程所占得比重较大，从而整体加速比较大；而当粒子数目变大的时候，权重差值所占的时间比重变大，因而整体加速比随之变小。程序各部分耗时在不同粒子数所占比重如图??所示。

在上述测试中使用的都是 $64 \times 64 \times 64$ 个格点数，因此求解泊松方程的时间和加速比都基本不变。在不同的格点数情况下，求解泊松方程的加速比如图4.12所示。可以看出，求解泊松方程的加速比随着格点数的增加逐渐变大，最终达到了接近70，这是因为格点数越大，其计算量越多，GPU上的负载能够分布得更均匀。在实际模拟中常用的 $64 \times 64 \times 64$ 个格点和 $128 \times 128 \times 128$ 个格点中，我们都取得了令人满意的加速比。

总之，在单GPU上，程序总体的加速比达到了40，而求解泊松方程的加速比超过到了60，。在一个普通的家用机GPU上运行的速度是在64核机器上运行的两倍。另外，如小节4.1.1所述，因为GPU内存大小一般是固定的，而不像CPU内存那样较容易的扩展，所以单GPU的粒子数目存在某个上限，当超过最大粒子数时，应该使用多GPU来进行计算。在接下来的一节中，我们将讨论PIC程序在多GPU上的效率。

4.3.2 多GPU

我们使用超级计算机Titan进行多GPU测试。Titan中每个节点只有一个GPU，

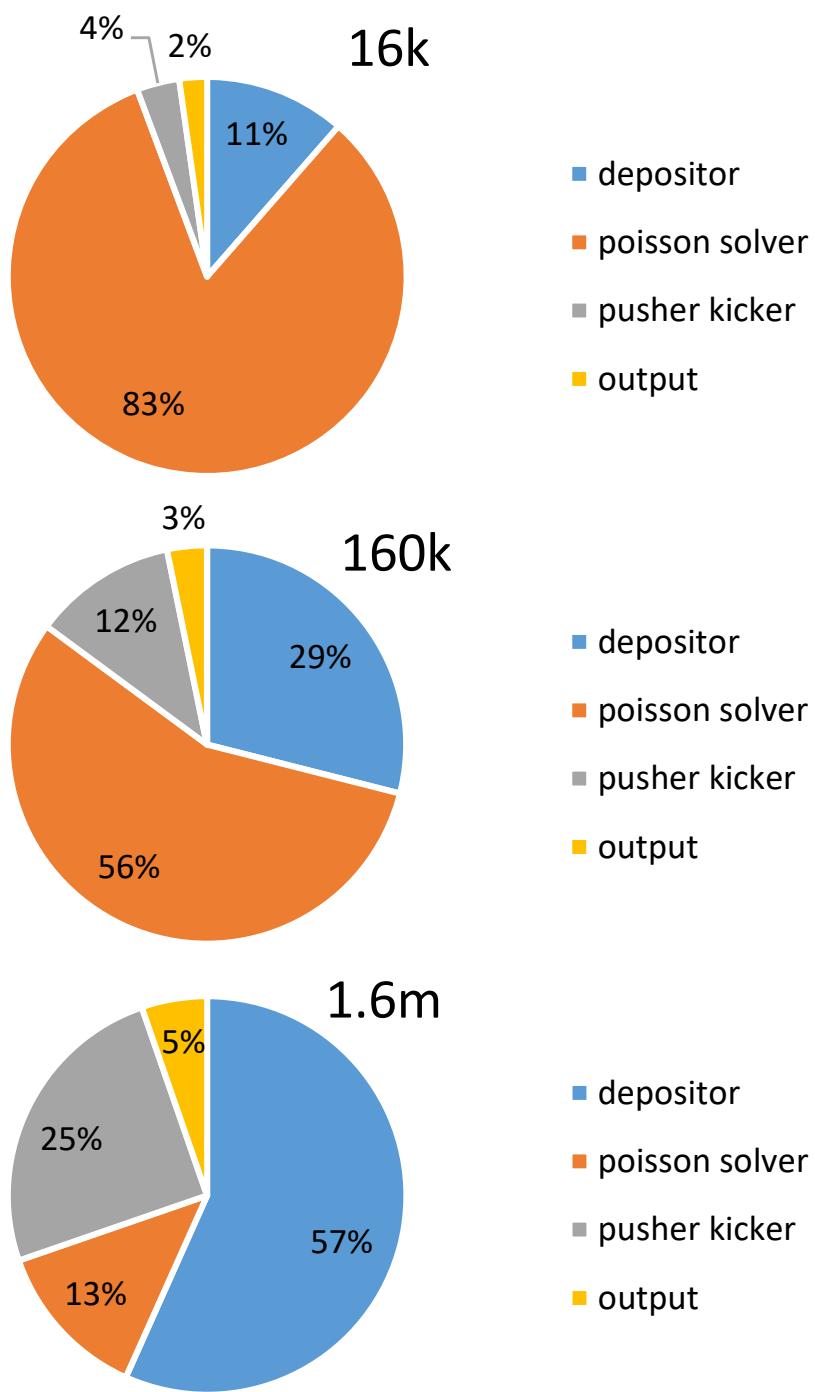


图 4.11: 程序各部分耗时在不同粒子数时所占百分比

型号为NVIDIA K20x，因此只能通过使用多个节点来使用多个GPU。在这种条件下，GPU之间的通讯只能够通过先将数据拷贝到CPU端，再通过CPU端的节点间网络进行通讯。

图4.13为复制模式求解泊松方程时各个部分所花费的时间随GPU个数的变化，可以看出，因为处于复制模式，求解泊松方程的耗时基本保持不变；由于跟多

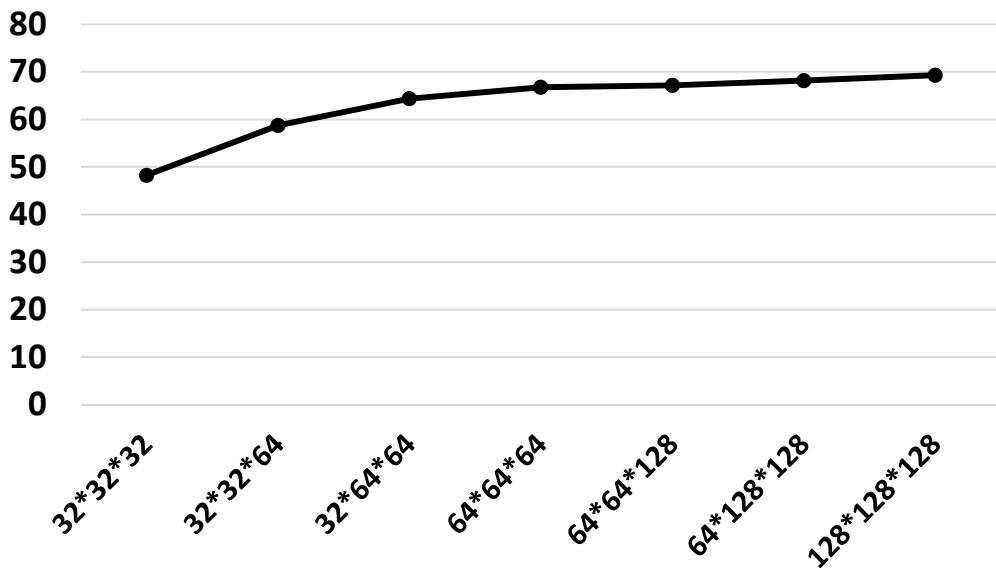


图 4.12: 不同的格点数情况下单GPU求解泊松方程的加速比

的GPU个数意味着每个GPU上的粒子数随之减小，粒子推动的耗时随着GPU数目的增加而减少；而由于通讯成本随着GPU数目增加而增加，信息统计和输出耗时也随之增加。各个部分对于GPU的数目并不相同，总体来看，总耗时基本保持不变。

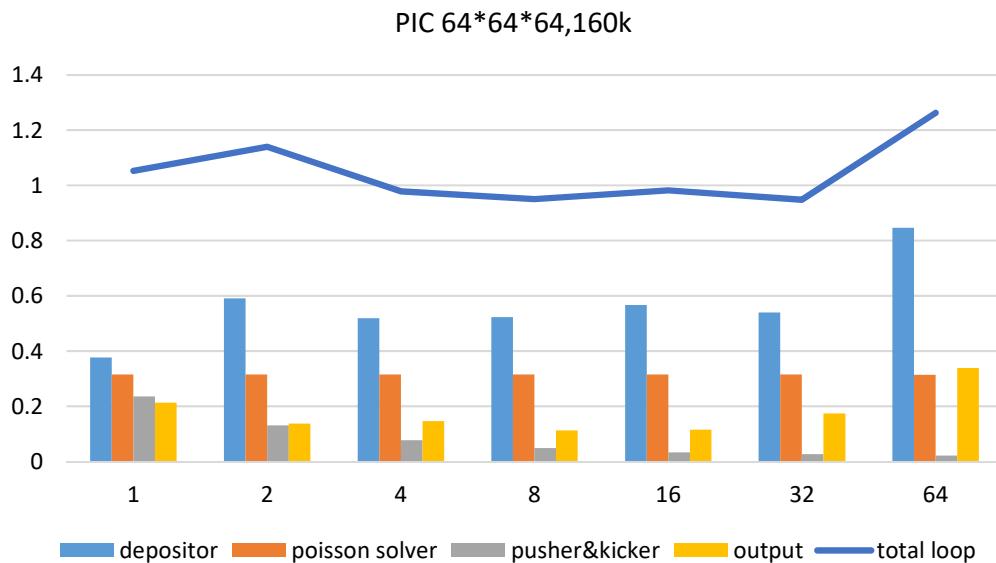


图 4.13: 64*64*64个格点，160k个粒子时，复制模式程序耗时随GPU个数的变化

图4.14和图4.13类似，也是64*64*64各个点，160k个粒子下程序总耗时随着GPU数目变化。但是在图4.14中，GPU数目大于等于2的时候，求解泊松方程使用域分解模式。其在各种情况下，都相较复制模式耗时更多，主要是拷贝数据花费了大

