2020 届硕士研究生学术学位论文

分类号：　　　　　　　　　　　　学校代码：10269

密　级：　　　　　　　　　　　　学　　号：51174500023

華東師範大學

East China Normal University

硕 士 学 位 论 文

MASTER’S DISSERTATION

论文题目：
基于混合并行架构的 SPH 算法研究

院　　系：软件工程学院

专 业 名 称：软件工程

研 究 方 向：计算机图形学与并行计算

指 导 教 师：王长波 教授

学位申请人：黄可蒙

2020 年 5 月

Dissertation for master degree in 2020 University Code: 10269

Student ID: 51174500023

EAST CHINA NORMAL UNIVERSITY

RESEARCH ON HYBRID PARALLEL FRAMEWORK FOR SPH ALGORITHMS

Department:	School of Software Engineering
Major:	Software Engineering
Research direction:	Computer Graphic and Super Computing
Supervisor:	Changbo Wang
Candidate:	Kemeng Huang

2020.5

华东师范大学学位论文原创性声明

郑重声明：本人呈交的学位论文《基于混合并行架构的 SPH 算法研究》，是在华东师范大学攻读硕士/博士（请勾选）学位期间，在导师的指导下进行的研究工作及取得的研究成果。除文中已经注明引用的内容外，本论文不包含其他个人已经发表或撰写过的研究成果。对本文的研究做出重要贡献的个人和集体，均已在文中作了明确说明并表示谢意。

作者签名:__________ 日期: 年 月 日

华东师范大学学位论文著作权使用声明

《基于混合并行架构的 SPH 算法研究》系本人在华东师范大学攻读学位期间在导师指导下完成的硕士/博士（请勾选）学位论文，本论文的研究成果归华东师范大学所有。本人同意华东师范大学根据相关规定保留和使用此学位论文，并向主管部门和相关机构如国家图书馆、中信所和“知网”送交学位论文的印刷版和电子版；允许学位论文进入华东师范大学图书馆及数据库被查阅、借阅；同意学校将学位论文加入全国博士、硕士学位论文共建单位数据库进行检索，将学位论文的标题和摘要汇编出版，采用影印、缩印或者其它方式合理复制学位论文。

本学位论文属于（请勾选）

（）1. 经华东师范大学相关部门审查核定的“内部”或“涉密”学位论文 *，于年月日解密，解密后适用上述授权。

（）2. 不保密，适用上述授权。

导师签名:__________ 本人签名:__________

年 月 日

* “涉密”学位论文应是已经华东师范大学学位评定委员会办公室或保密委员会审定过的学位论文（需附获批的《华东师范大学研究生申请学位论文“涉密”审批表》方为有效），未经上述部门审定的学位论文均为公开学位论文。此声明栏不填写的，默认为公开学位论文，均适用上述授权）。

黄可蒙 硕士学位论文答辩委员会成员名单

姓名	职称	单位	备注
陈志华	教授	华东理工大学	主席
全红艳	副教授	华东师范大学	
高岩	副教授	华东师范大学	

摘 要

光滑粒子动力学方法 (SPH) 是一种较为流行的数值模拟方法，该方法常被用于流体的模拟仿真。在用 SPH 方法进行数值模拟的时候，需要对空间用粒子进行插值。为了实现较为真实亦或是进行较大规模场景的仿真，需要用到大量的粒子。当仿真场景中的粒子数量以及粒子密集程度增加时，仿真算法的时间开销也会增大，故此仿真效果常和仿真效率存在一个权衡。为了尽可能得到较好的仿真效果，又不增加计算的时间开销，本文进行了关于 SPH 算法加速方面的研究，完成了在不降低 SPH 算法数值精度的前提下，实现算法性能突破。

借助 GPU 强大的并行计算能力，SPH 算法的 GPU 实现比 CPU 实现的性能效果突出。当前，较为高效的 GPU 加速算法主要是基于共享内存的任务调度。考虑到当前主流任务调度算法在进行数值计算时，存在较多冗余数据的重复加载计算，本文给出了一种新颖的双重任务调度策略，通过不同大小的计算任务的并行交错执行，在不增加额外闲置线程的情况下，提高了传统主流任务调度算法的计算性能。此外，考虑到传统主流方法在粒子较为稀疏的仿真场景中的计算效率高于共享内存的任务调度方法，本文设计了新的哈希编码方法，使得共享内存的任务调度方法和传统主流方法可以进行混合计算，从而提高算法的普适性和有效性。

在基于 GPU 的 SPH 算法中，当空间粒子较为密集时，邻居查找的时间开销较大。邻居查找计算过程中，主流的固定网格方法存在较多无关粒子的查找计算，对算法性能影响较大。简单的网格细分方法在算法循环迭代上的时间开销过大，这在很大程度上会影响算法的实际效率。为此本文给出了一种新颖的多层级垂直网格划分方法，该方法较为充分地考虑了邻居粒子数据的连续性，通过在不同层级上进行邻居粒子查找的方式，尽可能减少了循环迭代的次数，缩小了实际的邻居查找空间，进而实现了对 SPH 算法的加速。

综合本文的多层级任务调度方法和多层级邻居查找方法，本文设计了统一的加速框架。基于该框架，结合多种 SPH 算法，实现了完整的加速仿真框架，进行了多种自然场景的仿真，并对多种不同的 SPH 算法进行了性能测试。测试结果表明本文的加速框架在真实的仿真场景中对比主流共享内存任务调度方法可以达到 80% 左右的性能提升。综上所述可知，本文的框架不影响 SPH 算法的数值方法，普适于大多数的 SPH 算法，可直接投入 SPH 算法的实际应用。

关键词: 共享内存，任务调度，网格方法，邻居查找，混合方法

ABSTRACT

Smoothed Particle Hydrodynamics (SPH) is a popular simulation method, which discretizes simulation space with particles. In order to realize large-scale simulation with more details, a large number of particles are necessary. With the increase of particle number, the overhead of algorithm will be increased. This paper focuses on the research of SPH algorithm performance and devises an efficient framework, which can achieve great performance improvement without any compromise of numerical accuracy.

GPU based SPH performs better than the equivalent implementation on CPU. The task scheduling method of shared memory breaks the performance bottleneck for parallel SPH algorithm. However, such method involves the overload of neighbor particles. This paper proposes a novel task scheduling strategy, which can efficiently reduce the overload of neighbor particles. Moreover, a new hash coding method is designed to reduce the waste of computational resource.

In the process of performing neighbor traversal, uniform grid involves a large number of irrelevant particles. The simple subdivision of simulation space involves the more overhead of loop iterations, which greatly affect the actual efficiency of SPH algorithm. This paper proposes a novel hierarchical grid method associated with a novel hierarchical neighbor search method, which fully considers the continuity of the neighbor particles and avoids the large overhead of loop iterations.

Based on the hierarchical strategies of this paper, a novel parallel framework is constructed, and a variety of SPH algorithms are integrated into this framework, so as to perform a variety of physical simulations. The performance tests show that this framework can achieve about 80% improvement in real physical simulations compared with state-of-the-art frameworks.

Keywords:Shared Memory, Tasks Scheduling, Grid Method, Neighbor Traversal, Hybrid Method

目录

第一章 引 言 1
1.1 研究背景和意义 1
1.2 国内外研究现状 3
1.2.1 基于 SPH 物理仿真的算法研究 3
1.2.2 基于 GPU 并行加速的 SPH 算法性能研究 4
1.3 当前的问题和挑战 7
1.3.1 粒子数据分布的合理性问题 7
1.3.2 邻居空间的限定问题 8
1.3.3 任务分配调度的合理性问题 9
1.4 主要工作 9
1.5 文本结构 10
第二章 相关技术 12
2.1 基本 SPH 算法模型 12
2.2 GPU 硬件加速技术 14
2.2.1 GPU 基本硬件结构和 CUDA 编程基本技术 15
2.2.2 基于 GPU 的计数排序技术 17
2.2.3 SPH 算法的 GPU 实现技术 18
2.3 本章小结 19
第三章 SPH 算法的多层级任务调度 20
3.1 单一任务调度策略 20
3.2 双重任务调度策略 23
3.3 混合算法调度策略 29
3.4 实验分析 31
3.5 本章小结 36

第四章 SPH 算法的多层级邻居查找 37
4.1 多层级垂直空间网格 37
4.1.1 网格划分 37
4.1.2 数据编码 38
4.2 邻居查找和空间缩减 41
4.3 多层级网格数据加载方法 45
4.4 实验分析 48
4.5 本章小结 51
第五章 基于混合加速架构的流体仿真 52
5.1 混合加速框架和效率分析 52
5.2 物理仿真 56
5.2.1 海浪仿真 57
5.2.2 液体模型仿真 59
5.2.3 多流体仿真 61
5.2.4 弹性固体液体交互仿真 63
5.3 仿真测试分析 65
5.4 本章小结 67
第六章 总结和展望 68
6.1 本文工作总结 68
6.2 研究展望 69
参考文献 70
致谢 76
发表论文和科研情况 77

插图

图 2.1 影响范围示意图 13
图 2.2 并行前缀求和示例图. 18
图 3.1 基于任务调度策略的 SPH 算法框架流程图 20
图 3.2 计算任务划分示意图 21
图 3.3 计算任务的组织示意图 22
图 3.4 共享内存连续加载方法和传统方法对比示意图 23
图 3.5 计算任务合并流程示意图 24
图 3.6 线程组协作调度示意图 26
图 3.7 混合算法示意图 30
图 3.8 测试场景任务层级展示效果图 32
图 3.9 TB 等于 0 时多层级任务调度算法实验测试结果图 33
图 3.10 TB 等于 14 时多层级任务调度算法实验测试结果图 34
图 3.11 闲置线程阈值参数测试结果图 35
图 4.1 多层级网格划分形式示意图 38
图 4.2 多层级网格编码方式示意图 39
图 4.3 粒子数据连续性示意图 40
图 4.4 粒子数据一致性对比示意图 40
图 4.5 粒子有效邻居空间示意图 41
图 4.6 三维空间各维度视角下数据连续性示意图 42
图 4.7 邻居范围缩减示意图 43
图 4.8 各层级邻居粒子数据加载连续性示意图 46
图 4.9 动态三级网格裁剪示意图 47
图 4.10 参数测试场景效果图 48

图 4.11 不同层级网格裁剪方法实验测试结果图 50
图 4.12 动态网格裁剪方法实验测试结果图 50

图 5.1 整体框架加速效率与粒子数关系测试结果图 54
图 5.2 真实物理仿真整体框架与粒子密集度关系性能测试结果图 55
图 5.3 海浪仿真效果图 56
图 5.4 液体模型下落仿真效果图 57
图 5.5 柱状高速流体对冲仿真效果图 58
图 5.6 多流体耦合仿真效果图 59
图 5.7 弹性固体碰撞仿真效果图 61
图 5.8 弹性固体与液体交互效果图 63
图 5.9 海浪仿真性能测试结果图 64
图 5.10 液体模型下落仿真测试结果图 65
图 5.11 多流体耦合仿真测试结果图 66
图 5.12 固体碰撞仿真测试结果图 67

表格

表 3.1 多层级混合调度算法性能测试用例关键参数表 32
表 3.2 多层级混合调度算法阈值测试用例关键参数表 35

表 4.1 邻居空间缩减测试用例关键参数表 49

表 5.1 粒子密集度测试用例关键参数以及测试结果统计表. 54

第一章 引 言

近年来，随着计算机硬件和软件的发展，计算机图形学相关的算法得到很好的工业应用与性能提升，也由此产生了许多基于硬件和软件方面的图形学算法研究。在软件层面主要集中在新的图形算法的设计以及传统复杂算法的简化两个方面，其目标主要是在保证仿真视觉真实性的前提下减少冗余的计算或复杂的计算开销。在硬件方面的研究主要集中在提高中央处理器 (CPU) 的处理效率以及利用并行加速硬件 (GPU) 进行并行加速计算，进而在算法优化的基础上，实现进一步的性能提升。在众多图形算法中，较为典型的是光滑粒子流体动力学 (SPH) 算法，该算法常常应用于空气动力学模拟仿真以及流体仿真等方面，其计算过程中涉及到大量的粒子计算，计算量巨大，在进行较大场景的仿真时十分耗时。得益于近些年的软硬件发展，该算法效率得到了较大的提升。本章节首先介绍该算法以及相关软硬件优化的研究背景和意义，而后，详细介绍当前该算法的国内外研究现状，从而总结该算法当前面临的主要问题和挑战，最后总结概述本文的主要工作和贡献。

1.1 研究背景和意义

光滑粒子流体动力学方法是一种经典的拉格朗日方法。与传统的固定网格方法不同，该方法将连续的物理空间用粒子进行插值，基于光滑核函数求解每个粒子所表征的物理量，进而更新粒子的物理属性以达到仿真求解的目的。该方法不涉及网格的求解过程，灵活性较高，易于跟踪每个粒子的运动轨迹，获取仿真对象的表面，故此被广泛应用于流体动力学的研究，且在计算机图形仿真和材料力学等领域得到很好的研究应用。在计算机图形仿真领域，SPH 方法常常应用于流体的模拟，如奔涌的海浪，翻滚的泥石流等自然现象，这为相应电影特效以及灾害模拟提供技术支持。在材料力学领域，SPH 方法可应用于材料力学行为的模拟以及可行性和适应性的分析。故此，SPH 方法的研究具有十分重要的应用价值。然而该方法

在仿真模拟细节丰富且规模较大的场景时，插值所需的粒子的数量也会随着场景对象的扩大而扩大，粒子密集度 (粒子总数除以存在粒子的网格数并向上取整计算得到，用于表征仿真场景中粒子的密集程度) 也会因为细节的需求而变大。这意味着计算的时间开销也就越大，由此对计算机的性能需求也就越高。为了节约资源，控制成本开销，提升 SPH 算法的计算性能十分关键。

SPH 算法在计算过程中，涉及到大量的粒子计算。首先，对于每个粒子都需要计算该粒子当前的物理属性变化，故此，粒子的数量越多，计算开销也越大。其次，在每个粒子的属性计算过程中，每一个粒子都需要遍历周围影响区域内的邻居粒子的属性值，以计算更新自身的属性变化，这意味着，邻居领域内的粒子数量越多，计算效率也就越低，换言之，仿真场景的粒子密集度越大，计算开销也会越大。所以在 SPH 算法性能的研究中，如何限定邻居领域的范围是重要的研究点，因为通过限定邻居领域的范围，可以减少粒子进行属性计算过程中搜索的邻居粒子的数量。较为直观和常见的作法是将仿真空间用固定网格进行划分，从而达到限定范围的目的。减少仿真算法的粒子数量也能提高算法的计算效率，但直接减少仿真的粒子数量常常会导致仿真细节的丢失，影响仿真的效果。所以，在该研究方向上，如何保持仿真精度或仿真细节是该方法的关键，即在减少仿真粒子数量的情况下仍能保持较好的仿真精度和效果。当前较为流行的做法是动态合并局部空间的粒子以达到减少仿真粒子的数量，并动态拆分粒子以保持仿真的局部细节。

随着计算机硬件的日益发展，计算机的中央处理器性能日渐强大，尤其是随着计算机并行计算硬件的发展，通过硬件层面的优化提高算法性能成为了 SPH 算法性能研究中的一个要点。当今计算机 CPU 达到了很高的计算性能，不仅 CPU 的核心数和线程数得到提高，CPU 的处理频率也提高了很多，但受限于存储介质的读写效率，CPU 的真实计算性能无法得到充分发挥。因此，提高存储介质的读写效率对计算机性能起到至关重要的影响。通过使用优质的存储介质，可以提高数据读写的效率，但优质的存储介质属于材料学的研究范围，并且，优质的存储介质的成本也越高，在计算机领域，采用的做法是使用多层级的存储架构。级别越高的存储介

质数据读写效率越高，而存储空间也越小，计算机体系结构大多采用这种多层级存储结构。所以，为了充分发挥 CPU 的计算性能，需要尽量减少底层存储介质的读写，换言之，即提高 CPU 的缓存命中率，这在高性能计算研究中是个要点。此外，充分利用 CPU 的多线程，即利用 CPU 有限的并行能力也能提高算法的计算性能。而关于并行计算的研究，更多的研究者倾向于把注意力集中在并行性能远超 CPU 的 GPU 上。得益于 SPH 算法天然的并行性 (每一个粒子在计算各自的物理属性的时候不存在耦合)，基于 GPU 的 SPH 的性能研究进展较大，效果也十分明显。但大多的研究都是停留在 GPU 直观的并行计算上，很少有研究深入到 GPU 的架构和 GPU 中的任务调度上，故此，在近几年中，SPH 的 GPU 加速进展较小。

总而言之，SPH 算法性能研究的进展较大，成果也很多，但较为全面的通用的 SPH 并行加速框架却很少，本文将从算法和硬件层面充分提升 SPH 算法的计算性能以突破 SPH 当前的性能瓶颈。

1.2 国内外研究现状

随着 SPH 算法在计算物理和图形仿真等领域的推广应用，关于 SPH 算法的性能研究也逐渐受到学者们的关注。本节将从 SPH 算法本身的研究现状，传统 SPH 算法并行加速的进程以及当下 SPH 算法的并行加速研究现状出发，介绍相关技术研究的国内外现状。

1.2.1 基于 SPH 物理仿真的算法研究

SPH 算法首先是由 Gingold 和 Monaghan 提出 [1] [2]，用于解决天体物理学相关的问题。该方法的核心思想是将连续的仿真对象离散成众多的粒子，并基于光滑核函数近似求解连续的仿真对象的物理属性，如密度，速度，空间位置等物理属性。在该方法的仿真过程中粒子数量可以保持恒定，这意味着该方法可以保持仿真对象的质量守恒，并且该方法可以很容易地表征仿真对象的物理形态，粒子可以在满足约束的条件下在空间中自由移动，不受网格方法的格点限制。故此，该

方法被广泛扩展应用于计算物理和计算机图形学等领域，尤其在是流体的仿真方面 [3] [4]，SPH 的优点迎合了流体运动过程中的微观性质，易于理解和实现。

随着研究的深入，SPH 算法扩展到了固体的模拟仿真 [5] [6]，以及固体颗粒，如沙子的模拟。随着 SPH 算法应用的成熟，越来越多的学者将 SPH 算法应用于复杂的场景模拟，如固体和液体的交互，这个过程中涉及到较为复杂的边界处理问题。同时该方法也可以用于可变形固体与液体的交互，不同液体之间的耦合，包括固体在液体之间的溶化等较为复杂的自然现象。但在进行复杂的仿真模拟的时候，涉及到不同 SPH 算法之间的交互，算法实现复杂，仿真性能较低。部分学者试图从仿真的物理模型出发，研究新的 SPH 仿真算法，以简化基于传统的 SPH 算法的仿真方式。Ren 等人提出了一种多材质粒子模型 [7]，该方法用一个粒子表征多种液体，不同液体以百分比的参数形式被区分，该方法成功模拟了不同液体之间的耦合，效果较好，实现简便。而后，该方法被 Yan 等人扩展应用，使得该方法也同样适用于液体与多种材质固体之间的交互耦合 [8]。最后，基于 Ren 和 Yan 等人的工作，Yang 提出了一种统一的 SPH 仿真框架，可用于复杂的自然现象的模拟仿真，如生鸡蛋到荷包蛋的变迁 [9]。在近些年的研究中，也有学者关注 SPH 算法不可压缩性的研究，Bender 等人提出了一种高效稳定的方法去保持流体仿真过程中的不可压缩性 [10]。此外，Koschier 等人 [11] 对近些年里，SPH 算法研究相关的工作，做了较为全面和详细的归纳和总结，SPH 算法的技术研究达到了一个较高的高度。

1.2.2 基于 GPU 并行加速的 SPH 算法性能研究

为了较为详细地说明当前基于 GPU 并行加速的 SPH 算法研究，本小节将从三个方面介绍当前的研究现状：早期 SPH 算法的单 GPU 实现，基于 CUDA 的 SPH 算法的单 GPU 实现，多 GPU 的 SPH 算法实现。

早期 SPH 算法的单 GPU 实现

Kipfer 和 Kolb 等人在 GPU 上实现了最早的粒子系统 [12] [13]，该系统可以用于简单的流体的模拟，但该方法的粒子之间不存在互相作用，仿真效果较差。随

后 Kolb 等人在原有的工作基础上提出了粒子之间存在互相作用的粒子系统，该系统是最早的基于 SPH 算法的 GPU 实现 [14]，但该实现并非完全基于 GPU。Harada 等人在 GPU 上实现了 SPH 算法的邻居查找过程，该实现使得 SPH 算法成功完全在 GPU 上实现并行加速 [15]。但这些实现都是基于早期的图形管线接口，实现复杂，且灵活度较低，不易于迁移，难以在 GPU 上实现灵活的调度策略，以充分利用 GPU 的计算性能。

基于 CUDA 的 SPH 算法的单 GPU 实现

随着 GPU 硬件的发展，NVIDIA 推出了基于 GPU 的编程语言 CUDA，该语言风格近似 C 语言，易于学习和理解，也因此，不少学者基于 CUDA 展开并行加速的研究工作。在 2008 年，NVIDIA 首先提供了基于 CUDA 实现的并行加速的粒子系统 [16]。该系统为 SPH 的 CUDA 实现提供了较好的技术思路和研究指导，随后 Hérault 等人 [17] 提出了最早的基于 CUDA 的 SPH 算法的 GPU 实现，他们的工作在他们的 GPU 硬件条件下，相比同样的 CPU 代码有两倍左右的性能提升，效果十分明显。然而他们的实现只是简单地利用 GPU 的并行加速能力，实现较为简单。同年，Goswami 等人基于 CUDA 提出了最早的利用共享内存的 SPH 算法的 GPU 实现 [18]，由于他们在利用共享内存时，数据加载策略的不合理，相比当前的一些未使用共享内存的主流的 GPU 开源框架，其效率反而较低。DualSPHysics [19] 是当前流行的 SPH 加速计算框架，该框架提供了 CPU 的 SPH 实现，同时也提供了单 GPU 以及多 GPU 的实现方法，该框架整合了许多优化策略 [20]，如邻居网格的合并策略，以减少邻居查找过程中的循环迭代次数。在基于 GPU 的 SPH 算法研究中，关于邻居查找是个要点，因为在并行计算过程中，所有粒子求解自身属性变化的时候，可以达到理论上的并行，但是在进行邻居查找过程中，依然同 CPU 实现的算法一样，需要逐个遍历邻居粒子。故此，提高邻居查找的效率是提高基于 GPU 的 SPH 算法的关键。传统的固定网格方法中涉及到较多的无关的邻居粒子，部分学者提出了一种预先保存邻居信息的方法 [21] [22]，且该方法也得到了其他学者的扩展优化，如 Xia 等人就提出了一种新颖的邻居粒子预查找策略 [23]，即预先通过

八叉树的形式进行邻居查找，并将查找到的结果保存成数组，以便接下来计算过程中在保存的粒子数组中查找邻居粒子，从而避免不相关的粒子的重复计算。但基于保存邻居粒子数组的方法需要为所有粒子保存邻居信息，存储空间占用大，不适合较大规模的粒子仿真，通用性较低，且加速效果并不明显。故而，应用较多的依旧是在固定网格中进行邻居查找的方法，DualSPHysics 框架采用的正是固定网格的方法。GpuSPHASE [24] [25]，一个开源的二维空间的 SPH 算法的 GPU 加速框架，给出了其他的一些优化策略，如合并 CUDA 的核函数。该框架也提出了利用共享内存以减少全局显存访问的策略，然而该方法中共享内存并未得到充分利用。Ohno 等人提出了一种将固定网格进行细分的策略 [26]，从而使得搜素的邻居空间更加接近真实的球空间，进而达到减少不必要的邻居粒子的计算查找，然而由于该方法的细分策略较为简单直接，未考虑到粒子存储的连续性，会导致在子空间切换所需的循环跌代次数增加，适用性较低。阮 [27] 和 Huang 等人 [28] 通过合理的邻居粒子的加载策略以及任务分配策略，充分利用共享内存的特性，使得基于固定网格方法的 SPH 算法的 GPU 实现的性能实现突破。

多 GPU 的 SPH 算法实现

受限于单机上的存储空间的限制，为了实现具有更大规模的粒子数量的 SPH 算法，就必须将计算任务分配到多个 GPU 上，从而实现超大规模的模拟仿真。Zhang 等人 [29] 通过动态加载的方式，平衡不同 GPU 上的计算任务，从而实现了较早的多 GPU 的 SPH 算法。Hu 等人 [30] 提出了类似的方法，他们在平衡不同 GPU 上的计算任务做了相应的优化，同时优化了 GPU 间信息交互的策略。Rustico 等人 [31] 根据 GPU 的数量，对仿真空间进行剖分，将各个空间中的计算任务分配到不同 GPU 上。为了保证不同 GPU 上计算的完整性，Rustico 等人让每个子空间跟相邻的子空间都有重叠部分，以保证每个空间中的粒子在不同 GPU 中能被准确的进行计算。而后，Rustico 等人的工作在 2014 年被进一步扩展完善 [32]。Valdez-Balderas 等人 [33] 也提出了类似的空间剖分策略以实现多 GPU 的 SPH 算法，他们的方法使用了信息传递接口。这项工作被 Domínguez 等人通过使用优化过的信息传递接

口进行扩展应用 [34]，扩展后的系统可以实现上亿数量的粒子的模拟仿真。上述方法可以处理超大规模的粒子仿真算法，但在粒子数量较少的时候，上述方法将不再适用。Verma 等人 [35] 则提出了一种同时适用超大规模粒子数量和粒子数量较少情况下的模拟仿真。多 GPU 实现的 SPH 算法主要是为了用于在单一 GPU 上难以实现的模拟仿真。其加速比除受 GPU 的数量影响外，更关键的是每一块 GPU 上的计算性能，因此在 GPU 实现的 SPH 算法中，提高单 GPU 上的 SPH 算法的意义十分重大。

1.3　当前的问题和挑战

在近些年的 SPH 算法性能研究中，该算法的性能得到了很大的提升，尤其在 GPU 的并行加速研究上。但结合各个研究成果可以发现，当前的许多研究局限性较大，一些优化策略适用于某些特定情况，缺乏通用性。并且，SPH 算法的性能仍然存在较大的提升空间。存在的问题可总结为以下几点：粒子数据分布的一致性问题，邻居粒子的查找问题和任务分配调度问题。

1.3.1　粒子数据分布的合理性问题

在 SPH 算法中，涉及到大量的仿真粒子，每个粒子包含多种物理属性，故此涉及到大量的存储数据，随着粒子在场景中的移动，各个粒子的数据分布不再连续一致，这可能会造成数据加载的缓存命中率降低，从而导致数据加载缓慢，计算单元的性能不能得到充分发挥。因此，常见的做法是根据粒子的空间位置对粒子进行哈希编码，并根据粒子的哈希值，对粒子进行空间排序，由此即可使得在邻近的局部区域内的粒子的数据可以尽可能连续分布在内存中，从而减少内存空间的加载次数，即提 CPU 的缓存命中率。这个过程中涉及到的主要问题是对大量粒子进行排序需要较大的时间开销，但相比较低的缓存命中率带来的影响，该做法仍然能提高算法的计算效率。

GPU 的 SPH 算法实现延续了对粒子数据进行排序的做法，不同于 CPU 的是，

在 GPU 端实现粒子排序的排序算法效率远远高于 CPU 的排序算法，常用的是计数排序算法。计数排序算法可以较为充分地利用 GPU 的并行性能，排序时间远远低于整体的计算时间，较好地解决了排序时间较长的问题。但计数排序会造成另外一个问题：在 GPU 端实现计算排序的时候，涉及到统计每个子空间中粒子数量的过程，该过程需要通过原子性操作实现，而原子性操作的返回结果存在随机性，这会导致在局部子空间中粒子的分布存在随机性。这使得基于任务调度的 SPH 算法中，线程协作计算的时候，一致性较低，存在线程间交错等待的问题，在一定程度影响算法效率的稳定性。

1.3.2 邻居空间的限定问题

在 SPH 算法中，每一个粒子在查找周围影响半径范围内的邻居粒子时，需要根据周围空间中所有粒子的相对位置信息以确认该粒子是否在影响半径范围内。由此则会涉及到周围查找空间的范围问题。当前较为合理和普适的方法是将空间按照影响半径的大小划分为众多的立方体子空间。通过确认当前需要进行计算的目标粒子所在的子空间，根据该子空间的位置搜索目标粒子周围空间中最近的 27 个子空间中的邻居粒子。该做法避免了全局仿真空间的遍历，很大程度上加速了邻居粒子的查找。

然而该做法仍然涉及到大量无关粒子的查找计算，为此，针对仿真场景较小，粒子数较少的情况，较为常见的做法是维护每个粒子的邻居粒子数组信息。对于每一个粒子，通过一遍的邻居粒子查找，将查找到的邻居粒子的信息保存下来。故此，在接下来所有涉及到邻居粒子查找的计算过程全部以在已保存的邻居粒子信息中直接获取的形式替代。这在一定程度上避免了不必要的邻居粒子信息的重复计算。显然，由于需要为每一个粒子保存邻居粒子数组信息，该方法涉及到的存储空间远大于简单的空间划分查找方法。该方法的普适性较低，而所起到的加速效果不高。