量时间，这个结果和我们在小节4.1.3中讨论的相符。因此在之后的测试中，我们都使用复制模式。

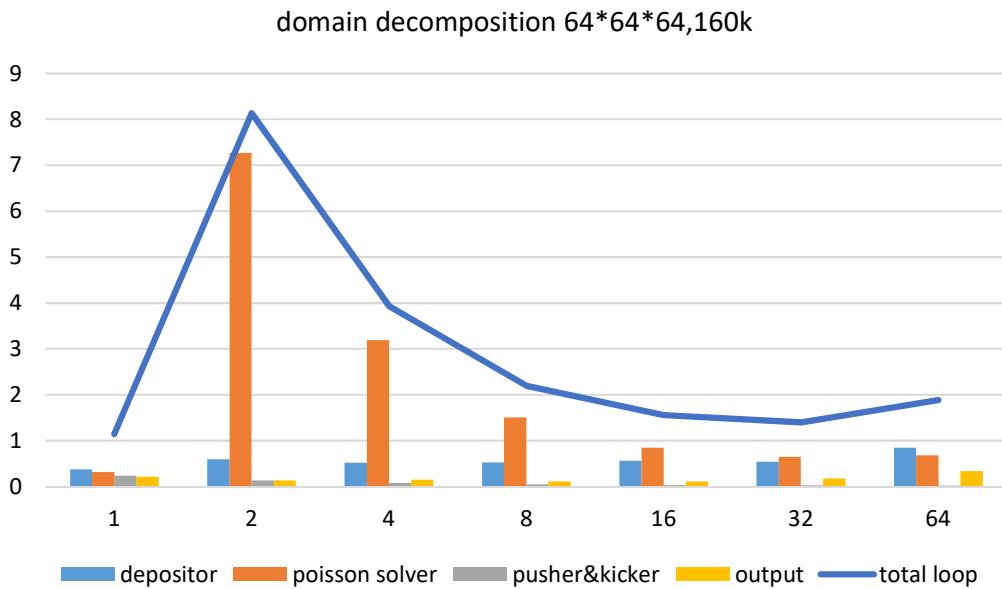


图 4.14: 64*64*64 个格点，160k 个粒子时，域分解模式程序耗时随 GPU 个数的变化

当粒子数更大时，耗时的趋势发生了变化。图4.15是粒子数为一百六十万 (1.6m) 的结果，粒子数较图4.13变大了十倍。在使用1.6m个粒子时，总耗时随着使用的GPU数目增加而明显下降，并在32个GPU处到达最小值。这是因为在大粒子数目情况下，推动粒子和权重差值所占用的时间占了总时间的绝大部分。而由于使用多GPU带来的每个GPU上的粒子数目变少，推动粒子和权重差值能够很好的被多GPU加速。

我们尝试使用更大的粒子数，一千六百万个粒子 (16m)，更十倍于前，如图4.16所示。由于GPU内存大小的限制，程序在这个粒子数下不能仅仅使用1或2个GPU运行，因此在图4.16中，GPU数目等于1和等于2的时候没有数据。

超级计算机Titan上使用的GPU型号为NVIDIA K20x，每个GPU有5GB的内存。理想情况下，一个5GB内存能够使用的最大粒子数为六千万 (60m)。但这个粒子数是在空间均匀分布下才能使用，而实际上，我们很难达到这个粒子数。一个实际的加速器模拟中我们通常使用KV，WaterBag，或者高斯分布，因此其可用的粒子数目远小于理想数目。

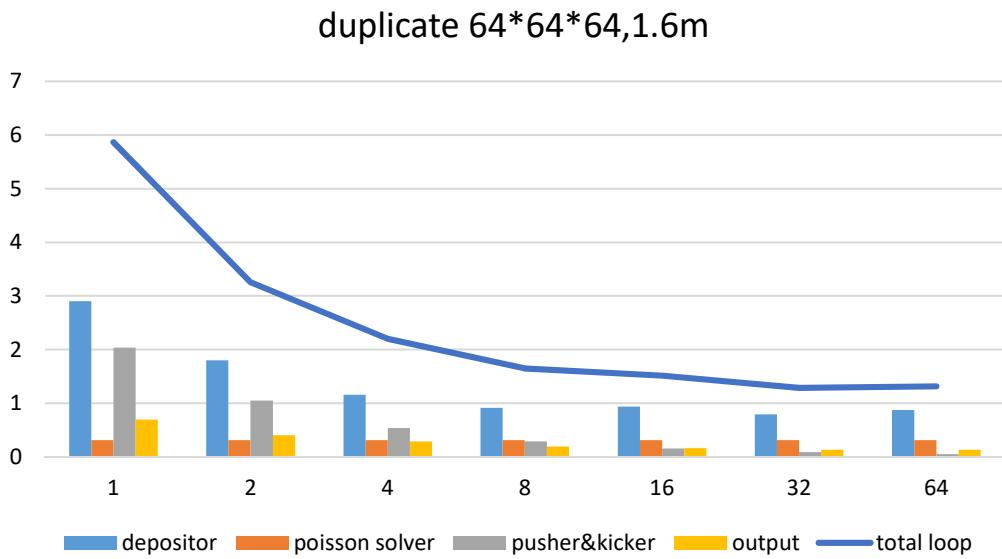


图 4.15: 64*64*64个格点, 1.6m个粒子时, 程序耗时随GPU个数的变化

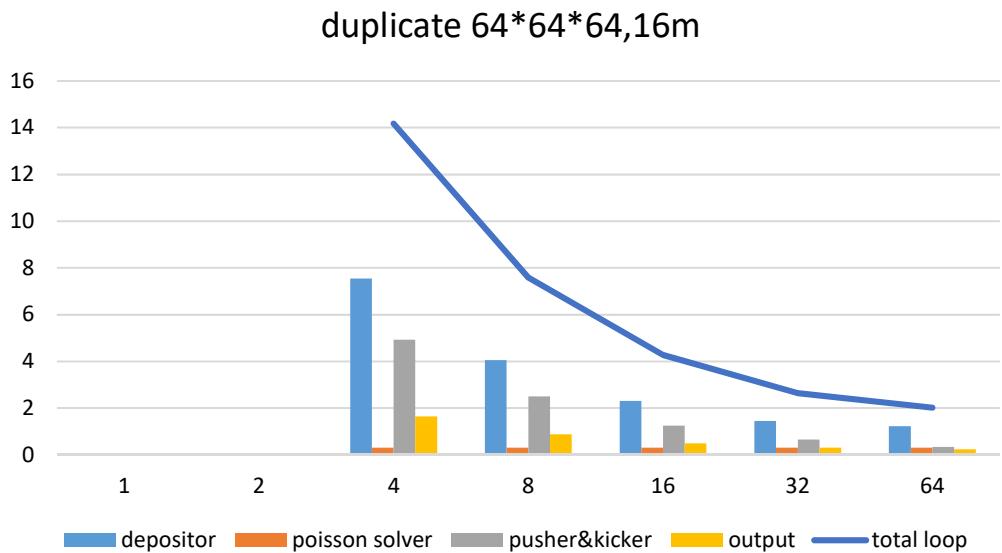


图 4.16: 64*64*64个格点, 16m个粒子时, 程序耗时随GPU个数的变化

4.4 CPU集群

为了和GPU做比较, 我们也在CPU集群上实现了PIC程序, 其使用的计算机为Cori Knight Landing, 使用的是Intel最新的众核处理器, 每个处理器有68个核心。我们的程序使用和MPI和OpenMP混合并行, 以提高运行的效率。

首先, 我们在一个节点上研究不同OpenMP线程数和MPI进程数下程序的效率和内存占用情况, 并找到最优混合并行配置。之后, 我们使用多个节点, 研究程序在不同节点数下的效率变化。

4.4.1 单节点

首先，我们使用 $64*64*64$ 个网格，在 $1.6m$ 粒子数下测试不同的混合并行配置，如图4.17所示。其中横轴为不同的并行配置，MPI进程数由1指数增加到64，而OpenMP线程数有64指数得减小到1，线程数乘以进程数则保持不变。在每种并行配置下，总时间（浅蓝色实线），和权重差值，求解泊松方程，推动粒子，信息输出（各色柱行）所耗时间由左纵轴以秒为单位表示，而内存占用（绿色实线）由右纵轴以GB为单位表示。

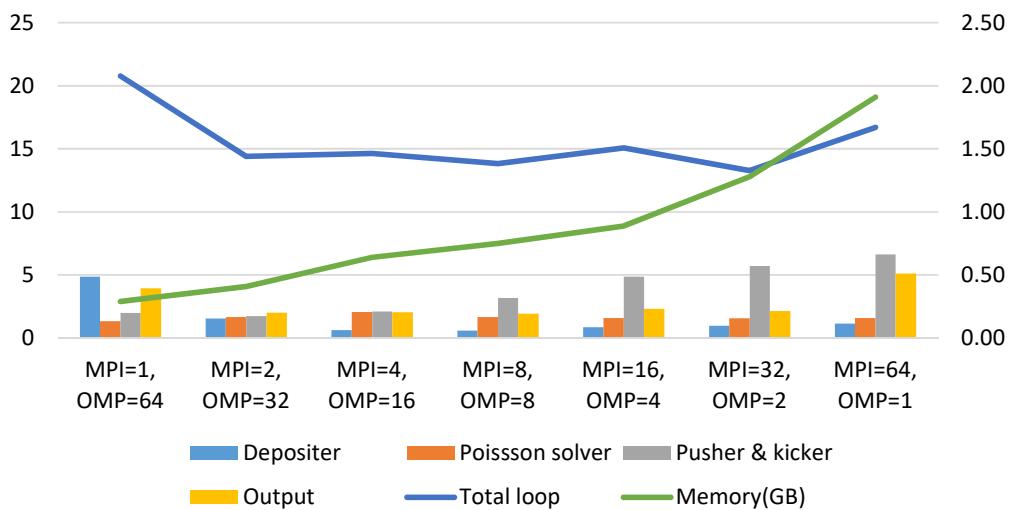


图 4.17: $1.6m$ 粒子数下，不同混合并行配置的耗时与内存占用

从图4.17可以看出，除了纯OpenMP并行（MPI=1，OMP=64）明显较慢外，其他各个并行配置下所消耗的总时间的差别并不大，其中耗时最小的并行配置为使用32个MPI进程，每个进程使用2个OpenMP线程。一般而言，使用较大的MPI进程数是比较有效率的选择。而内存占用基本随着MPI进程数目线性增加。

程序的不同部分对于并行配置的反应并不相同。随着MPI进程数的增加和每个进程所用线程数的减小，权重差值的耗时先减小后增加，一开始先从MPI=64, OMP=1时的1.14秒减少到了MPI = 8, OMP = 8时的0.58秒，然后其开始剧烈增加，最终到达MPI = 1, OMP = 64时的4.85秒。其原因是权重差值需要在不同的线程之间进行归约以避免线程冲突，而归约操作在线程数很大时有一个较大的启动时间。粒子推动耗时随着MPI数变大单调增加，这是因为OpenMP更适合Knight Landing众核架构，并能更有效的利用矢量处理器。