1.3.3 任务分配调度的合理性问题

在 SPH 算法的 GPU 加速研究中，众多学者根据 SPH 天然的可并行性，直观地将 SPH 算法在 GPU 中实现，却少有关于 GPU 硬件较深层次的加速研究，只有少部分学者关注了 GPU 自身的存储结构。在 GPU 的存储架构中，类似 CPU 的多层级缓存结构，但不同于 CPU 的是，在 GPU 上存在一块可编程控制的共享内存。所以在考虑缓存命中率的基础上，GPU 的性能优化还涉及到如何合理应用调度共享内存，因为共享内存的读写效率远远高于全局显存的读写效率。

然而，共享内存的存储空间很小，无法加载算法所需的全部数据。正因如此，充分利用共享内存的优点，则涉及 GPU 中任务调度和分配问题，这个过程中涉及到线程之间的协作。只有很少一部分学者在 SPH 的 GPU 加速中考虑了 GPU 中的任务划分和调度，而这些工作仍存在未考虑充分的问题。共享内存的任务调度适用于在粒子相对较为密集的区域，而对于那些粒子数十分稀疏的区域，传统简单的加速方式效果更为理想。而且，即便在粒子数密集的区域，简单的任务分配调度也没有充分利用发挥 GPU 的运算效率。因为相邻的任务之间常常存在数据的耦合，有效地利用这个特点，可以进一步提高算法的计算效率。

1.4 主要工作

针对本章提出的问题，本人给出了相应的解决方案，故此，本文的主要工作是针对 SPH 算法当前存在的问题展开的，本文的主要贡献点分为以下三个部分：

本文设计了一种多层级的任务调度方法，该方法考虑了相同任务之间的数据耦合性，尽可能减少了冗余数据的加载，同时也考虑了线程的有效利用问题，即采用多个线程尺度去调度处理不同计算尺度的任务，从而避免增加闲置线程的数量。为了进一步提高算法本身的普适性，本文还充分考虑了空间中粒子分布的随机性，从而设计了一种传统主流算法与任务调度算法混合的调度策略。针对粒子数密集的区域采用高效的任务分配调度算法，而对粒子数较为稀疏的区域，采用更加合适的传统主流方法。

本文设计了一种适用于 SPH 算法的空间网格划分方法。在传统的单一网格的划分基础上，按照垂直的空间划分方式对网格进行划分，并逐层按照垂直划分的方式细分至子网格成立方体形状。根据该网格划分方法，设计相应的数据编码方法，使得粒子数据按照该空间划分方式分布。该方法降低了传统子空间中粒子分布的随机性，提高了粒子数据在存储空间中的一致性。基于多层级网格方法，本文进一步设计了一种在本文设计的多层级网格中进行邻居粒子查找的方法，该方法不仅缩减了邻居粒子的查找范围，根据粒子数据在存储空间中分布的连续性，规避了过多的循环迭代次数，使得多层级邻居查找策略可以起到较好的性能提升。相比于简单的网格划分方法，该方法同样具有普适性，而且加速效果十分明显。相比于维护邻居粒子的方法，该方法没有涉及到邻居数组信息的维护，故此存储空间的占用较小，而加速效果较为理想。

基于上述两项工作，本文设计了一个高效的基于 GPU 的 SPH 仿真加速框架，并展开了针对该框架的性能分析。基于该框架，本文集成了多种不同的 SPH 算法，从而实现了多种复杂的物理现象的仿真效果，同时通过对这些物理现象的仿真测试，本文对多种不同 SPH 算法在本文框架中的计算效率和当前的高效的加速框架进行了对比分析，充分说明了本文的仿真框架的通用性和高效性。

1.5 文本结构

本文的行文结构围绕本文的主要工作展开，全文的结构可分为以下六个章节：

第一章：介绍本文的研究背景和研究意义，并详细总结了当下相关课题的国内外研究现状，由此总结当前 SPH 算法在 GPU 加速中存在的问题，针对这些问题，引述了本文的主要工作和贡献。

第二章：介绍了本文进行实践研究中涉及到的相关技术，主要包括基本的 SPH 仿真技术和 GPU 的加速技术。

第三章：首先介绍了高效的单一任务调度方法，从而介绍了本文的多层级任务调度策略，详细说明了不同计算尺度的任务如何进行同时计算，以及如何进行不同

计算方法之间的调度计算，最后进行了相应的实验对比分析，从而证明本章方法的合理性和有效性。

第四章：详细介绍了本文的空间网格划分方法，给出了该方法的具体空间结构以及特点。针对该空间网格划分方法设计了其对应的数据编码方式，进而详细介绍了本文的空间范围的缩减策略和不同层级上的邻居查找策略，通过相应的实验对比分析，证明了动态多层级任务调度策略的合理性以及本章方法的高效性。

第五章：详细介绍了本文整体框架的整合和设计，给出了整体框架的性能测试分析，并概述了集成到该框架下的多种仿真算法以及基于各个仿真算法进行的物理仿真，最后针对每种算法在本文框架下的计算效率进行了测试分析，证明了本文框架的通用性和高效性。

第六章：对本文的全部工作进行了总结，给出了本文工作中的优缺点，由此对未来的研究内容和方向进行了探讨。

第二章 相关技术

本章节重点介绍本文研究内容涉及到的主要相关技术，分为两个方面，SPH 的物理仿真技术和 GPU 并行加速相关的技术。在 SPH 的物理仿真技术方面，主要探究了基本的 SPH 算法模型；在 GPU 的硬件加速技术方面，主要介绍 GPU 的基本硬件结构以及本文所用的 CUDA 编程技术，并简要介绍了本文研究中涉及到的 GPU 并行算法，包括基于 GPU 的计数排序算法和经典的 SPH 的 GPU 并行实现算法。

2.1 基本 SPH 算法模型

SPH 算法在进行流体仿真的时候，类似于分子动力学的原理，可将每一个单一的粒子视为一个分子个体。流体的流动可视为分子在空间中运动的宏观表象。为了模拟流体的运动，SPH 方法利用粒子在空间中进行空间插值，该过程可视为用粒子模拟流体的分子。而问题从而转变为模拟计算每个分子的受力和运动轨迹，最终实现模拟求解著名的纳维-斯托克方程 [36]：

$$\begin{cases} \frac{D\mathbf{u}}{Dt} = \nu\nabla^2\mathbf{u} - \frac{\nabla p}{\rho} + \mathbf{a}, \\ \nabla\cdot\mathbf{u} = 0, \end{cases} \tag{2.1}$$

其中 $\mathbf{u}$ 代表矢量速度，ν 代表流体的粘滞系数，∇^2 代表拉普拉斯算子，∇ 代表梯度，p 代表压强，ρ 代表密度，$\mathbf{a}$ 代表体加速度 (通常情况下为重力加速度)。在 SPH 算法进行求解纳维-斯托克方程的过程中，粒子的质量通常视为定值，粘滞系数 ν 取决于仿真的流体类型，通常亦为定值，仿真流体的不可压缩性可以通过优化 SPH 算法的计算过程实现，通过调节物理参数也能在一定程度上实现较好的不可压缩性。故此公式2.1中需要求解的量为速度场的散度 $\nu\nabla^2\mathbf{u}$ 以及压强场的梯度 ∇p 和流体各个粒子所在位置的密度 ρ。

在 SPH 算法中，完成粒子在空间中的插值后，需要基于特定的光滑核函数，根

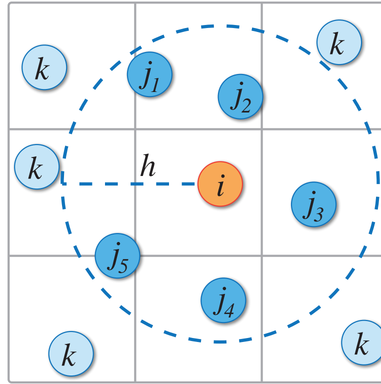

图 2.1: 影响范围示意图

据特定的数学模型进行计算：

$$A(\mathbf{x}_i) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{x}_i - \mathbf{x}_j, h), \tag{2.2}$$

其中，$A(\mathbf{x}_i)$ 代表位于 $\mathbf{x}_i$ 位置处粒子当前待求解的物理量，同理 $A(\mathbf{x}_j)$ 代表位于 $\mathbf{x}_j$ 处的邻居粒子的当前的物理属性值，m_j 和 ρ_j 则分别代表 $\mathbf{x}_j$ 处粒子的质量和密度，h 为该数学模型进行累加计算的影响半径，W 则代表光滑核函数。如图2.1所示，小圆圈代表空间中的粒子，橙色粒子代表需要进行求解的粒子，深蓝色的粒子代表参与求解的邻居粒子，浅蓝色的粒子代表和当前待求解粒子无关的邻居粒子，每一个待求解的目标粒子，根据影响半径确定其有效的邻居粒子，根据公式2.2对在影响半径范围的粒子属性值进行累加求和从而近似计算出目标粒子的属性值。

光滑核函数通常是一个径向对称和归一化的函数，其表达形式根据仿真的要求不同而具有差异性。在求解密度的时候，Müller 等人的做法较为流行，采用 Poly6 函数进行求解 [3]，其三维表述形式为：

$$W_{poly6}(\mathbf{x}_i - \mathbf{x}_j, h) = \begin{cases} \frac{315}{64\pi h^9}(h^2 - |\mathbf{x}_i - \mathbf{x}_j|^2)^3, & 0 \le |\mathbf{x}_i - \mathbf{x}_j| \le h \\ 0, & |\mathbf{x}_i - \mathbf{x}_j| > h \end{cases}. \tag{2.3}$$

在求解速度的散度时，涉及到拉普拉斯算子，采用的数学模型为：

$$\nabla \cdot \nabla A(\mathbf{x}_i) = \sum_j m_j \frac{A_j}{\rho_j} \nabla^2 W(\mathbf{x}_i - \mathbf{x}_j, h), \tag{2.4}$$

这里的光滑核函数通常采用 viscosity 函数，在公式2.4中 $\nabla^2 W$ 最终表示为：

$$\nabla^2 W_{viscosity}(\mathbf{x}_i - \mathbf{x}_j, h) = \begin{cases} \frac{45}{\pi h^6}(h - |\mathbf{x}_i - \mathbf{x}_j|), & 0 \le |\mathbf{x}_i - \mathbf{x}_j| \le h \\ 0, & |\mathbf{x}_i - \mathbf{x}_j| > h \end{cases}, \tag{2.5}$$

需要注意的是在公式2.4中代入 A_j 的速度不是位于 $\mathbf{x}_j$ 处粒子的绝对速度，而是该处粒子相对于目标粒子的相对速度 $\mathbf{u}_j - \mathbf{u}_i$。在求解压强的梯度前，需要先计算各个粒子所在位置处的压强，这里可以采用理想气体状态方程进行计算：

$$p = \kappa(\rho - \rho_0), \tag{2.6}$$

公式2.6中的 ρ_0 代表的是静态流体的密度，通常视为定值，κ 代表的是流体相关的常数。为了在仿真过程中较好地保持部分仿真流体的不可压缩性，如液体的仿真，较为常用的是 Tait 方程 [8][37][38]：

$$p = \frac{\kappa \rho_o}{\gamma}((\frac{\rho}{\rho_0})^{\gamma} - 1), \tag{2.7}$$

这里的 γ 通常为定值 7，本文在具体的仿真实践中均采用公式2.7。通过压强的计算公式得到各个粒子所处位置的压强后，则可进行压强梯度的计算，采用的数学模型为：

$$\nabla A(\mathbf{x}_i) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(\mathbf{x}_i - \mathbf{x}_j, h), \tag{2.8}$$

其对应的光滑核函数是 spiky 函数，故而 ∇W 的最终表述形式为：

$$\nabla W_{spiky}(\mathbf{x}_i - \mathbf{x}_j, h) = \begin{cases} (\mathbf{x}_j - \mathbf{x}_i)\frac{45}{\pi h^6}(h - |\mathbf{x}_i - \mathbf{x}_j|)^2, & 0 \le |\mathbf{x}_i - \mathbf{x}_j| \le h \\ 0, & |\mathbf{x}_i - \mathbf{x}_j| > h \end{cases}. \tag{2.9}$$

通过上述公式即可实现对纳维-斯托克方程的求解，进而实现基本流体效果的仿真。

2.2 GPU 硬件加速技术

本节将重点介绍本文中涉及的 GPU 并行相关的技术。首先概述 GPU 的基本硬件结构，再介绍 CUDA 的基本操作技术。而后概述本文中涉及到的两种 GPU 并行加速技术。

2.2.1　GPU 基本硬件结构和 CUDA 编程基本技术

对于不同架构的 GPU，其硬件结构存在差异，即便是相同架构的 GPU，其硬件结构也存在差异。故此本小节主要概述本文内容所涉及的 GPU 的基本结构，主要分为两个方面，GPU 的存储结构以及 GPU 的处理器结构。并且，在介绍 GPU 基本结构的同时，对基于 CUDA 的多线程操作技术也会进行概述。

在 GPU 中，其存储结构与 CPU 的存储结构相似，存在多种存储结构。其中，寄存器是线程独享的，也就是说寄存器中的数据原则上只能被当前占有的线程访问。在 GPU 进行读取操作时，如果被读取的数据会被多次使用，考虑到寄存器读写的高效性，往往会把这些数据先加载的寄存器中，以提高算法的效率。在 GPU 中，也存在多级缓存架构，通常情况下由一级缓存和二级缓存构成。同 CPU 的优化策略相同，如果能提高数据读写的缓存命中率也能提高 GPU 算法的效率。这里值得一提的是，在英伟达早期的 Kepler 显卡架构和近些年流行的 Turing 架构中，一级缓存和共享内存共用同一块存储区间，而在 Maxwell 和 Pascal 架构中，共享内存和一级缓存是相互独立，在这些架构的显卡中，共享内存都拥有很高的访问效率，并且共享内存是编程人员可以编程控制的，相比于考虑不可控的缓存命中率，充分利用共享内存具有更大的研究空间和意义。共享内存相比于寄存器而言，其数据是可共享的，但并非所有线程均可访问任意共享内存中的数据，只有同一线程块中的线程可以访问该线程块占有的共享内存。

不论是寄存器还是共享内存，存储空间相对都较小，GPU 中存储空间最大的是全局显存。相比于寄存器和共享内存，全局显存的数据是所有线程都可以进行的读写的，但其读写的效率远低于寄存器和共享内存。由于全局显存的读写效率较低，故而，对 SPH 这类数据密集型的算法而言，全局显存的读写效率对算法性能的影响较大，通过合理地减少全局显存的操作，可以明显提高这类算法的性能。为了提高全局显存的访问效率，可以通过有效地利用 GPU 的合并策略，这里涉及到 GPU 的一个重要概念，线程簇 (warp)，后面将做进一步介绍。对于半个 warp(通常是 16 个线程)，当其访问连续的数据，且数据大小满足一定规律和起始地址对称

的情况下，其访问的存储空间会被合并作为一整块存储空间进行访问，从而避免多次的数据加载和全局显存的访问。

在存储上，本文的内容主要涉及全局显存，缓存，寄存器和共享内存。在 GPU 中还存在一些其他的存储空间的概念，如 GPU 中的纹理空间和常量空间。当多个线程在访问不连续的地址空间，而这些地址在存储空间中相邻比较近的时候，使用全局显存是无法进行合并加载的。但使用纹理空间，借助于特有的纹理缓存，这样的数据访问可以获得较好的加速。通常情况下纹理空间和常量空间在编程使用时，是只读不可写的存储空间，但在 CUDA 中提供了修改纹理空间数值的函数。常量空间拥有常量缓存，该存储空间适合于存放被许多线程访问的常量数据。

GPU 突出的并行能力主要来自于 GPU 中的众多运算单元。不同架构的 GPU 中运算单元的组织形式往往不同，但大致结构基本一致。在大多数 NVIDIA 的 GPU 中，存在多个流多处理簇 (SM)，而每个 SM 又拥有多个流处理器 (SP)。在 CUDA 编程中定义的一个线程块只能在一个 SM 上运行，而 SM 上的 SP 数量有限，线程块中的线程数往往远超运算单元的数量，所以 GPU 在进行运算的时候需要通过不断地切换实现一个线程块仅由一个 SM 负责运算。正因如此，利用 CUDA 定义的线程并非都是同时运行的。每个线程块中的线程会被分解成多个 warp，而 warp 则是 GPU 线程调度的基本单位。

在了解上述的 GPU 的相关技术后，还要了解 CUDA 的基本操作。首先需要了解 CUDA 的线程组织形式，上述内容已经提及了线程块的概念，线程块即一组逻辑线程的集合，在组织线程的时候，CUDA 将线程组织成一维或多维的线程块，每个线程块中的线程数是一样的。同一个线程块中的线程可以进行同步操作，可以共同访问同一块共享内存。线程的组织可以在启动 CUDA 核函数的时候进行定义。在完成线程的定义后，每个线程需要确定自己的线程序号 id，CUDA 为每个线程提供了获取该线程所在的线程块的序号 b_{id} 以及该线程在当前线程块中的序号 t_{id} 和当前的每个线程块中线程的数量 t_{size}，由此根据公式：

$$id = b_{id} \times t_{size} + t_{id}, \tag{2.10}$$

可以计算出当前的线程序号，从而根据该序号从组织好的数据存储区域读取每个线程所需的数据，进而实现单指令多数据的并行计算模式。

2.2.2　基于 GPU 的计数排序技术

在 SPH 算法的性能研究中，预处理过程起到了关键的作用。全局显存的数据读写效率较低，为了提高算法的效率，应尽量减少全局显存的访问。如上所述，应尽可能让数据的分布具有良好的连续性，从而可以利用 GPU 的数据合并机制以减少全局显存的访问次数，此外提高数据的连续性还能在一定程度上提高缓存的命中率。基于 CPU 的 SPH 算法性能研究中也涉及到类似的问题。当前流行的做法是根据粒子的位置信息对其进行编码，根据编码对粒子的数据进行排序，从而使得粒子的数据分布具有较好的连续性。在这个预处理过程中，排序算法的效率十分重要。当前基于 GPU 的 SPH 算法中，较为高效的是计数排序算法，本小节对 SPH 算法中基于 GPU 的计数排序算法进行概述。

在当前应用较广的 SPH 算法中，将仿真空间按照 SPH 算法中的影响半径划分为等大的立方体子空间 (三维仿真中)，并对每一个子空间进行哈希编码。在用粒子进行插值的时候，根据粒子所处子空间的哈希值，给每一个粒子赋值，作为粒子的哈希值。在这个过程中，处于同一个子空间的粒子拥有一样的哈希值，在进行哈希值运算的过程中，哈希值的范围是确定的，因此，可以使用计数排序对粒子的哈希值进行排序。首先需要为每一个粒子分配一个线程以统计在哈希值范围内每个哈希值对应的粒子数量。在这个并行加速的过程中，由于一个哈希值对应多个粒子，统计过程中涉及到计数问题，而拥有相同哈希值的粒子会对计数变量同时进行累加，此时会产生写冲突。为了避免这个问题可以用 CUDA 的原子性求和函数 atomicAdd，该函数在进行统计求和的时候会返回当前对求和变量进行累加的粒子的操作次序 s，即粒子排序后在子空间中的相对顺序。

在完成粒子哈希值的统计后，可以得到了每个哈希值对应的粒子的数量，即每个子空间中的粒子数量 N_c，然后，从该序列的第二个哈希值开始，对该序列进行

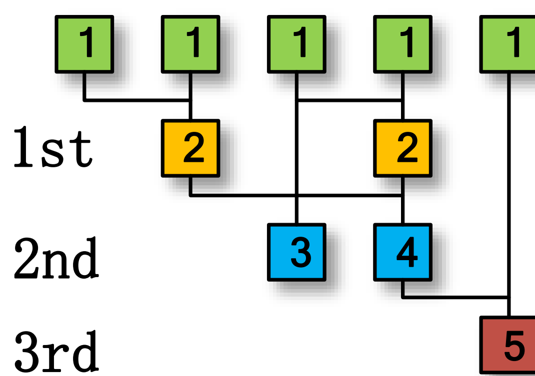

图 2.2: 并行前缀求和示例图.

前缀和计算。在进行前缀和计算的过程中，后面元素的计算依赖于前面元素的计算结果，故此存在数据耦合，不可直接并行展开。通过类似二分的方式将进行简化计算，利用 GPU 的并行能力可以将该过程的计算时间复杂度从 $O(n)$ 降为 $O(\log_2 n)$。其并行计算过程如图2.2所示，每一个线程负责两个元素进行加和操作，计算结果保存在图中靠右的元素的存储位置，故在同一竖直线上位于下方的元素值代表当前该存储位置上的值。在 CUDA 中提供了与该方式计算效率接近的前缀求和函数 exclusive_scan，这两种方式的前缀求和操作十分高效。

在得到每个粒子的累加次序和对应哈希值的统计结果的前缀和计算结果后，即可进行排序，首先根据粒子的哈希值去前缀求和的结果中获取对应的值，而该值代表的是哈希值小于该粒子的粒子数量 O_c，由此根据该值以及当前粒子在进行原子性求和的操作次序相加，即代表该粒子在排序结果中的次序，每个粒子根据自身的次序将各自的数据结果写入对应的数组中，即可实现数据的排序。通过这种排序方式，可以使得位于同一子空间和哈希值左右相邻的子空间中的粒子数据位于一段连续的存储空间，这在一定程度上提高了数据的连续性。

2.2.3 SPH 算法的 GPU 实现技术

本章的上述内容对基于 GPU 的 SPH 算法基础内容做了较为详细的概述，本小节将重点介绍基于 GPU 的 SPH 算法的基本实现技术。

正如第一章所述，由于粒子之间的计算互不影响，SPH 算法具有天然的可并

行性。在 CPU 的实现中，需要利用循环遍历所有的粒子，从而实现对每一个粒子的求解计算。与 CPU 版本的 SPH 算法不同的是，GPU 版本的实现中，通过对每一个粒子分配线程，来完成每一个粒子的求解计算，在 CPU 中通过循环变量来确定粒子的序号，而 GPU 中则是根据线程块序号和线程块中线程的序号来确定粒子的序号，其具体的求解计算过程与 CPU 版本的实现基本一致。

在 SPH 算法的 GPU 实现中，数据的初始化最初需要在 CPU 中进行初始化，然后再将初始化完成的数据拷贝到对应的 GPU 的存储空间。同样，当 GPU 完成仿真计算需要将仿真结果保存到本地的时候，需要将数据从 GPU 中拷贝到 CPU 内存中，再由 CPU 完成数据保存。需要注意的是，SPH 算法的 GPU 实现中，粒子数据的存储空间大小确定以后不可再改变，为了增加存储空间以动态增加仿真粒子，需重新申请存储空间，并将原有空间中的数据全部拷贝到新申请的存储空间中，该过程较为繁琐且对算法的性能影响较大。所以在进行 GPU 的算法实现过程中涉及到动态内存分配问题时，往往是分配足够的存储空间以应对扩容的需求。

2.3　本章小结

本章概述了本文中涉及到的相关技术，包括基本 SPH 算法的技术原理，关于 GPU 硬件架构，主要概述了 GPU 中的存储结构以及 GPU 中的运算单元的基本结构，并简要介绍了 CUDA 的基本操作。基于 GPU 的加速技术，本章主要介绍了基于 GPU 的计数排序算法以及如何通过 GPU 实现 SPH 算法。

第三章 SPH 算法的多层级任务调度

本章提出了一种新颖的任务调度方法，该方法是在阮的任务调度方法 [27] 的基础上改进扩展得到。不同于阮的并行任务调度方法，本章提出的方法进一步考虑了空间粒子的分布情况，针对不同分布情况的粒子群，采用不同尺度的任务划分方式进行调度计算；同时，通过结合传统主流方法的算法优势，提出了一种混合算法调度的策略，从而提高计算资源的利用率和算法的普适性，本章将会对这些方法进行详细介绍。

3.1 单一任务调度策略

本章提出的方法是在阮的任务调度方法的基础上改进扩展得到，故此需要先介绍阮提出的单一任务调度策略，该策略是通过改进 Goswami 等人提出的基于共享内存的任务调度方法 [18] 得到。在具体的算法细节上，阮和 Goswami 等人的方法存在较大差异，但在总体的 SPH 算法设计上大致相同，其过程如图3.1所示，首先进行空间粒子插值，然后进行粒子的编码排序，这两个过程与当前大多数的 SPH 算法相同。在完成粒子的排序后，则需要完成每个子空间网格中针对粒子的计算任务划分，而后根据划分的计算任务进行基于共享内存的并行求解计算，这两个过程是任务调度方法的主要创新所在。最后根据求解结果更新粒子的空间位置从而实现仿真计算。

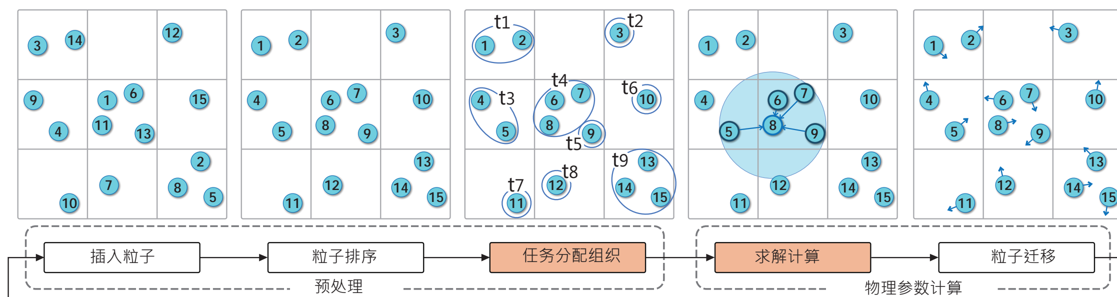

图 3.1: 基于任务调度策略的 SPH 算法框架流程图

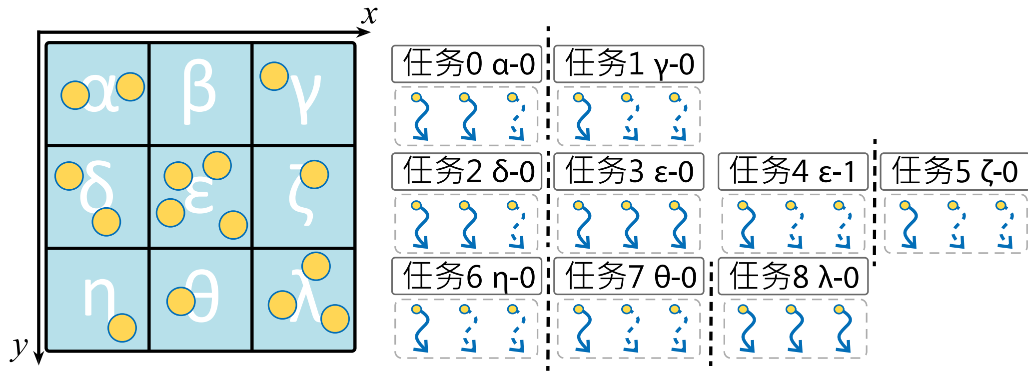

图 3.2: 计算任务划分示意图

在进行任务配分前，需要进行任务的划分，划分的原则是根据给定的计算任务尺度 S_i，将每一个子空间中的粒子分成多个计算任务，当粒子的数量少于计算任务的尺度时，当作一个计算任务处理。Goswami 等人未对计算任务的尺度大小进行研究和说明，阮通过对不同尺度的任务划分方式进行了较为详细的测试，认为将任务尺度大小设置为 32 是比较合理的。具体的任务划分过程如图3.2所示 (为了说明方便，图中的 $S_i = 3$)，完成任务划分后，根据计算任务的数量，组织相同数量且大小为 S_i 的线程块对每一个计算任务进行计算，实线的线程代表参与粒子计算的线程，虚线的线程代表的是闲置线程，线程的闲置数量取决于计算任务中实际的粒子数量。

在传统的基于 GPU 实现的 SPH 算法中，针对所有的仿真粒子，进行线程的组织安排，线程与粒子之间存在一一对应的关系，故此，通过 CUDA 中简单的线程序号计算方式就可以实现线程和粒子之间的对应。但是，在任务调度方法的线程块中存在着离散的闲置线程，无法通过传统的线程序号的计算方式完成线程与粒子之间的对应关系。为了在任务调度方法中实现线程和粒子的一一对应，需要基于计算任务进行线程的组织安排。其计算流程如图3.3所示，针对图3.2中的粒子分布情况，首先根据每个子空间中粒子的数量 N_c，计算其对应的计算任务的数量，即其所需的线程块的数量，然后根据每个子空间中线程块数量的计算结果进行前缀求和操作，以计算小于每个子空间编码值的子空间中的线程块的数量。最后在每一个子空间中，针对每一个计算任务，计算每个任务在子空间内的相对序号 b_{id} 以