我们也测试了不同的粒子数下的并行配置，测试的粒子数为 $160k$ 和 $16m$ ，分别是之前粒子数的十分之一和十倍，其结果如图4.18和图4.19所示。在 $160k$ 个粒子的情况下，总时间随着MPI进程数的增加单调减少，这表明小粒子数情况更适合

纯MPI并行。而在1.6m个粒子的情况下，纯OpenMP并行由于无MPI进程间通讯，显示出一些优势，但是其总耗时依然大于MPI并行，其耗时最小的配置为MPI=64, OMP=1。

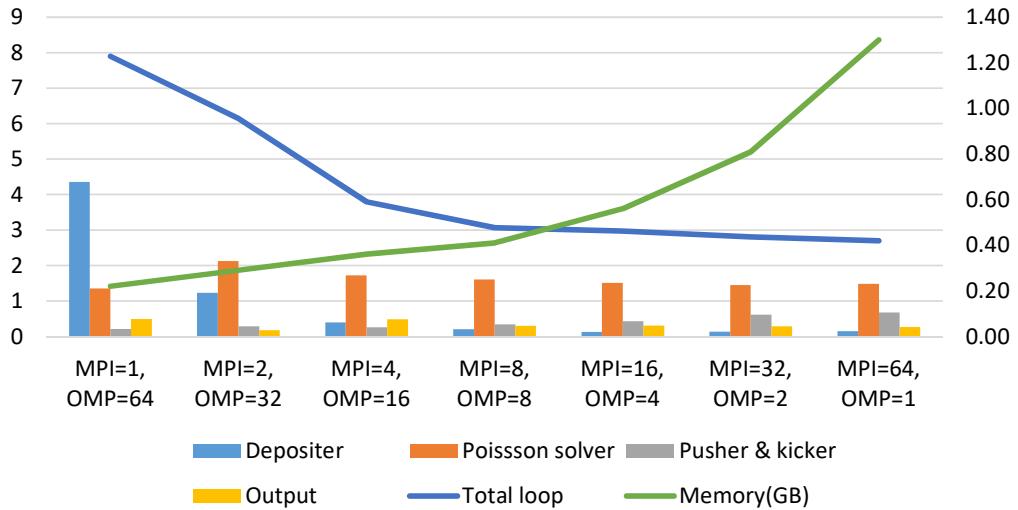


图 4.18: 160k 粒子数下，不同混合并行配置的耗时与内存占用

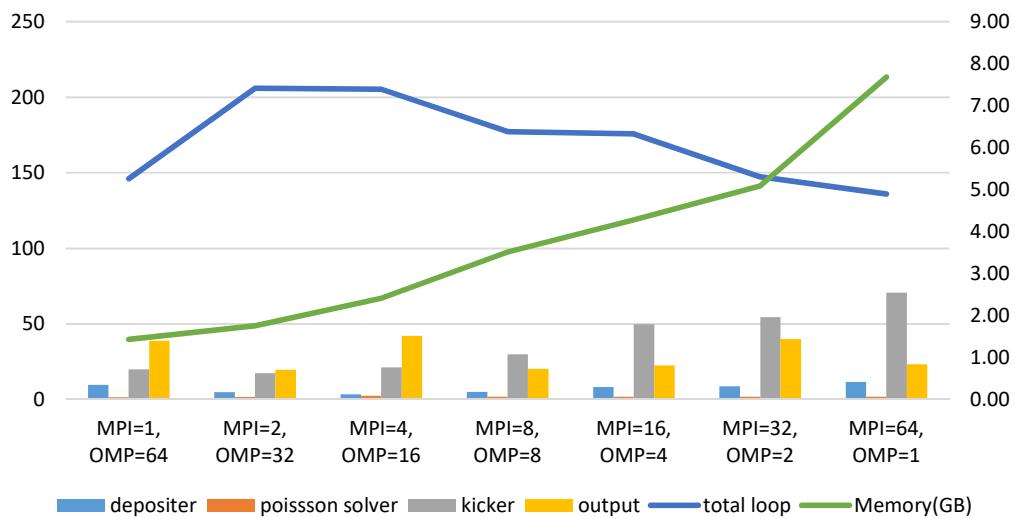


图 4.19: 16m 粒子数下，不同混合并行配置的耗时与内存占用

在各种情况下，无论粒子数多少，内存占用总是随着MPI进程数变大而单调增加。

4.4.2 多节点

从上面单节点的测试中可以得到，使用较大的MPI进程数和较小的OpenMP线程数是更有效率的并行配置。因此在多节点的测试中，我们首先选用纯MPI并行，

测试在不同的节点数下程序总体以及各个部分的耗时情况。之后，我们也测试了OMP=2和OMP=4的情况，并与纯MPI程序进行了比较。

图4.20是纯MPI配置下PIC程序在CPU集群多节点下的耗时情况。每个节点我们使用64个核，图4.20的横轴为节点数目；左纵轴为时间，以秒为单位；右纵轴为内存使用，以GB为单位。随着使用更多的节点，程序总耗时先降低后增加，在32个节点处到达最小值。总耗时先减小是因为使用的节点数越多，每个节点上需要进行的运算越少；后增加是因为随着节点数上升，节点间的通讯时间也会随之增加。图4.21为使用64个节点时程序各个部分消耗时间所占的百分比，可以看出此时通讯耗时已经占了总耗时的60%。

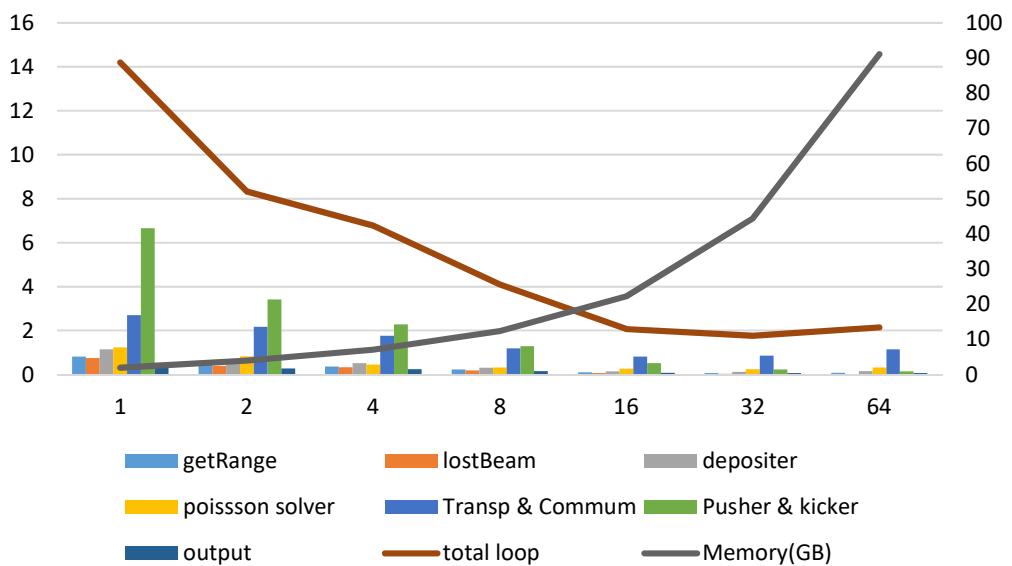


图 4.20: PIC 程序使用多个 CPU 节点的耗时

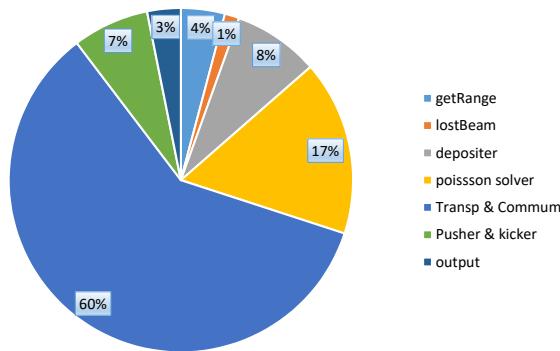
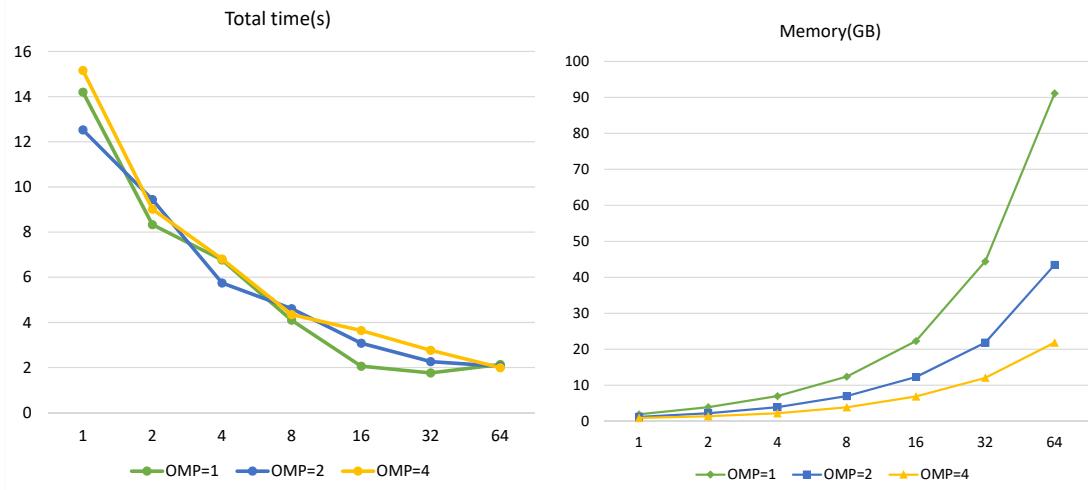


图 4.21: 使用 64 个节点时程序各个部分消耗时间所占的百分比

接下来，我们对OMP=2和OMP=4的并行配置进行了测试，并与纯MPI程序(OMP=1)进行了比较，如图4.22所示。在大部分情况下，不同并行的耗时差别很

小；但是在某些情况下差别很大，比如节点数为16时，纯MPI的速度是OMP=4的1.8倍。对于内存使用情况，使用更多的OpenMP线程和更少的MPI进程总是占优势。综合考虑，在内存足够大的情况下，在Knight Landing上运行的PIC程序使用纯MPI并行配置依然是一个很好的选择。



(a) 不同并行配置下的多节点运行耗时比较 (b) 不同并行配置下的多节点运行内存比较

图 4.22: 不同并行配置下的多节点运行情况比较

4.4.3 与GPU对比

在本章之前的测试中，我们使用的GPU型号为Nvidia GeForce GTX 1060，其包含1280颗核心，时钟频率为1.71GHz。而我们使用的CPU型号为Intel® Xeon Phi™ Processor 7250 ("Knights Landing")，共包含68颗核心，时钟频率为1.40GHz，但是实际上，我们只是使用了64颗核心。

而由于GPU程序做了一些简化，缺少一部分步骤，例如粒子丢失判据等，我们在进行比较的时候也在CPU程序的耗时中减去了相应部分。在64*64*64个格点数，1.6m个粒子的情况下，对于同样长度的加速器，使用一个GPU代码的总耗时为3.56秒，这类似于CPU程序在300个核心上的运行时间。换而言之，对于我们的PIC程序，一个GPU卡的运算效率与4到8个CPU节点的运行效率相当。

4.5 小节

我们在GPU上使用使用CUDA库实现了基于PIC方法的多粒子模拟程序，并对如何避免竞争条件和实现更高性能的GPU代码结构和并行策略进行了介绍。在单

个GPU卡上，我们使用普通的家用GPU（GTX 1060）实现了超过50倍的加速。当粒子数较大时，程序在GPU集群上也显示出良好的可扩展性；而当粒子数目较小时其可扩展性较差。我们也新的CPU架构Cori Knight Landing上实现了PIC程序，并探索了其最佳性能的并行线程配置。通过比较，单GPU运行的程序和使用4或8个节点的程序性能相当。在未来的研究中，我们将继续开发此代码并提高效率。

第五章 Symplectic算法在GPU上的实现

数值模拟在束流动力学研究和加速器设计中非常重要。目前，主流的模拟软件都是用质点网格法（PIC）求解空间电荷效应[27–33]。PIC算法是求解空间电荷效应非常快速有效的方法，在PIC算法中，粒子首先被根据位置权重到网格上，然后根据网格上的电荷密度求解泊松方程，得到网格上的电势，从而得到网格上的电场，再根据粒子位置反推出粒子所处位置的电场。使用这种方法，计算复杂度由直接的粒子-粒子方法的 N_p^2 降低到了 $\alpha N_p + \beta N_{cells} \log N_{cells}$ 。其中 N_p 是粒子数，而 N_{cells} 是网格点数目。

然而，PIC算法需要将粒子权重到网格上，不可避免的会带来网格热噪声，因此PIC算法是否可以保障辛条件在目前仍然有较大的争议。如果不能报障辛条件，那么计算就会被引入一些数值算法带来的非物理的效应。这些效应对最终的强流束流物理的分析和讨论会带来一些干扰。因此，在需要长程模拟（比如环形加速器）的物理分析中，我们需要一种保辛的算法。

最初，保辛算法本身是为了保障哈密顿系统中的辛条件而被研究 [34, 35]。后来，无网格保辛多粒子追踪模型被引入到加速器研究和模拟中，作为在长距离模拟中空间电荷求解器 [36]。这种模型并不利用网格，而是利用高阶分解来求解空间电荷效应。相比PIC算法，这种方法能够显著的降低由于网格数值噪声带来的发射度增长。然而，虽然能够保证辛条件，这种算法也有缺陷，最显著的就是计算量要大得多。无网格算法的计算复杂度为 $\alpha N_p * N_{modes}$ ，其中 N_{modes} 为分解的阶数，在通常模拟中我们一般使用 $16 \times 16 \times 16$ 阶，在这种配置下花费的时间比PIC算法要高两到三个数量级。所以我们必须提高这个算法的运行速度，以提高算法的实用性。

幸运的是，无网格算法很适合并行，有很好的可扩展性。这种特性非常适合使用GPU进行加速计算。通过使用CUDA和GPU，可以显著加快无网格粒子跟踪代码的运行速度。当运行在单个GTX 1060时，与CPU串行相比，GPU代码实现了超过400倍的加速。此外，加速比会随着GPU的数目线性增长。

在接下来的这一章中，我们将会首先在小节5.1 中介绍保辛空间电荷算法。接下来我们会在小节5.2中介绍算法在GPU上的代码结构和优化。在小节 5.3中，我们会讨论代码的加速比. 在小节5.4中，我们展示了使用这个代码的一个应用实例。

5.1 Symplectic算法

在束流动力学模拟中，当且仅当雅可比矩阵 M_i 满足以下条件时，传输矩阵 m_i 才是保辛的[37, 38]:

$$M_i^T J M_i = J \quad (5.1)$$

其中， J 是如下 $6N \times 6N$ 的矩阵:

$$J = \begin{pmatrix} 0 & I \\ -I & 0 \end{pmatrix} \quad (5.2)$$

对于考虑其空间电荷力的多粒子系统，哈密尔顿量可以写为:

$$H = H_1 + H_2 \quad (5.3)$$

其中

$$H_1 = \sum_i p_i^2/2 + \sum_i q\psi(r_i) \quad (5.4)$$

$$H_2 = \frac{1}{2} \sum_i \sum_j q\varphi(r_i, r_j) \quad (5.5)$$

其中 H_1 只包含外场信息，而 H_2 只包括空间电荷效应。根据 H_1 和 H_2 得到的两个传输矩阵 m_1 和 m_2 ，一个二阶传输矩阵 $m(\tau)$ 可以被定义为:

$$m(\tau) = m_1(\tau/2) m_2(\tau) m_1(\tau/2) \quad (5.6)$$

如果 m_1 和 m_2 都是保辛的，那么 m 就是保辛的。在大多数加速器元件中，我们可以通过单粒子动力学获得相应的外场传输矩阵 m_1 ，而内场传输矩阵 m_2 可以写为:

$$r_i(\tau) = r_i(0) \quad (5.7)$$

$$p_i(\tau) = p_i(0) - \frac{\partial H_2(r)}{\partial r_i} \tau \quad (5.8)$$

其雅克比矩阵为:

$$M_2 = \begin{pmatrix} I & 0 \\ L & I \end{pmatrix} \quad (5.9)$$

其中 $L_{ij} = \frac{\partial p_i(\tau)}{\partial r_j} = -\frac{\partial^2 H_2(r)}{\partial r_i \partial r_j} \tau$ 是一个对称矩阵，所以 M_2 满足保辛条件.

对于一个3D束团， H_2 可以表示为:

$$H_2 = \kappa\gamma_0 \sum_i \sum_j \varphi(r_i, r_j) \quad (5.10)$$

其中 $\kappa = q/(lmC^2\gamma_0^2\beta_0)$, $l = C/\omega$ 。而在束流坐标系下的静电势可以有泊松方程得到:

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} + \frac{\partial^2 \phi}{\partial z^2} = -\frac{\rho}{\epsilon_0} \quad (5.11)$$

其中边界条件为:

$$\begin{aligned} \phi(x=0, y, z) &= 0, & \phi(x=a, y, z) &= 0 \\ \phi(x, y=0, z) &= 0, & \phi(x, y=b, z) &= 0 \\ \phi(x, y, z=0) &= 0, & \phi(x, y, z=c) &= 0 \end{aligned} \quad (5.12)$$

其中, a, b, c 分别是X, Y, Z方向上的零电势边界长度。如果我们将c设的足够大, 电势将在无穷远处为零。

泊松方程中的电势 ϕ 和电荷密度 ρ 可以展开为:

$$\rho(x, y, z) = \sum_{l=1}^{N_l} \sum_{m=1}^{N_m} \sum_{n=1}^{N_n} \rho^{lmn} \sin(\alpha_l x) \sin(\beta_m y) \sin(\gamma_n z) \quad (5.13)$$

$$\phi(x, y, z) = \sum_{l=1}^{N_l} \sum_{m=1}^{N_m} \sum_{n=1}^{N_n} \phi^{lmn} \sin(\alpha_l x) \sin(\beta_m y) \sin(\gamma_n z) \quad (5.14)$$

其中 N_l, N_m, N_n 分别为电势和电荷密度在X, Y, Z三个方向上展开的阶数, 而 ρ^{lmn} 和 ϕ^{lmn} 可以表达为:

$$\rho^{lmn} = \frac{8}{abc} \int_0^a \int_0^b \int_0^b \rho(x, y, z) \sin(\alpha_l x) \sin(\beta_m y) \sin(\gamma_n z) dx dy dz \quad (5.15)$$

$$\phi^{lmn} = \frac{8}{abc} \int_0^a \int_0^b \int_0^b \phi(x, y, z) \sin(\alpha_l x) \sin(\beta_m y) \sin(\gamma_n z) dx dy dz \quad (5.16)$$

其中

$$\alpha_l = \frac{l\pi}{a}, \beta_m = \frac{m\pi}{b}, \gamma_n = \frac{n\pi}{c} \quad (5.17)$$

将上面的展开代入泊松方程, 我们可以得到:

$$\phi^{lmn} = \frac{\rho^{lmn}}{\epsilon_0(\alpha_l^2 + \beta_m^2 + \gamma_n^2)} \quad (5.18)$$

据此, 我们得到了粒子密度 rho 和电势 phi 之间的关系:

$$\begin{aligned} \phi(x, y, z) &= \frac{1}{\epsilon_0 abc} \omega \times \\ &\sum_{j=1}^{N_j} \sum_{l=1}^{N_l} \sum_{m=1}^{N_m} \sum_{n=1}^{N_n} \frac{\sin(\alpha_l x_j) \sin(\beta_m y_j) \sin(\gamma_n z_j) \sin(\alpha_l x) \sin(\beta_m y) \sin(\gamma_n z)}{(\alpha_l^2 + \beta_m^2 + \gamma_n^2)} \end{aligned} \quad (5.19)$$

其中 ω 是粒子的电荷。

于是哈密顿量 H_2 可以表示为:

$$H_2 = \frac{1}{\varepsilon_0} \frac{8}{abc} \omega \kappa \gamma_0 \times \\ \sum_{i=1}^{N_i} \sum_{j=1}^{N_j} \sum_{l=1}^{N_l} \sum_{m=1}^{N_m} \sum_{n=1}^{N_n} \frac{\sin(\alpha_l x_j) \sin(\beta_m y_j) \sin(\gamma_n z_j) \sin(\alpha_l x_i) \sin(\beta_m y_i) \sin(\gamma_n z_i)}{(\alpha_l^2 + \beta_m^2 + \gamma_n^2)} \quad (5.20)$$