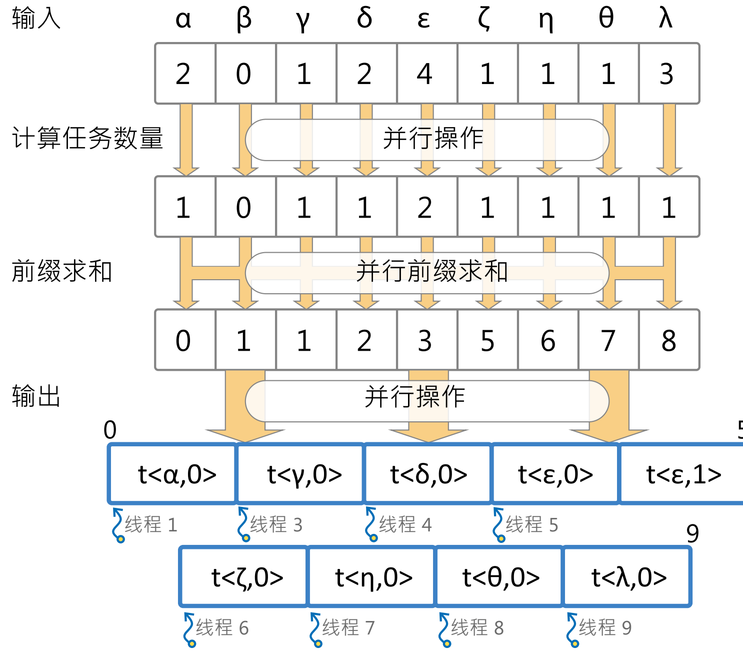

图 3.3: 计算任务的组织示意图

及该任务所在子空间的编码值 H，从而完成任务队列的组织，线程与粒子之间对应关系的计算公式如下：

$$\begin{cases} id = O_c + b_{id} \times S_i + t_{id} \\ id < N_c \end{cases} . \tag{3.1}$$

完成任务的划分和组织后，即可进行线程的协作调度，从而达到利用共享内存的目的。对处于同一个计算任务中的粒子而言，他们都是处于同一个子空间。按照固定网格的邻居查找方法，每一个粒子要搜索周围 27 个子空间中的粒子，显然同一个任务中的粒子具有相同的搜索空间，故而存在大量相同的邻居粒子，在这种情况下，共享内存可以发挥重要作用。在利用共享内存的方式上，Goswami 等人分别为每一个子空间分配一个线程，利用该线程从每一个子空间中分别加载一个邻居粒子的数据到共享内存，计算任务中的目标粒子则分别从共享内存中加载邻居粒子的数据进行计算，该策略可以减少全局显存的访问，理论上可以起到加速作用。但是，由于 27 个子空间中，每个子空间的粒子数量各不相同，使得粒子数相对较少的子空间中的粒子数据较早被加载完毕，导致部分线程闲置，而且 27 个线程的使用不能充分发挥 GPU 的并行效率。故此，该方法的局限性较大，在很多情况下，

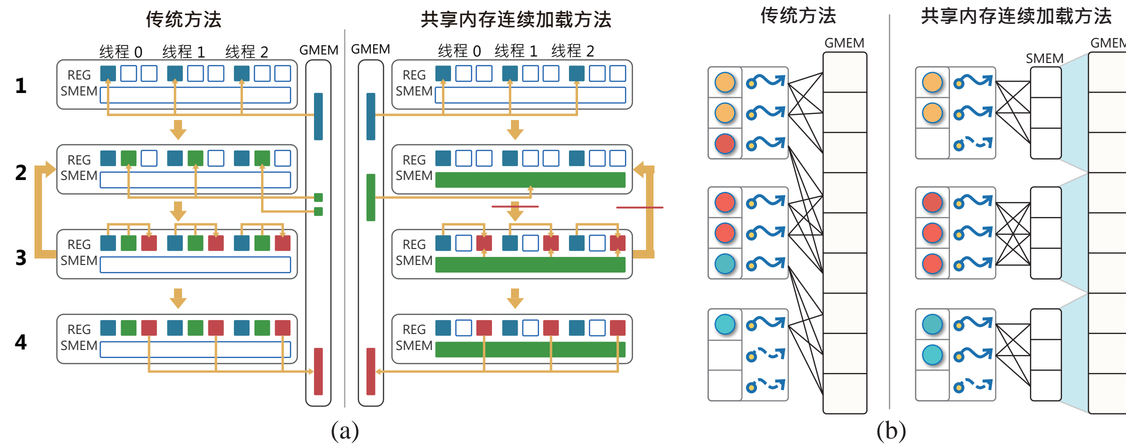

图 3.4: 共享内存连续加载方法和传统方法对比示意图

加速效果甚至低于优化后的传统主流的 GPU 加速方法。

阮等人则采用较为合理的加载策略，即用线程块中的所有线程连续加载同一块邻居存储区域中的粒子数据，由于他采用了邻居合并的优化策略，故而，需要加载的邻居粒子空间从原来的 27 个区域缩减为 9 个存储区域。该方法充分利用了数据存储的连续性以及 GPU 线程之间的协作能力。图3.4展示了基于共享内存的连续加载策略与传统主流方法的区别和优势；SMEM 代表的是共享内存，GMEM 代表的是全局显存，REG 代表的是寄存器；阮的加载策略可以使得全局显存的合并策略发挥作用，其共享内存的利用方式在很大程度上减少了全局显存的访问。

3.2　双重任务调度策略

单一任务调度策略在一定程度上提高了 SPH 算法的计算效率，但是由于其任务尺度的单一性，使得该调度算法存在较多问题。如图3.2所示，连续的计算任务可能来自同一子空间，这样的计算任务中的目标粒子也拥有相同的邻居查找空间。单一尺度的任务调度算法将这样的任务分别交给不同的线程块进行处理，这使得同样的邻居粒子被多次重复加载到共享内存中，造成了不必要的计算开销。在这种情况下，应当将这样的两个或多个任务合并为一个尺度更大的计算任务，同时也将其对应的线程块合并进行协同计算。通过合并的策略不仅可以减少冗余数据的重复加载，同时也提高了数据加载的效率，因为更大的线程块一次可以加载更多的

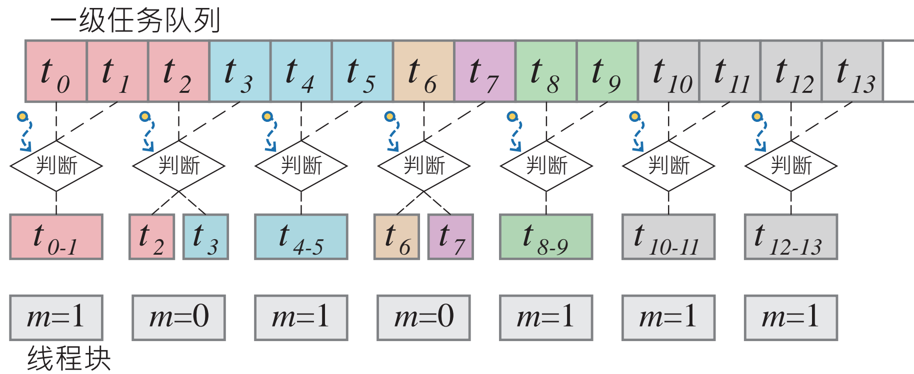

图 3.5: 计算任务合并流程示意图

邻居粒子的数据，从而减少了数据加载操作所需的循环迭代的次数。为此，本节提出了一种高效的二任务合并策略，从而将任务分为一级任务和二级任务。

在进行 GPU 的线程组织时，线程块的大小是明确统一的，不存在不同尺度的线程块同时执行的方法。为了实现上述的两种不同尺度的任务混合并行执行的操作，本节采用了一种线程块内线程分组调度的方法：将线程块中的线程数设置为单任务调度方式的两倍，将线程块中的线程分为两个线程组，块内线程组序号的计算公式为：

$$z_{id} = \lfloor \frac{t_{id}}{32} \rfloor, \tag{3.2}$$

根据任务的尺度决定两个线程组是否需要协同计算。计算流程如图3.5所示，图中相同颜色的一级任务来自同一子空间，首先为连续的两个一级任务对分配一个线程，该线程用于判断两个一级任务是否来自同一子空间，如果两个一级任务来自同一子空间，那么这两个任务则会进行合并。需要注意的是，本节中的合并并非真实地将两个任务合并在一起，而是采用标记变量进行标记的方法；对于需要合并的一级任务，将其标记变量 m 设置为 1，而不需合并的一级任务，将 m 设置为 0。合并过程的具体的实现细节如算法1所示，在进行合并前需要确定一级任务数量的奇偶性，根据其奇偶性做相应的处理。需要注意的是，线程的序号从 0 开始，并且在一级任务数量为奇数条件下，对最后一个处理任务，均采用两个线程组协同加载邻居粒子的方式，从而减少闲置线程的数量，同时减少邻居加载所需的循环迭代次数。

算法 1: 一级任务合并算法

输入: 一级任务队列 **T**，一级任务总数 N_t

```
1  foreach 线程 do
2      根据公式2.10计算线程 id
3      if N_t%2 == 0 then
4          //一级任务总数为偶数的时候
5          if id%2 == 0 then
6              if T[id].H == T[id+1].H then
7                  T[id].m = 1;    T[id+1].m = 1
8              else
9                  T[id].m = 0;    T[id+1].m = 0
10             end
11         end
12     else
13         //一级任务总数为奇数的时候
14         if id%2 == 0   And   id < N_t − 1 then
15             if T[id].H == T[id+1].H then
16                 T[id].m = 1;    T[id+1].m = 1
17             else
18                 T[id].m = 0;    T[id+1].m = 0
19             end
20         end
21         if id == N_t − 1 then
22             //最后一个计算任务的标记
23             T[id].m = 1;    T[id+1].m = 1
24         end
25     end
26 end
```

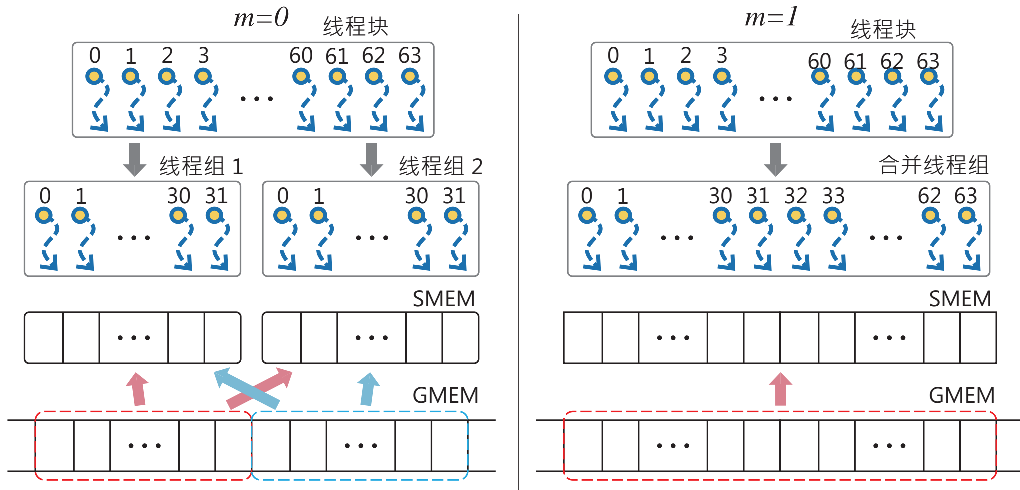

图 3.6: 线程组协作调度示意图

对于一级任务，其尺度大小采用阮提出的 32 的设定方式，所以本文中的线程块大小为 64。在完成一级任务的标记以后，根据一级任务中标记变量的值 m，需要重新计算当前线程块内的线程序号，计算公式为：

$$t_{id} = t_{id}\%32 + 32 \times m \times z_{id}. \tag{3.3}$$

通过公式3.3可将线程块内的线程动态分组，在 $m = 0$ 的时候，线程块内的线程被分为两个编号从 0 到 31 的线程组，当 $m = 1$ 的时候，线程块内的线程编号为一个 0 到 63 的线程组。通过不同编号的定义，使得线程可以进行不同的操作。图3.6展示了在不同分组情况下线程的操作流程和区别，在两个线程组分开的情况下，每组线程分别将全局显存中的数据通过 32 个线程依次加载到各自分管的共享内存中，且每个线程组仅访问各自分管的共享内存，即便这些数据是完全相同的。在两个线程组进行合并操作的情况下，两组线程合并为一个线程组，利用 64 个线程将数据从全局显存中连续加载到共享内存中，此时线程块中的线程从当前线程块的共享内存中加载数据。

显然 64 大小的线程组在一定程度上具有优势，然而，根据这样的优势而采用 64 大小的单一尺度任务组织的方法是不合理的。在空间中，粒子的分布具有随机性，存在很多任务中的粒子数远小于 32，这会使得存在较多的闲置线程。当任务

尺度大小设置为 64 的时候，这样的闲置线程将会更多。故此，为了不增加闲置线程的数量，采用 32 和 64 尺度大小的任务交错执行的策略，该策略能更好地适应空间中粒子分布的随机性，在很大程度上减少了资源浪费，线程组交错定位邻居空间的具体实现细节如算法2所示。

算法 2: 双重任务邻居空间定位算法

输入: 标记变量 m，当前子空间坐标 C，子空间粒子总数 N_c 及其偏移量 O_c

```
1  foreach 线程 do
2      根据公式3.2计算线程组序号 z_id
3      根据公式3.3计算线程的组内序号 t_id
4      if t_id < 9 then
5          t_p = 9 × z_id + t_id
6          根据 C 将线程 t_id 定位到对应的合并后的长方体邻居空间
7          o[t_p] ← 目标长方体邻居空间之前的粒子总数
8          n[t_p] ← 目标长方体邻居空间的粒子总数
9      end
10     进行线程同步操作
11     d = 9 × z_id × (1 − m)
12     c[t_id] = 9 + d    //初始化起始邻居查找区间
13     q[t_id] = 0    //初始化起始查找空间中已搜索的粒子数
14     i = 9 × z_id × (1 − m)
15     while i < 9 + d do
16         if n[i] ≠ 0 then
17             c[t_id] = i    //将第一个存在粒子的邻居空间设为起始位置
18             退出循环
19         end
20         i + +
21     end
22 end
```

在完成不同尺度的计算任务的邻居搜索空间的定位后，需要实现不同尺度的计算任务的邻居粒子数据的并行加载。具体的实现细节如算法3所示。

算法 3: 双重任务邻居粒子加载算法

输入: 标记变量 m, 目标粒子序号 id

```
1  foreach 线程 do
2      根据公式3.2计算线程组序号 z_id
3      根据公式3.3计算线程的组内序号 t_id
4      d = 9 × z_id × (1 − m)
5      while true do
6          if n[c[t_id]] == q[t_id] then
7              c[t_id] + +
8              q[t_id] = 0
9              if c[t_id] ≥ 9 + d then
10                 粒子加载完毕退出循环
11             end
12         end
13         ld = n[c[t_id]] − q[t_id]
14         if t_id < min(32 + 32 × m,    ld) then
15             pi = o[c[t_id]] + q[t_id] + t_id
16             将位于 pi 处的粒子的数据加载到共享内存
17         end
18         q[t_id] = q[t_id] + min(32 + 32 × m,    ld)
19         进行线程同步操作
20         if id < N_c + O_c then
21             根据 m 和 z_id 的值从共享内存中加载邻居粒子数据
22         end
23     end
24 end
```