我们得到了保辛的空间电荷传输矩阵 m_2 。以X方向为例:

$$x_i(\tau) = x_i(0) \quad (5.21)$$

$$p_{xi}(\tau) = p_{xi}(0) - \tau \frac{1}{\varepsilon_0} \frac{8}{abc} \omega \kappa \gamma_0 \times \\ \sum_{j=1}^{N_j} \sum_{l=1}^{N_l} \sum_{m=1}^{N_m} \sum_{n=1}^{N_n} \frac{\alpha_l \sin(\alpha_l x_j) \sin(\beta_m y_j) \sin(\gamma_n z_j) \cos(\alpha_l x_i) \sin(\beta_m y_i) \sin(\gamma_n z_i)}{(\alpha_l^2 + \beta_m^2 + \gamma_n^2)} \quad (5.22)$$

在Y和Z方向的传输矩阵与之类似。

5.2 GPU程序结构

在大多数加速元件中，粒子会被推动若干步。在每一步中，粒子将首先被外场传输矩阵传输推动半个时间步长，然后由空间电荷推动一个时间步长，并再次被外场传输矩阵传输推动半个时间步长。其中，求解空间电荷将消耗整个计算时间的90%以上。计算空间电荷效应的程序由下面的三个子程序组成，而每个子程序由一个或两个内核组成。

1. 遍历三角函数
2. 计算 Φ^{lmn}
3. 计算 ep_i 并推动粒子

我们对程序进行了许多优化，使其更适合GPU架构，程序的性能得到显着改善。每个子程序的优化策略各不相同，下面介绍各个内核的详细优化策略。

5.2.1 遍历三角函数

根据粒子的位置，我们首先计算每个粒子的三角函数临时变量。设：

$$\begin{aligned} S_j^l &= \sin(\alpha_l x_j), & C_j^l &= \cos(\alpha_l x_j) \\ S_j^m &= \sin(\alpha_m x_j), & C_j^m &= \cos(\alpha_m x_j) \\ S_j^n &= \sin(\alpha_n x_j), & C_j^n &= \cos(\alpha_n x_j) \end{aligned} \quad (5.23)$$

其中下标 j 是指不同的粒子，下标 l, m 和 n 是指三个方向的频谱模式。

然后，X方向上的传输矩阵 m_2 （等式 ref eq: map1）可以表示为：

$$p_{xi}(\tau) = p_{xi}(0) - \tau \frac{1}{\varepsilon_0 abc} \omega \kappa \gamma_0 \sum_{j=1}^{N_j} \sum_{l=1}^{N_l} \sum_{m=1}^{N_m} \sum_{n=1}^{N_n} \frac{\alpha_l S_j^l S_j^m S_j^n C_i^l S_i^m S_i^n}{(\alpha_l^2 + \beta_m^2 + \gamma_n^2)} \quad (5.24)$$

与等式5.22相比，新的传输矩阵节省了很大的计算量。计算 S_j 和 C_j 是非常有必要的，这样可以避免后期处理的重复计算。在这个子程序中，我们采用按照粒子分配线程的策略，即每个线程处理一个粒子，这段程序结构相对简单，只占整个空间电荷效应求解所需时间的2%左右。然而，它生成了 $2 \times (N_l + N_m + N_n) \times N_j$ 的数据，占了大部分的内存空间，而GPU的内存大小在给定型号的情况下是一个固定值，并不能通过添加内存条来增加，在这个意义上，这段程序决定了模拟中粒子数目的最大值，是花费存储空间以节省计算量的典型示例。根据我们的测试，考虑到不可避免的内存碎片，假设展开阶数为 $64 \times 64 \times 64$ ，程序可以在GeForce GTX 1060 6GB上处理大约100万个粒子。

由于需要遍历内存，该子程序的速度受到全局内存带宽的限制。为了改进内存读写，粒子数据是以阵列结构（SoA）排列的形式而不是结构数组（AoS），以达到对齐读取（coalesced memory read）。并且，在CPU侧分配页锁存存储器，以实现CPU与GPU之间的数据复制速度更快。

5.2.2 计算 Φ^{lmn}

我们注意到在传输矩阵5.24中，对下标 j 的求和是对每个粒子的求和，我们可以改变求和的顺序以节省计算量。如果我们定义：

$$\Phi^{lmn} \equiv \sum_{j=1}^{N_j} S_j^l S_j^m S_j^n \quad (5.25)$$

如果首先完成 $N_l \times N_m \times N_n$ 个 Φ^{lmn} 的计算。传输矩阵5.24可以重写为：

$$p_{xi}(\tau) = p_{xi}(0) - \tau \frac{1}{\varepsilon_0 abc} \omega \kappa \gamma_0 \sum_{l=1}^{N_l} \sum_{m=1}^{N_m} \sum_{n=1}^{N_n} \frac{\Phi^{lmn} \alpha_l C_i^l S_i^m S_i^n}{(\alpha_l^2 + \beta_m^2 + \gamma_n^2)} \quad (5.26)$$

以这种方式，计算复杂度从 $\alpha N_p^2 * N_{modes}$ 减少到了 $\alpha N_p * N_{modes}$ ，这使这个保辛粒子跟踪算法的计算量大大降低。

该子程序的目的是为每个展开模计算 Φ^{lmn} ，自然地我们使用每个线程处理一个模的方式。然而，在通常情况下，我们使用 $16 \times 16 \times 16$ 模式就可以得到收敛的

结果，也就是说会有 $16 \times 16 \times 16 = 4096$ 个线程，而GPU通常具有几百甚至几千个核心，这个线程数量相对于的GPU的核心数目来说较少。为了实现负载平衡，我们将粒子分为几个部分，并采用CUDA流技术来获得高并发性。我们定义临时变量 $\Phi_{temp,i}^{lmn}$ ：

$$\Phi_{temp,i}^{lmn} \equiv \sum_{j=Nstart_i}^{Nend_i} S_j^l S_j^m S_j^n \quad (5.27)$$

那么， Φ^{lmn} 由以下求和得到：

$$\Phi^{lmn} = \sum_i \Phi_{temp,i}^{lmn} \quad (5.28)$$

这个子程序的速度也是受到内存带宽的限制。我们在计算 Phi^{lmn} 之前，会首先对 S_j 进行转置以达到对齐读取。

5.2.3 计算 ep_i 并推动粒子

以x方向为例，我们定义：

$$ep_{xi} \equiv \tau \frac{1}{\varepsilon_0} \frac{8}{abc} \omega \kappa \gamma_0 \sum_{l=1}^{N_l} \sum_{m=1}^{N_m} \sum_{n=1}^{N_n} \frac{\Phi^{lmn} \alpha_l C_i^l S_i^m S_i^n}{(\alpha_l^2 + \beta_m^2 + \gamma_n^2)} \quad (5.29)$$

我们首先根据前两个小节中得到的 Φ^{lmn} ， S_j 和 C_j 计算得到 ep_{xi} ，随后，使用 ep_{xi} 对粒子进行推动。传输矩阵5.26可以重新写为一个简洁的形式：

$$p_{xi}(\tau) = p_{xi}(0) - ep_{xi} \quad (5.30)$$

由于GPU寄存器数量的限制， ep_i 的计算和粒子的推动在X，Y，Z三个方向上分别进行，以提高GPU占用率和运算效率。在这个子程序中，内层最内层循环会以不同的顺序访问 Phi^{lmn} ，所以在调用此子例程，我们需要对 Phi^{lmn} 进行转置，以实现对齐读取，提高效率。

在个方向上，我们根据粒子来分配线程。这段程序受到GPU常量内存大小和GPU共享内存大小的限制有两个分支：一个是在阶数小于 $20 \times 20 \times 20$ 的情况下，另一种是阶数大于 $20 \times 20 \times 20$ 的情况下。

5.2.3.1 分支1：阶数小于 $20 \times 20 \times 20$

当阶数小于 $20 \times 20 \times 20$ 时，我们使用常量内存保存 Phi^{lmn} 。常量内存是专门针对广播进行优化的，即多个线程同时访问同一常量内存地址的速度很快，正适合

需要被每个线程访问的 Φ^{lmn} 。一个普通GPU中的常量内存总量为65536字节，只能容纳8192个双精度浮点数。正是这个常量内存的大小，决定了小于 $20 \times 20 \times 20$ 和大于 $20 \times 20 \times 20$ 两个分支的分界点。

在这个分支中，每个方向的内核使用共享内存存储最内部的循环中 S_j 和 C_j 。共享内存是一种片上内存，内存大小较小，每个GPU的每个流处理器只有64kb，但是共享内存的速度远远快于全局内存。由共享内存大小限制，这段程序的GPU占用率仅为25%，但是共享内存的访问延迟要比全局内存低大约100倍。

还进行了一次测试来评估使用全局内存而不是共享内存的速度。在这个测试中，我们通过数据结构的设计来使让warp中的所有线程访问连续的内存地址，尽量避免全局内存的访问延迟。由于不再受共享内存大小的限制，GPU占用率可以达到接近100%。然而，由于全局内存访问过于频繁，程序运行花费的时间接近使用共享内存程序的两倍。

5.2.3.2 分支1：阶数大于 $20 \times 20 \times 20$

当阶数大于 $20 \times 20 \times 20$ 时，受限于共享内存和常量内存的大小，上述直接计算 ep_i 并推动粒子的方式将无法有效工作，我们必须进行改变。

首先， Φ^{lmn} 被存储在全局内存中。在这段程序中多个线程会访问相同内存地址，而由于GPU全局内存的合并读取机制，使用全局内存的速度只比使用常量内存的速度慢10%。另外，受限于共享内存的大小，我们将 ep_i 的计算和推送粒子分开。而在 ep_i 的计算中，不同的阶被分为若干部分来，分别使用共享内存。它类似于小节 5.2.2 中对 Φ^{lmn} 的计算。每个粒子会使用若干线程，而每个线程处理相应的阶，并获得临时变量 $ep_i^{temp,j}$ 。

$$ep_i^{temp,j} \equiv \tau \frac{1}{\varepsilon_0} \frac{8}{abc} \omega \kappa \gamma_0 \sum_{l=1}^{N_l} \sum_{m=1}^{N_m} \sum_{n=N_{start_j}}^{N_{end_j}} \frac{\Phi^{lmn} \alpha_l C_i^l S_i^m S_i^n}{(\alpha_l^2 + \beta_m^2 + \gamma_n^2)} \quad (5.31)$$

然后我们队 $ep_i^{temp,j}$ 求和并根据等式5.30推动粒子：

$$ep_i = \sum_j ep_i^{temp,j} \quad (5.32)$$

分成若干部分分别计算也有相应的缺点，分开计算需要更多的内存空间来保存 $ep_i^{temp,j}$ 。额外的内存使用量与粒子数和阶数成正比。在相同的内存大小分别计算的最大粒子数会比直接计算的减少约为20%。

5.3 性能

我们使用GPU程序的性能和可拓展性进行了两个测试，第一个测试使用一个普通的家用GPU：GTX 1060 6GB，测试的结果与在CPU串行运行的程序进行比较。第二个测试使用美国能源部下属的橡树岭国家实验室（ORNL）的GPU集群泰坦，用于测试程序的可扩展性，即程序在多GPU下的表现。

5.3.1 单GPU性能提升-GTX1060

首先，我们将GPU代码的性能与CPU代码进行比较。GPU代码使用GeForce GTX 1060 6GB（Pascal架构）进行测试，而作为对比的CPU代码使用AMD Opteron (tm) 6134的一个核心运行，测试的软件环境为Ubuntu 16.04，使用CUDA 8.0版。

加速比由CPU版本运行的运行时间，除以GPU版本的运行时间得到。在本次测试中，我们对空间电荷求解器和整个程序分别进行了比较。空间电荷求解器包括将数据从CPU侧复制到GPU侧，计算空间电荷效应，推动粒子，并将数据复制回CPU侧，而整个程序则包括了除了空间电荷求解器之外的所有其他部分，比如外场推动，输入，统计，输出等等。

从CPU代码简单的移植到GPU代码就可以实现大约200倍的加速，但是我们通过优化可以获得更大的加速比。在采用了小节5.2中的优化策略后，程序能够充分利用GPU，取得了超过400的加速比。接下来，如图5.1所示，我们会讨论分别讨论空间电荷效应和整个程序在不同的问题规模大小下的加速比。

图5.1a为优化后的空间电荷效应求解器的加速比在不同的问题规模下的变化。其中，横坐标为展开的阶数，而纵坐标是加速比，不同的线是不同粒子数目的结果。可以看到，阶数越大加速比越高，这与我们的预期符合，因为阶数越大意味着空间电荷求解器占用的时间在总时间中的比例越大。其中，在 $8 * 8 * 8$ 阶时加速比很低，因为在这种模式下的计算量很低。随着阶数的增加，计算量也随之增加，GPU的负载更为平衡，因此取得了更大的加速比。另一方面，粒子数目对于加速比的影响很小，在大阶数的情况下尤为如此。这是因为粒子数目本身远远超出了一个GPU的核心数目（在GTX 1060 上为1280个核心），即时在最低粒子数（10000个粒子）的情况下，程序也能有效的使用所有的核心，而随着粒子数目的轻微提升是因为GPU能够在更大运算量的情况下更好的协调和平衡计算资源。

图5.1b为程序整体的运行时间的加速比，除了空间电荷效应求解之外，程序总体运行时间还包括外部传输矩阵，从Z坐标到T坐标的变换，粒子信息统计以及输入输出。同空间电荷效应求解器相比，整体时间的加速比在各个问题规模下都略

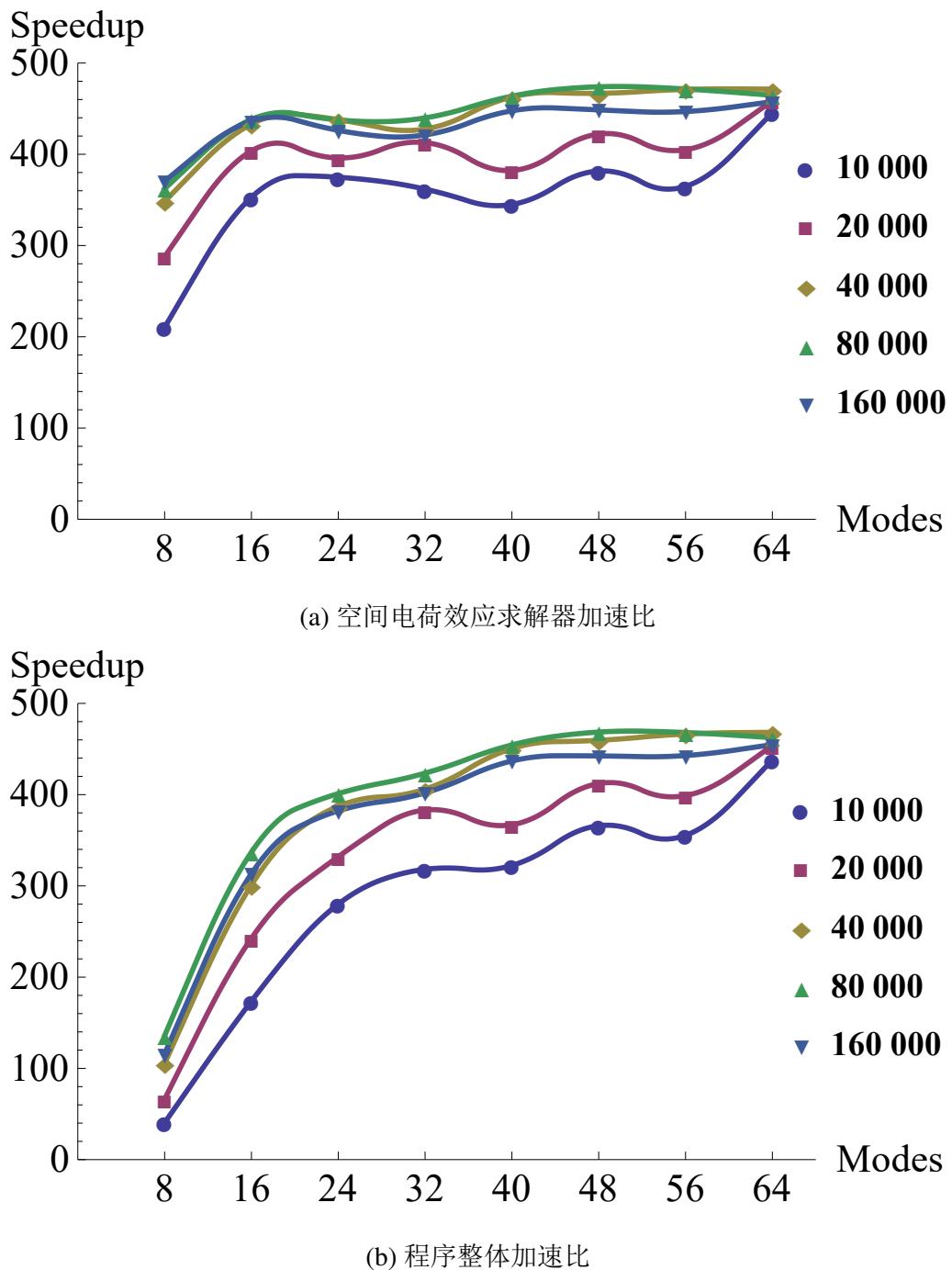


图 5.1: 单GPU加速比

有下降，但是加速比变化的趋势却保持一致。其中，在低阶情况下，比如 $8 \times 8 \times 8$ 阶或 $16 \times 16 \times 16$ 阶，因为空间电荷效应求解器所占总时间的比重不大，所以加速比的下降更为严重。然而，在高阶情况下，空间电荷效应求解所占的时间会占总时间的绝大部分，所以总时间的加速比和空间电荷效应求解的加速比能够基本保持一致。

总之，我们取得了非常好的加速比。对于程序整体的总运行时间，优化的GPU代码比CPU代码运行速度高450倍；而如果仅仅比较空间电荷效应求解器，其最大加速比超过了460。

5.3.2 多GPU性能提升-泰坦

泰坦使用CPU和GPU混合架构，是目前世界上最快，规模最大的计算机集群之一。泰坦的每个节点拥有一个AMD Opteron（16核心）的CPU和一个NVIDIA Tesla K20x（2688核心）的GPU。为了测试多GPU的程序可扩展性，我们最多使用了1024个节点进行测试。其中，为了在不同节点的GPU上交换信息，我们需要先把GPU上的数据拷贝到CPU侧，再使用MPI协议在不用节点之间交换数据，最后再将交换后的信息拷贝回GPU侧。在本次测试中，我们使用 $16 \times 16 \times 16$ 阶进行测试，使用这个阶数是为了精确度和计算速度之间的平衡，也是我们在实际模拟中最常用的阶数。如图5.2所示，我们分别讨论了空间电荷效应和整个程序在不同的粒子数目下，在不同的节点数上的加速比。其中，横轴为节点数目，纵周围加速比，加速比有多节点的运行时间除以单节点的运行时间得到，不同线是不同粒子数目的结果。

由图5.2a可得，在一开始，空间电荷效应求解器的加速比随着GPU的数目几乎线性增加，随后，加速比逐渐到达一个极限。一方面，加速比的线性增长主要是因为GPU之间的数据交换量很少。这是无网格保辛粒子跟踪算法的一个很大的优势，其数据交换量仅与模式数量有关，而与粒子数量无关。另一方面，它可以实现的最大加速度主要受粒子数量的限制，线性增加的范围和极限也随着粒子数量的增加而增加。以160000粒子为例，加速比最大可以达到40。在Titan集群上，每个GPU包含2688个内核，当使用64个GPU时，我们使用了 $64 \times 2688 = 172032$ 个内核，即使用的内核数多于粒子数。在这种情况下，我们无法通过简单地使用更多的GPU来获得进一步加速。而粒子数目增加时，比如320000个粒子或640000个粒子，所能够使用的GPU数目也会随之增加，最大加速比和线性范围也就会增加。

图5.2b是程序整体总时间的加速比，其中外部传输矩阵，坐标变换，粒子信息统计这些部分也是并行化的。但是因为它们的计算量较低，很难取得很高的并行度。所以加速比略微下降。