3.3　混合算法调度策略

通过双重任务并行计算的方式，可以在不增加额外的闲置线程的情况下，有效地减少冗余数据的重复加载。然而，改进后的任务调度算法仍然存在问题，因为在进行一级任务划分的时候，在粒子数量较少的情况，依然按照一个计算任务划分会的方式会导致部分线程闲置，在空间中的粒子较为稀疏的时候，存在大量的计算任务中的粒子数小于 32，此时会有大量的闲置线程。在这种情况下，任务调度算法的计算效率将会低于传统的计算方式，其一是因为闲置线程数量较多，GPU 的并行效率没有得到充分发挥；其二是粒子稀疏的情况下，粒子之间耦合的数据量较少，共享内存的优势难以较好地发挥。阮在他的工作中也给出了相同的结果，即在粒子数稀疏的情况下，传统主流方法的计算效率高于单一任务调度方法。

为了解决任务中粒子数不足或粒子分布较为稀疏而产生的问题，本节给出了一种混合调度算法，即将传统主流方法和任务调度方法进行耦合形成混合方法。混合调度算法的核心是将粒子数较少的计算任务从任务队列中筛选出来，并且需要将这些任务中的粒子组织在一起，从而能够利用连续的 GPU 线程对其集中处理，从而较少闲置线程的数量和避免不必要的基于共享内存的数据加载；与此同时，任务队列也需要重新组织，以保证任务队列的连续性。为了解决这些问题，本节提出了一种新的针对 SPH 算法的哈希编码方法，通过新的哈希编码方法，可以有效地区别粒子数较少的计算任务和粒子数较多的计算任务，再通过对粒子哈希值的重排序，可以实现将稀疏的粒子从仿真空间中筛选出来。本节将对具体的实现细节进行详细介绍。

在固定网格方法中，空间中的子空间数量是固定的，故此，其对应的哈希值的最大值也是固定已知的；为了将稀疏的粒子从仿真空间中区分出来，新的哈希编码方法将哈希值的范围扩大到原来的两倍，具体的计算公式为：

$$H(\mathbf{P}(x,y,z)) = \begin{cases} \lfloor \frac{x}{h} \rfloor + \lfloor \frac{y}{h} \rfloor \cdot X + \lfloor \frac{z}{h} \rfloor \cdot X \cdot Y + M, & R = 1 \\ \lfloor \frac{x}{h} \rfloor + \lfloor \frac{y}{h} \rfloor \cdot X + \lfloor \frac{z}{h} \rfloor \cdot X \cdot Y, & R = 0 \end{cases}, \tag{3.4}$$

$\mathbf{P}(x,y,z)$ 代表粒子的空间位置；X 和 Y 分别代表 x 轴和 y 轴方向上立方体子空间

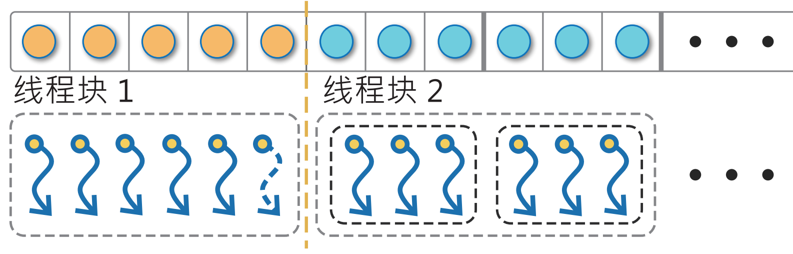

图 3.7: 混合算法示意图

的个数；M 是仿真空间中立方体子空间的个数；R 代表当前粒子所在子空间的稀疏性，0 代表稀疏，1 代表密集，其计算公式为：

$$R = \begin{cases} 1, & 32 \cdot B' \geq s \\ 0, & 32 \cdot B' < s \end{cases}, \tag{3.5}$$

B' 代表当前子空间中的计算任务的数量，不同于前面的任务计算方式，本节中每个子空间中计算任务的数量的计算公式为：

$$B' = \begin{cases} 0, & \overline{N} < TM \\ \lfloor (N_c + TD)/32 \rfloor, & \overline{N} \geq TM \end{cases}, \tag{3.6}$$

TD 是闲置线程的阈值，该值决定线程组中闲置的线程数量；$\overline{N}$ 是当前子空间和周围最近的六个子空间中粒子数的平均值；TM 平均粒子数的阈值，当 $\overline{N}$ 不小于这个阈值的时候进行任务划分，小于这个阈值的时候不进行计算任务划分。通过公式3.4、公式3.5和3.6可以计算出当前粒子新的哈希值，该哈希值考虑了目标粒子所处局部空间中粒子的稀疏性，同时也考虑了目标粒子在自身子空间中的次序。对于被认定为稀疏区域或传统任务划分策略中自身闲置线程过多的区域的粒子，其新的哈希值一定小于 M，而进行任务划分区域的粒子的哈希值一定不小于 M。所以，通过对新的哈希值的重新排序，可以将稀疏任务中的粒子从任务队列中被集中区分开来。

通过新的哈希编码方法和重排序，粒子被分为了稀疏粒子和密集粒子。如图3.7所示，稀疏粒子对应的线程块采用传统主流的计算方式，密集粒子对应的线程块则采用任务调度策略的计算方式。由于传统主流方法预处理过程较为简单，

本章计算方法的预处理过程较为复杂，在粒子极为稀疏的时候，预处理时间在总仿真时间中占比较大，本章计算方法的效率将低于传统主流方法，在这种情况将完全使用传统的计算方法，混合计算框架的整体实现如算法4所示。

算法 4: 混合框架算法

输入: 仿真场景的初始化的所有变量值

1 进行粒子空间插值
2 **repeat**
3 　//$\overline{NA}$ 是粒子密集度，TB 是粒子密集度的阈值
4 　**if** $\overline{NA} > TB$ **then**
5 　　进行计数排序预处理
6 　　通过公式3.6计算新每个子空间中的任务数量
7 　　通过公式3.5和公式3.4计算粒子新的哈希值
8 　　对新的哈希值进行第二次计数排序
9 　　确定稀疏粒子和密集粒子的边界
10 　　根据每个子空间的计算任务数进行任务队列的组织
11 　　**if** 线程块属于稀疏粒子 **then**
12 　　　每一个线程按照传统的计算方式计算目标粒子的物理属性
13 　　**else**
14 　　　线程块按照双重任务调度算法计算目标粒子物理属性
15 　　**end**
16 　**else**
17 　　完全执行传统主流方法的运算流程
18 　**end**
19 **until** 仿真结束;

3.4 实验分析

本节对本章节的工作进行了实验测试分析，采用的硬件平台为 Intel(R) Core(TM) i7-9750H 和 NVIDIA GeForce RTX 2070。本节采用的测试场景如图3.8所示，模拟

表 3.1: 多层级混合调度算法性能测试用例关键参数表

测试用例	粒子总数	粒子密集度	非空网格数	总网格数
用例 1	132651	5	29791	300763
用例 2	226981	8	29791	300763
用例 3	438976	15	29791	300763
用例 4	1030301	35	29791	300763
用例 5	3511808	118	29791	300763
用例 6	8365427	281	29791	300763
用例 7	27818127	934	29791	300763

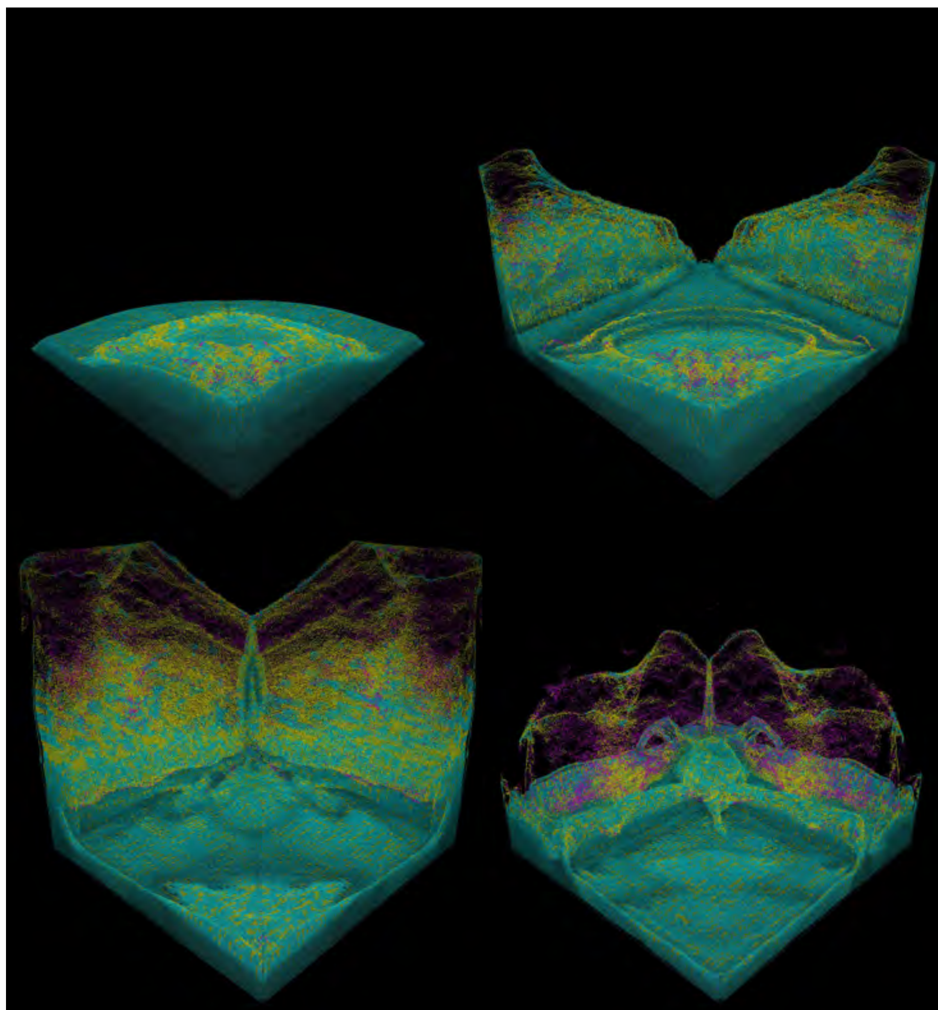

图 3.8: 测试场景任务层级展示效果图

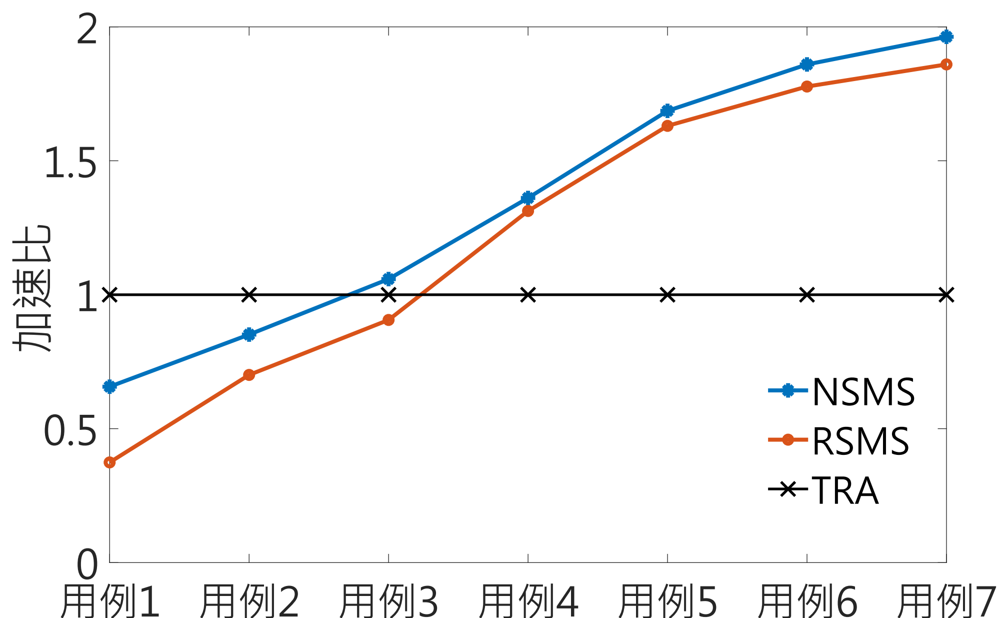

图 3.9: TB 等于 0 时多层级任务调度算法实验测试结果图

了水的溃坝效果。粒子的不同颜色体现了不通的计算方式，绿色的粒子采用的是二级任务的计算方式，黄色的粒子采用的是一级任务的计算方式，红色的粒子采用的是传统主流计算方法。从图中可以看出，在粒子密集的区域主要采用二级任务的计算策略，而在相对稀疏的区域采用的是一级任务的计算方式，对于离散程度较高的区域，传统主流方法起到主要作用。

本节测试用例的关键参数如表3.1所示，其中粒子密集度是 SPH 仿真中的关键参数，其对仿真性能的影响非常明显，粒子密集度越大，仿真所需的时间开销越大。同样，粒子的数量对仿真的时间开销影响也非常明显，粒子数量越多，仿真时间也越长。在这些测试用例中，仿真空间的大小相同，核半径的大小也相同，故此表中的网格总数也相同。因为每个用例中，进行初始粒子插值的空间范围相同，所以表中的非空网格数量也相同。测试用例中的粒子密集度从稀疏逐渐往密集变化，通过这些测试用例可以较为充分地反映系统在不同粒子密集度下的性能情况。

本节采用的实验对比对象是阮的单一任务调度算法 (这里称该方法为 RSMS) 以及传统主流方法 (这里称这种方法为 TRA)。在进行实验前，需要对本章的算法所涉及的阈值进行初始设定，$TD = 14$，$TM = 25$，$TB = 0$。实验结果如图3.9所示，