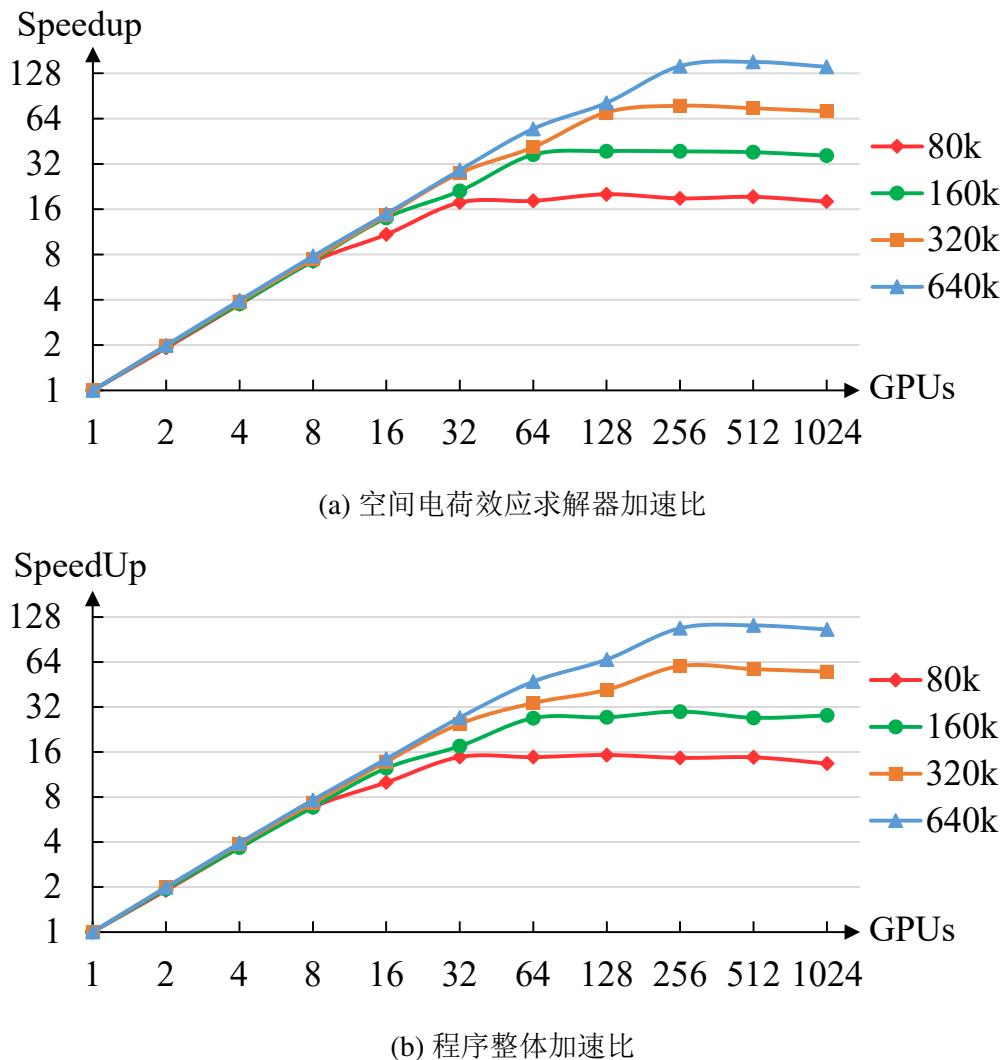


图 5.2: 多GPU加速比

5.4 模拟

我们使用这套保辛算法在周期聚焦结构中进行粒子跟踪和模拟。我们使零流强的周期相移为86.3259度，如果将10个周期视为一个环，则环的工作点为2.3979。随着流强的增加，相移会被压缩，工作点会在0.6A左右穿过2.3333的三阶共振线。环中还存在一个六极磁铁。

图5.3为不同流强下的束流发射度增长变化，可以看到，发射度在流强为0.1A和0.2A时基本保持不变，这时的工作点约为2.40。但是发射度在流强为0.4A, 0.6A, 和0.8A时持续增长，这时的工作点在2.3333附近，可知其发射度增长是由于三阶共振导致。

图5.4是工作点为2.3333附近时的粒子坐标的庞加莱截面，其中颜色越暗表示

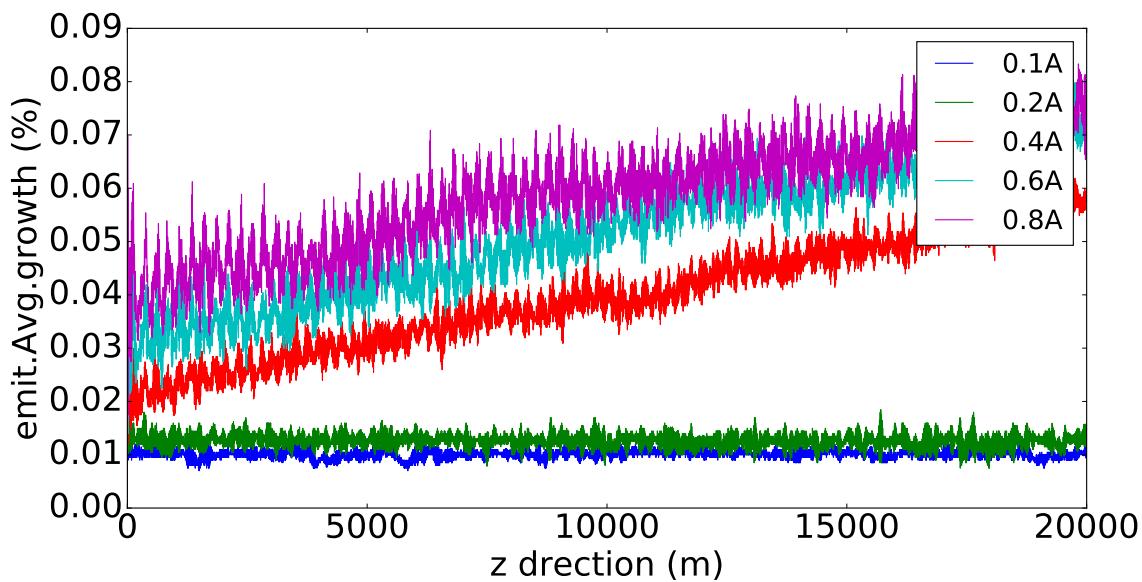


图 5.3: 不同流强下的发射度增长

粒子密度越大，而不同的图片代表粒子处于不同的初始位置。受空间电荷效应驱动，庞加莱截面会被被扭曲，并塑造成三角形形状。受到三阶共振影响的粒子的横向位置会逐渐变大。最后，粒子会成为束晕的一部分并丢失。

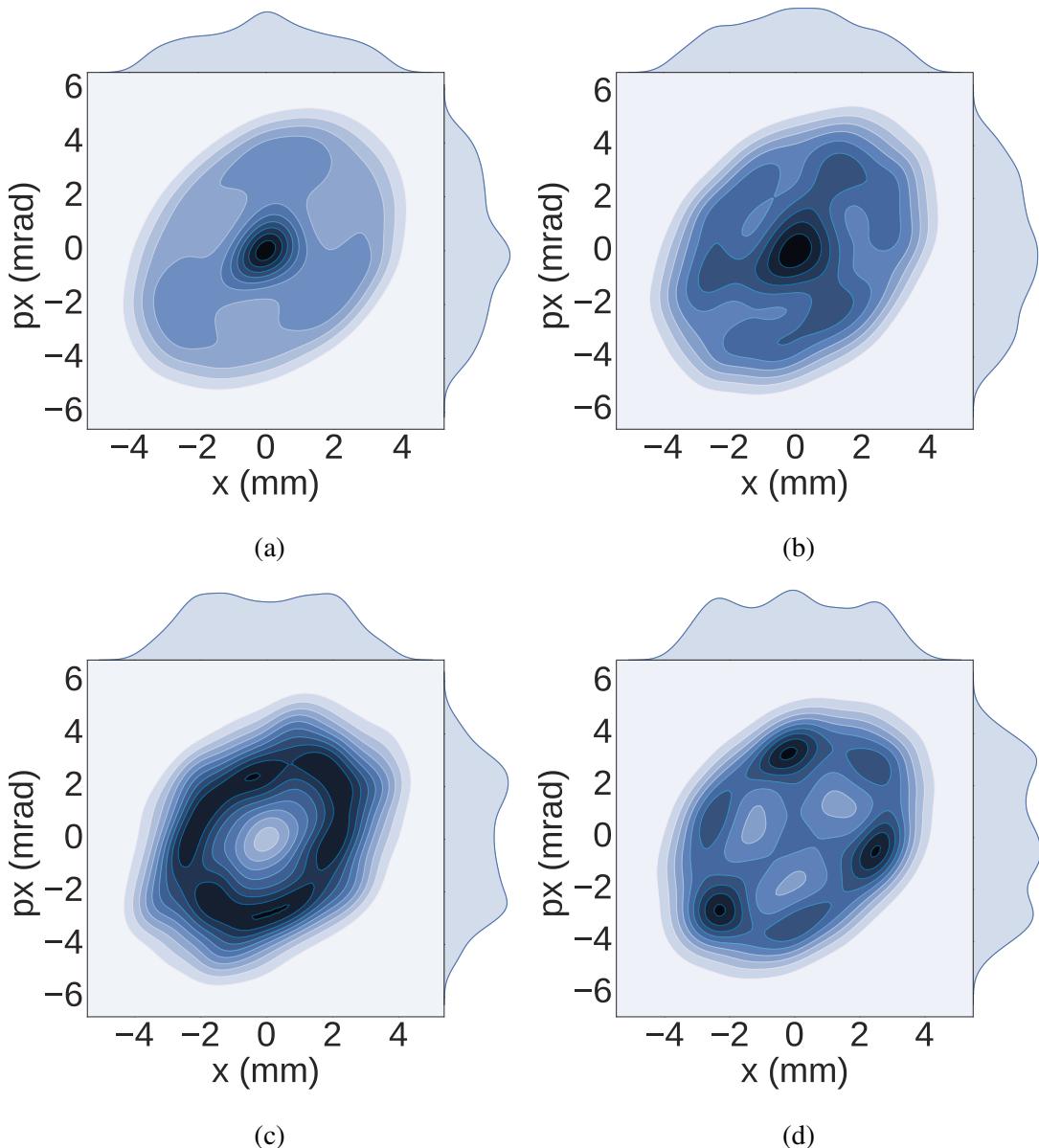


图 5.4: 三阶共振附近的庞加莱截面

5.5 小结

我们使用CUDA库在GPU上实现了无网格保辛粒子跟踪算法。这个算法能够保障辛条件并有效降低由于网格热效应带来的发射度增长。在一个普通家用GPU上，程序获得了超过450倍的加速比。同时，我们在GPU集群泰坦上的测试还显示出这种算法有良好的可扩展性，程序的加速比随着GPU数目几乎线性增加。我们在周期性聚焦结构中使用这个程序进行了几个应用模拟，当工作点远离共振线时，束流不会出现发射度增长，而当其接近共振线时发射度会持续增长。在未来的研究中，我们将继续扩展此程序，在不同架构的计算机上比较保辛算法的效率。