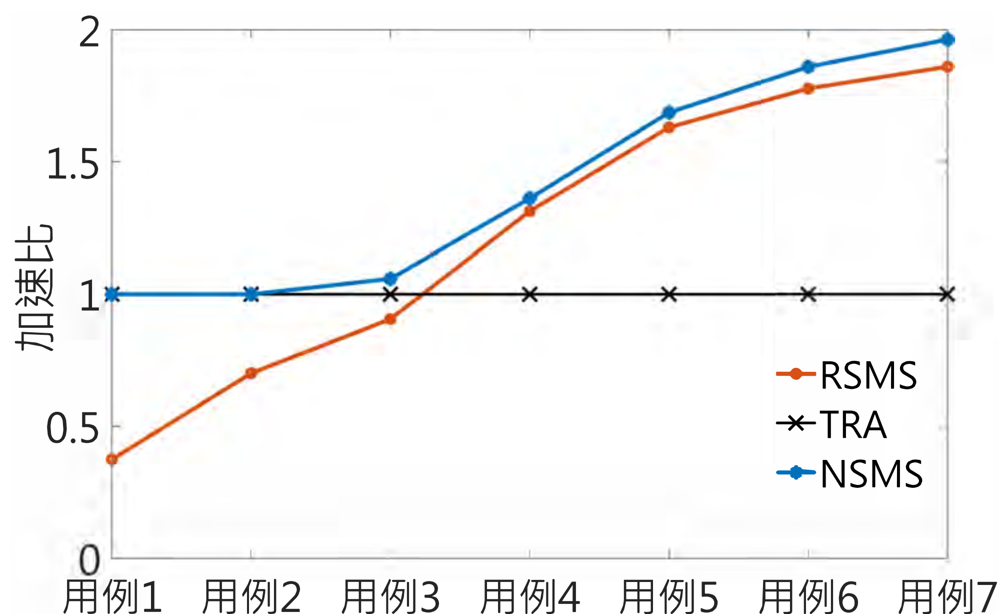

图 3.10: TB 等于 14 时多层级任务调度算法实验测试结果图

为了较好地体现不同方法间的性能差异，以 TRA 的测试结果为基准，对实验的测试结果进行单位化处理，故此 TRA 的加速比均为 1。用例 1 和用例 2 的测试结果表明，当粒子密集度很小的时候，TRA 的运行效率最高，由于在实验的时候，TB 设为 0，这意味着，算法4中的第 18 行在任何情况下都不会执行，根据测试用例的实验结果，本文将 TB 设置为 11，当再进行用例 1 和用例 2 的测试时，本章的算法 NSMS 的仿真效率和 TRA 相同，如图3.10所示。用例 3 的测试结果表明，NSMS 的计算效率相比 RSMS 提升明显，从粒子密集度可以看出，用例 3 的测试实验中，任务合并策略难以发挥作用，效率的提升主要是因为 NSMS 通过混合调度计算的方式减少了闲置线程的数量。在用例 4 和用例 5 的测试过程中，网格内的任务划分数量相对较小，且在该粒子密集度下，闲置线程的相对数量也较小，NSMS 中的优化策略难以发挥作用，但仍然比 RSMS 高效。在用例 6 和用例 7 中，网格中的任务数量较多，数据耦合度较大，NSMS 的任务合并策略可以发挥较大的作用，故此，NSMS 的加速比相比 RSMS 逐渐扩大。

由于本章提出的方法中还存在两个阈值常数 TD 和 TM，根据前面的测试分析本章中的 TD 设置为 14，为了选取较好的阈值常数 TM，本章进一步进行了不

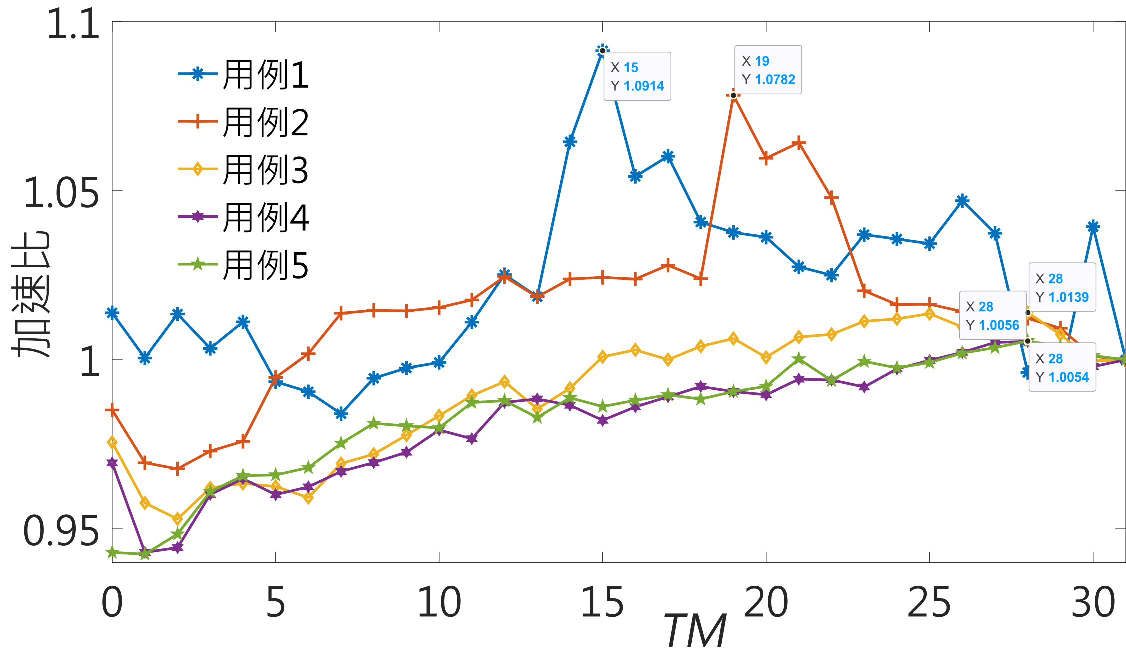

图 3.11: 闲置线程阈值参数测试结果图

表 3.2: 多层级混合调度算法阈值测试用例关键参数表

测试用例	粒子总数	粒子密集度	非空网格数	总网格数
用例 1	438976	15	29791	300763
用例 2	1030301	35	29791	300763
用例 3	1815848	61	29791	300763
用例 4	2460375	82	29791	300763
用例 5	3511808	118	29791	300763

同阈值条件下的性能测试，测试用例的关键参数如表3.2所示。针对每个测试用例，根据本章的方法在不同阈值下的进行帧率的帧率。以 $TM = 31$ 的测试结果为基准，计算各阈值条件下的相对加速比，测试结果如图3.11所示。在测试用例 1 上，最佳的阈值为 15；在测试用例 2 上，最佳的阈值为 19；在测试用例 3 到 5 上的最佳阈值均为 28；从中可以发现阈值的最佳取值和粒子密集度存在一定关系，在一定程度上，粒子密集度越大最佳阈值逐渐趋向于 28 左右。

3.5 本章小结

本章提出了一种针对基于 GPU 的 SPH 算法的调度策略，考虑到在计算不同粒子的物理参数的时候，粒子之间存在较多的数据冗余，设计了一种双重任务交错调度的方法，该方法在不产生额外的闲置线程的情况下，不仅减少了冗余数据的重复加载，同时提高了数据加载的效率。考虑到空间中粒子分布的随机性以及在不同粒子密集度下，不同算法之间的优缺点，设计了一种在不同计算方法之间进行混合调度的方法，该方法使得不同算法之间的优点在不同条件下得以发挥，从而减弱了单一计算方法的局限性，提高了算法的普适性。

第四章　SPH 算法的多层级邻居查找

不论是当前传统主流方法 TRA 还是阮的任务调度方法 RSMS，都是基于固定网格的计算方法。从图2.1中可以看出，固定网格方法中存在较多的无效邻居粒子，为了确定影响半径范围内的有效邻居粒子，需要计算邻居空间中邻居粒子的相对距离。根据粒子的影响半径，可以计算出粒子有效的影响空间大小为 $\frac{4\pi h^3}{3}$，而搜索空间的大小为 $27h^3$，故此有效搜索空间的占比为 $\frac{4\pi}{81}$，由此可以得到无效空间和有效空间的空间比例为 $1-\frac{4\pi}{81}:\frac{4\pi}{81}$，可见无效空间的空间占比较大，在空间粒子较为密集的情况下，固定网格方法涉及到较多的无效粒子数据的加载和计算，这对 SPH 算法的性能影响较大。为了解决这个问题，本章设计了一种新颖的多层级网格划分的方法，通过在不同层级上进行网格裁剪，有效地减小了邻居空间的范围，从而减少了无效邻居粒子数据的加载计算。由于更精细的网格划分，固定网格方法中粒子分布的连续性和一致性也得到提高，这在一定程度上也提高了任务调度算法中线程间的协作效率，本章将会对该方法进行详细的介绍。

4.1　多层级垂直空间网格

4.1.1　网格划分

当前存在较多的多层级网格划分方法，较为主流且应用较多的是八叉树结构的多层级空间网格划分方法，该方法在物理仿真中也得到了广泛的应用。在近些年的物质点方法 (MPM) 中，流行的稀疏网格也是一种八叉树结构的网格 [39] [40]，然而考虑到 SPH 算法与 MPM 等算法的差异，为了保证邻居查找过程中粒子数据的连续性，以及邻居子空间合并优化方法的应用，本文不采用类似八叉树网格结构的方法，而是提出了一种多层级垂直网格划分方法。

在三维空间中网格的划分方式如图4.1所示，对于第一层网格，本文采用固定网格的划的方式，图中为了表述方便，假设光滑核半径的长度是仿真空间边界长度

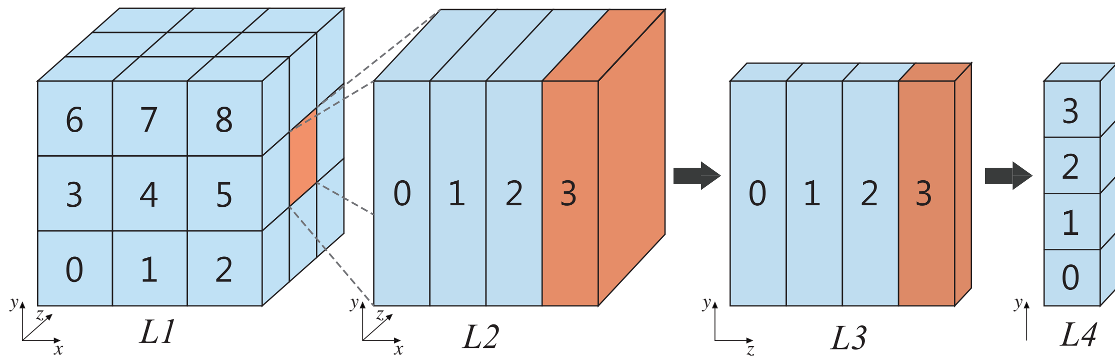

图 4.1: 多层级网格划分形式示意图

的三分之一；对于每一个一级网格，在 x 轴方向上进行垂直划分，将一级网格划分为等大的四个长方体子空间，至此完成二级网格划分；对于二级网格，在 z 轴方向上进行垂直划分，将二级网格划分为四个等大的长方体子空间，至此完成三级网格划分；对于三级网格，在 y 轴方向上进一步垂直划分，将网格划分为四个等大的立方体子空间，至此完成四级网格划分，本章中采用的是四级网格层次的划分方式。

4.1.2 数据编码

在完成网格的划分后，需要组织网格的编码方式。在八叉树结构的网格划分方法中，应用较多的编码方式是莫顿编码 [41] [42]，本节根据本章的网格划分方式为其设计了新的数据编码方法。对于一级网格，采用的是固定网格的编码方式，对于二级网格，则沿着 x 轴从 0 到 3 依次进行网格编码；对于三级网格和四级网格执行类似的编码方式，至此完成了网格在各自层级内的编码。除一级网格外，其余网格编码的数值范围均为 0 到 3，所以用两位二进制位就可以表示各层级的层内编码，为了进一步实现网格的全局编码，本文采用位操作的编码方法，具体编码过程如图4.2所示，先计算得到固定网格的编码值，再分别计算其他各层级的编码值，通过移位操作，得到最终编码值，作为不同空间区域的哈希值。

在明确仿真空间中各处的哈希值以后，需要确定粒子所处位置处的哈希值，从而完成粒子的哈希编码，位于 $\mathbf{P}(x, y, z)$ 的粒子与哈希值的对应关系为：

$$H(\mathbf{P}) = (I(\mathbf{P}) << 6) + I_c, \tag{4.1}$$

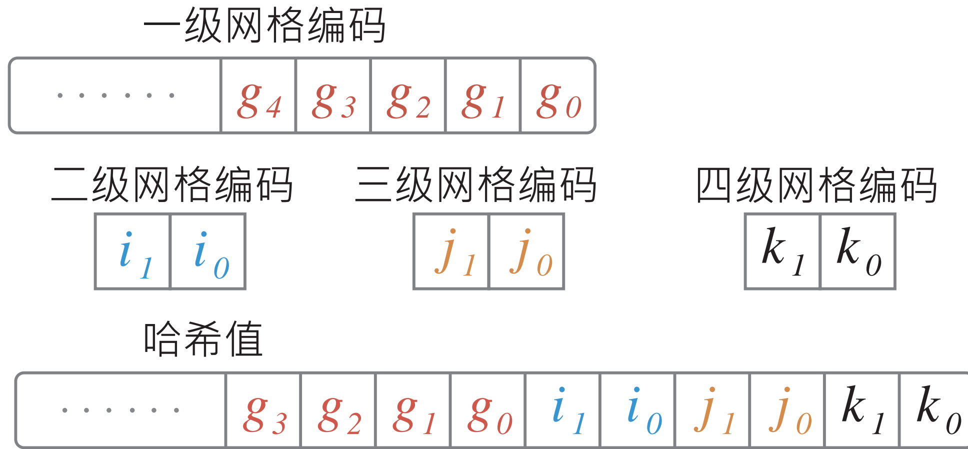

图 4.2: 多层级网格编码方式示意图

I 是粒子所处的一级网格的编码值，其具体的计算公式为：

$$I(\mathbf{P}(x,y,z)) = \left\lfloor \frac{x}{h} \right\rfloor + \left\lfloor \frac{y}{h} \right\rfloor \cdot X + \left\lfloor \frac{z}{h} \right\rfloor \cdot X \cdot Y, \tag{4.2}$$

I_c 是粒子在一级网格中的相对位置 $\mathbf{P}_c(x_c,y_c,z_c)$ 对应的编码值，计算公式为：

$$I_c(\mathbf{P_c}(x_c,y_c,z_c)) = \left\lfloor \frac{y_c}{h_c} \right\rfloor + \left\lfloor \frac{z_c}{h_c} \right\rfloor << 1 + \left\lfloor \frac{x_c}{h_c} \right\rfloor << 2, \tag{4.3}$$

h_c 是四级网格的边长，其值为 $h/4$。因为粒子所处位置处的哈希值中包含了一级网格编码值，同时也包含了其余三个层级网格的编码值，通过移位操作可以还原出各层级的编码信息，从而实现粒子在不同层级间的空间定位，为不同层级间的邻居查找策略奠定基础。

在多层级网格中完成粒子的哈希编码后，通过对粒子哈希值的排序，粒子在存储空间中的连续性如图4.3所示。在四级网格中，粒子数据的分布依然具有随机性，在三级网格中，每个四级网格的粒子数据在 y 轴方向上连续分布，在二级网格中，三级网格的粒子数据在 z 轴上连续分布，在一级网格中，二级网格的数据在 x 轴上连续分布；每一个一级网格首先在 x 轴上连续分布，而后相同的分布方式沿着 y 轴逐层推进，进而同样的分布方式在 z 轴上逐层推进直至充满仿真空间。相比固定网格方法，该方法中的粒子仅在四级网格中随机分布，而固定网格方法中的粒子是在一级网格中随机分布。所以，该网格细分方法在一定程度上提高了空间中粒