第六章 总结

数值模拟在束流动力学研究和加速器设计中非常重要。

我们在介绍了根据PIC算法编写的粒子模拟程序P-TOPO，主要处理强流加速器中的非线性效应。P-TOPO的PIC部分和整体正确性都得到了验证，并应用在C-ADS注入器I的RFQ和超导段，分别对其进行了模拟并且与其他程序进行了比较。模拟结果证明了现有设计的合理性，束团的尺寸和发射度都得到了有效控制，束损以及能散也在合理范围之内，完全满足需求。之后，我们将继续对P-TOPO进行拓展并加入更多新的功能，以满足强流加速器的各种需求。

我们在GPU上使用使用CUDA库实现了基于PIC方法的多粒子模拟程序，并对如何避免竞争条件和实现更高性能的GPU代码结构和并行策略进行了介绍。在单个GPU卡上，我们使用普通的家用GPU（GTX 1060）实现了超过50倍的加速。当粒子数较大时，程序在GPU集群上也显示出良好的可扩展性；而当粒子数目较小时其可扩展性较差。我们也新的CPU架构Cori Knight Landing上实现了PIC程序，并探索了其最佳性能的并行线程配置。通过比较，单GPU运行的程序和使用4或8个节点的程序性能相当。在未来的研究中，我们将继续扩展此代码并提高效率。

我们使用CUDA库在GPU上实现了无网格保辛粒子跟踪算法。这个算法能够保障辛条件并有效降低由于网格热效应带来的发射度增长。在一个普通家用GPU上，程序获得了超过450倍的加速比。同时，我们在GPU集群泰坦上的测试还显示出这种算法有良好的可扩展性，程序的加速比随着GPU数目几乎线性增加。我们在周期性聚焦结构中使用这个程序进行了几个应用模拟，当工作点远离共振线时，束流不会出现发射度增长，而当其接近共振线时发射度会持续增长。在未来的研究中，我们将继续扩展此程序，在不同架构的计算机上比较保辛算法的效率。

附录 A 粒子模拟程序用户界面

为了提高模拟程序的易用性，我们为其进行了界面设计。

参考文献

- [1] WEI J. Synchrotrons and accumulators for high-intensity proton beams[J]. *Reviews of Modern Physics*. 2003, 75 (4): 1383.
- [2] CHOU W. Synchrotron based proton drivers[C]//AIP, AIP Conference Proceedings: volume 642. [S.l.]: AIP, 2002: 29–37.
- [3] Accelerator code list[M/OL]. https://oraweb.cern.ch/pls/hhh/code_website.disp_allcat.
- [4] TAKEDA H, BILLEN J. Recent Developments in the Accelerator Design Code PARMILA[R]. [S.l.]: [s.n.], 1998.
- [5] QIANG J, RYNE R D, HABIB S, et al. An object-oriented parallel particle-in-cell code for beam dynamics simulation in linear accelerators[C]//ACM, Proceedings of the 1999 ACM/IEEE conference on Supercomputing. [S.l.]: ACM, 1999: 55.
- [6] URIOT D, PICHOFF N. Tracewin[J]. CEA Saclay, June. 2014.
- [7] TANKE E, VALERO S, OTHERS. Dynac: A multi-particle beam dynamics code for leptons and hadrons[J]. LINAC02, Gyeongju, TH429. 2002, 667.
- [8] SHISHLO A, COUSINEAU S, DANILOV V, et al. The orbit simulation code: benchmarking and applications[C]//Proceedings of ICAP. [S.l.], 2006.
- [9] ASEEV V, OSTROUMOV P, LESSNER E, et al. Track: The new beam dynamics code[C]//IEEE, Particle Accelerator Conference, 2005. PAC 2005. Proceedings of the. [S.l.]: IEEE, 2005: 2053–2055.
- [10] OWENS J D, HOUSTON M, LUEBKE D, et al. Gpu computing[J]. *Proceedings of the IEEE*. 2008, 96 (5): 879–899.
- [11] NVIDIA C. Programming guide[M]. [S.l.]: [s.n.], 2010.
- [12] BIRDSALL C K, LANGDON A B. Plasma physics via computer simulation[M]. [S.l.]: CRC Press, 2004.
- [13] LUCCIO A, D' IMPERIO N, BEEBE-WANG J. Space charge dynamics simulated in 3-d in the code orbit “[C]//Proceedings of EPAC2002, Paris, France. [S.l.], 2002.
- [14] LI Z, CHENG P, GENG H, et al. Physics design of an accelerator for an accelerator-driven subcritical system[J]. *Physical Review Special Topics-Accelerators and Beams*. 2013, 16 (8): 080101.
- [15] HENDERSON S, ABRAHAM W, ALEKSANDROV A, et al. The spallation neutron source accelerator system design[J]. *Nuclear Instruments and Methods in*

- Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment. 2014, 763: 610–673.
- [16] ESHRAQI M, DANARED H, DE PRISCO R, et al. Ess linac beam physics design update[J]. MOPOY045, IPAC16, Busan (these proceedings). 2016.
- [17] LI C, LIU Z, ZHAO Y, et al. Nonlinear resonance and envelope instability of intense beam in axial symmetric periodic channel[J]. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment. 2016, 813: 13–18.
- [18] LI C, ZHAO Y L. Envelope instability and the fourth order resonance[J]. Physical Review Special Topics-Accelerators and Beams. 2014, 17 (12): 124202.
- [19] LI C, QIN Q. Collective beam instability and beam halo due to space charge[C]//JACoW, Proc. of ICFA Advanced Beam Dynamics Workshop on High-Intensity and High-Brightness Hadron Beams (HB'16). [S.l.]: JACoW.
- [20] LI C, QIN Q. Space charge induced beam instability in periodic focusing channel a[J]. Physics of Plasmas. 2015, 22 (2): 023108.
- [21] STUDIO M. Cst-computer simulation technology[J]. Bad Nuheimer Str. 2008, 19: 64289.
- [22] HOCKNEY R W, EASTWOOD J W. Computer simulation using particles[M]. [S.l.]: crc Press, 1988.
- [23] WANGLER T, CRANDALL K, RYNE R, et al. Particle-core model for transverse dynamics of beam halo[J]. Physical review special topics-accelerators and beams. 1998, 1 (8): 084201.
- [24] CHEN C, DAVIDSON R C. Nonlinear properties of the kapchinskij-vladimirskij equilibrium and envelope equation for an intense charged-particle beam in a periodic focusing field[J]. Physical Review E. 1994, 49 (6): 5679.
- [25] CRANDALL K R, WANGLER T P, YOUNG L M, et al. RFQ design codes[R]. [S.l.]: [s.n.], 1998.
- [26] NVIDIA C. CUFFT library[M]. [S.l.]: [s.n.], 2010.
- [27] BIRDSALL C K. Particle-in-cell charged-particle simulations, plus monte carlo collisions with neutral atoms, pic-mcc[J]. IEEE Transactions on Plasma Science. Apr 1991, 19 (2): 65-85. DOI: 10.1109/27.106800.
- [28] FRIEDMAN A, GROTE D P, HABER I. Three-dimensional particle simulation of heavy-ion fusion beams[J]. Physics of Fluids B: Plasma Physics. 1992, 4 (7): 2203–2210.

- [29] QIANG J, RYNE R D, HABIB S, et al. An object-oriented parallel particle-in-cell code for beam dynamics simulation in linear accelerators[J/OL]. Journal of Computational Physics. 2000, 163 (2): 434 - 451. <http://www.sciencedirect.com/science/article/pii/S0021999100965707>. DOI: <http://dx.doi.org/10.1006/jcph.2000.6570>.
- [30] QIANG J, FURMAN M A, RYNE R D. A parallel particle-in-cell model for beam-beam interaction in high energy ring colliders[J]. Journal of Computational Physics. 2004, 198 (1): 278–294.
- [31] AMUNDSON J, SPENTZOURIS P, QIANG J, et al. Synergia: An accelerator modeling tool with 3-d space charge[J/OL]. Journal of Computational Physics. 2006, 211 (1): 229 - 248. <http://www.sciencedirect.com/science/article/pii/S0021999105002718>. DOI: <http://dx.doi.org/10.1016/j.jcp.2005.05.024>.
- [32] URIOT D, PICHOFF N. Tracewin[J]. CEA Saclay, June. 2014.
- [33] BATYGIN Y K. Particle-in-cell code beampath for beam dynamics simulations in linear accelerators and beamlines[J]. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment. 2005, 539 (3): 455–489.
- [34] CHANNELL P, SCOVEL C. Symplectic integration of hamiltonian systems[J]. Nonlinearity. 1990, 3 (2): 231.
- [35] YOSHIDA H. Construction of higher order symplectic integrators[J]. Physics Letters A. 1990, 150 (5-7): 262–268.
- [36] QIANG J. Symplectic multiparticle tracking model for self-consistent space-charge simulation[J/OL]. Phys. Rev. Accel. Beams. Jan 2017, 20: 014203. <http://link.aps.org/doi/10.1103/PhysRevAccelBeams.20.014203>. DOI: [10.1103/PhysRevAccelBeams.20.014203](https://doi.org/10.1103/PhysRevAccelBeams.20.014203).
- [37] LEE S Y. Accelerator physics[M]. [S.l.]: World Scientific Publishing Co Inc, 2004.
- [38] CHAO A W, MESS K H, TIGNER M, et al. Handbook of accelerator physics and engineering[M]. [S.l.]: World scientific, 2013.

作者简历

作者基本情况

刘志聪，男，1992年出生于河北省邢台市平乡县，未婚，中国科学院高能物理研究所博士研究生。

教育情况

2009年9月到2013年7月，天津大学，本科。

专业：应用物理学。

2013年9月到2018年7月，中国科学院高能物理研究所，博士研究生。

专业：粒子物理与原子核物理，专业方向：加速器物理。

联系方式

通讯地址：北京市石景山区玉泉路19号（乙）918信箱，中国科学院高能物理研究所

邮编：100049

E-mail: liuzhicong@ihep.ac.cn

攻读学位期间发表的学术论文及科研成果

[1] Li Chao, Liu Zhicong, Zhao Yaliang, Qin Qing, Nonlinear resonance and envelope instability of intense beam in axial symmetric periodic channel, NIMA. 2016, 813: 13-18.

项目资助情况

致 谢

值此论文完成之际，谨在此向多年来给予我关心和帮助的老师、学长、同学、朋友和家人表示衷心的感谢！