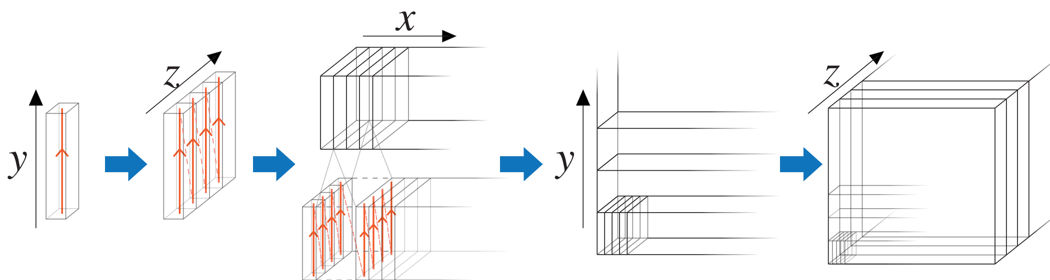

图 4.3: 粒子数据连续性示意图

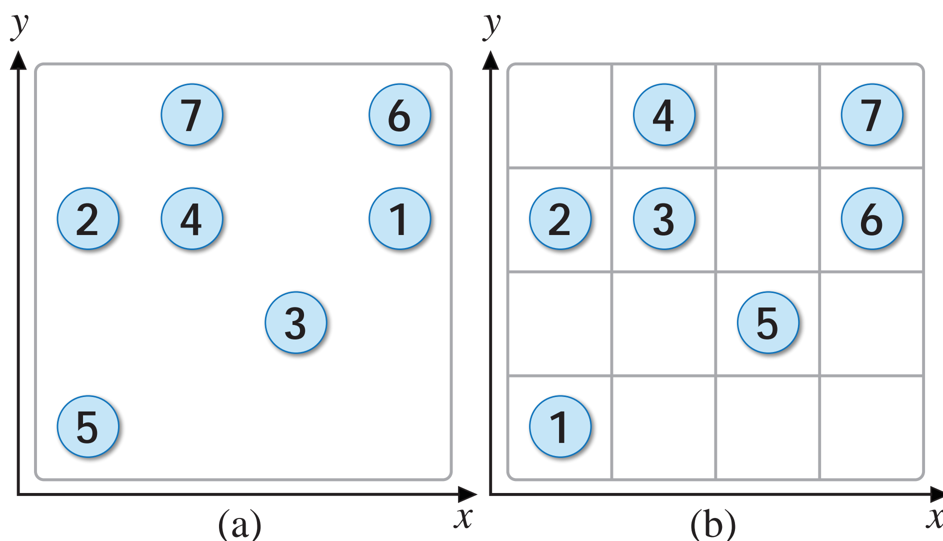

图 4.4: 粒子数据一致性对比示意图

子数据分布的一致性，图4.4展示了二维空间中固定网格中的粒子分布和本节方法中的粒子分布，细分后网格中粒子的空间序号和其相对位置的一致性要高于固定网格方法。

针对任务调度算法而言，同一个计算任务中的粒子必定来自同一个子空间，所以同一计算任务的粒子共享同一块邻居搜索空间。然而，同一计算任务的粒子不一定拥有完全相同的有效邻居粒子，一方面是因为各个粒子之间存在距离，另一方面是因为线程块中粒子分布的随机性。图4.5展示了两个计算任务中不同粒子的有效邻居空间，在计算任务 t_1 中，深蓝色粒子的有效邻居粒子是三个浅蓝色粒子，然而这三个浅蓝色粒子并不是计算任务中其余粒子的有效邻居粒子，在加载完这些粒子进行求解计算的时候，只有深蓝色粒子对应的线程在执行计算操作，而其余线

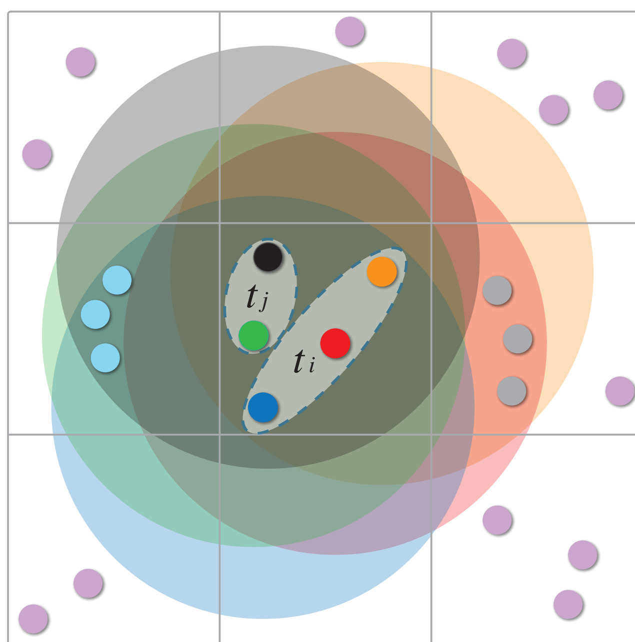

图 4.5: 粒子有效邻居空间示意图

程全部处于闲置状态；同理，在其余线程进行计算的时候，深蓝色粒子对应的线程处于停滞等待状态。如果采用本节的网格划分方法，蓝色粒子将会并入任务 t_2 中，图中的线程交错等待问题在一定程度上得到解决。

4.2 邻居查找和空间缩减

在完成多层级垂直网格划分和粒子的编码排序后，可进行邻居空间范围的缩减以及邻居粒子的查找。本节将具体介绍不同层级上的网格裁剪方法以及邻居粒子的查找策略。在介绍网格裁剪方法前，需要先进一步介绍粒子的连续性。如图4.6所示，在 x 轴方向上，一级网格中的粒子数据是完全连续的，在 x 轴方向上的网格裁剪不影响剩余搜索空间中粒子数据的连续性；在 z 轴方向上，粒子数据仅在一级网格内连续，在 z 轴方向上进行网格裁剪将会使得位于 z 轴两侧的一级空间网格内的数据连续性被破坏；在 y 轴方向上，粒子数据亦只在一级网格内连续，在 y 轴方向上进行网格裁剪将会使得位于 y 轴两侧的一级空间网格和二级空间网格数据的连续性同时被破坏。

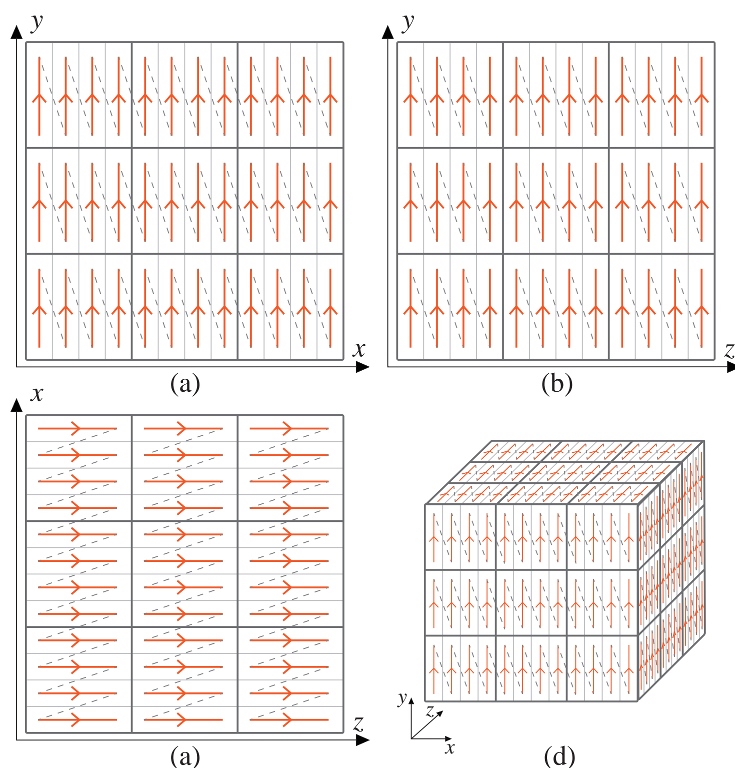

图 4.6: 三维空间各维度视角下数据连续性示意图

如上所述，一级网格的空间划分方式和固定网格方法相同，在进行邻居空间范围缩减的时候，一级网格的作用也同固定网格方法，即将搜索的邻居范围限定在 27 个一级网格空间中，此处不再赘述。为了实现较为通用的邻居限定方法，考虑的目标对象是单个计算任务，单一的粒子可以视为特殊的计算任务，即一个计算任务内只有一个粒子的情况。为了实现在二级网格上进行邻居空间范围缩减，首先需要明确当前计算任务所处的二级网格范围，根据粒子的连续性可知，粒子的二级网格范围取决于计算任务中的第一个粒子和最后一个粒子所处的二级网格位置，第一个粒子所处的二级网格即在 x 轴方向上最左侧的二级网格，最后一个粒子对应的二级网格即在 x 轴方向上最右侧的二级网格。在得到计算任务的二级网格范围后，则可进行邻居空间范围缩减，为了保证核半径范围内的邻居粒子均处于缩减后的邻居空间，应当根据计算任务的二级网格范围沿着 x 轴方向进行左右缩减。核半径长度等于一级网格长度，故核半径长度范围内存在四个二级网格，根据计算

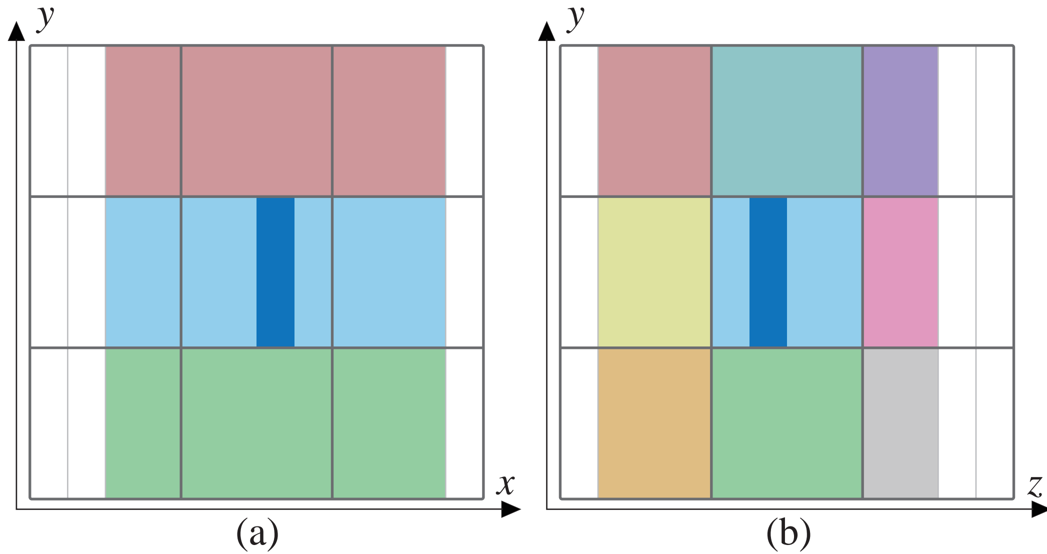

图 4.7: 邻居范围缩减示意图

任务所处的二级网格范围分别向左和向右扩展四个二级网格，当超过一级网格限定的范围时，以一级网格为准，三级网格的限制方法与二级网格的限制方法类似。

图4.7展示了在二级网格和三级网格上的空间缩减示例，深蓝色的长方体网格代表当前计算任务所处的网格空间。a 图中，根据二级网格的范围进行空间扩展实现二级网格的范围缩减，图中的颜色代表缩减后的邻居空间中数据的连续性，如上所述，x 轴方向上的网格裁剪不影响邻居空间中粒子分布的连续性，邻居空间的合并策略依然可以使用，27 个一级网格空间依然可以合并为 9 个较大的子空间 (二维仿真中是三个长方体子空间)。b 图展示了三级网格上的邻居范围缩减，缩减后的结果与 a 图类似，不同的是，z 轴方向上的网格裁剪破坏了二级网格中粒子数据的连续性，相应的一级网格不再适用空间合并策略，此时需要逐个访问被裁剪后的二级网格中的数据。由于二级网格的邻居范围限制，减少了邻居范围中二级网格的数量，从 a 图可知，在 x 轴方向上，邻居范围中单个由一级网格合并的长方体网格中的二级网格的数量为 9，所以，在三级网格上进行邻居空间缩减的区域，原本可以连续加载的粒子数据，需要分九次从被裁剪后的二级网格空间中依次加载。这在一定程度上增加了邻居粒子加载所需的循环迭代的次数，在一定程度上影响算法的效率。但在粒子数量较为密集的时候，在三级网格上的邻居空间缩减仍然

可以起到加速作用。在进行四级网格上的邻居空间缩减时，y 轴方向上的网格裁剪对数据连续性的破坏较为严重，会显著增加算法中循环迭代的次数，减少的粒子数又相对较少，故此本节的邻居空间缩减方法仅作用在前三级网格上，第四级网格的划分主要是为了提高空间中粒子分布的一致性。

如上所述，邻居空间缩减方法的关键是准确定位粒子所处的各级网格位置，根据哈希值的编码方式,通过位操作可以很容易得到相应的位置信息。对于一级网格，其网格编码值 I 的计算公式为：

$$I = H >> 6, \tag{4.4}$$

对于二级网格，其网格编码值 I' 的计算公式为：

$$I' = (H\&24) >> 4, \tag{4.5}$$

对于三级网格，其网格编码值 I'' 的计算公式为：

$$I'' = (H\&6) >> 2. \tag{4.6}$$

在计算得到粒子所处各级网格的编码值后，需要确定计算任务所处的各级网格的编码范围，如上所述计算任务的范围可以通过计算任务中的第一个粒子和最后一个粒子所处的网格位置确定。具体的计算细节如算法5所示。

在进行数据加载的过程中，为了能够连续加载某一层级网格内的数据，需要获取该网格的粒子数偏移量以及粒子总数。在针对本节的哈希值进行计数排序的过程中，各哈希值的粒子数队列 $\mathbf{N}_c$ 以及偏移量队列 $\mathbf{O}_c$ 可以视为是对四级网格进行统计计算的结果，而四级网格的连续排列构成了其他各层级的网格，因此，各层级网格的粒子数和偏移量可以通过数组 $\mathbf{N}_c$ 和 $\mathbf{O}_c$ 计算得到。以一级网格为例，其他各层采用类似的计算方式，一级网格 C_I 的统计量的计算公式为：

$$\begin{cases} \mathbf{N}_I[C_I] = \mathbf{O}_c[64\cdot(C_I+1)] - \mathbf{O}_c[64\cdot C_I] \\ \mathbf{O}_I[C_I] = \mathbf{O}_c[64\cdot C_I] \end{cases}. \tag{4.7}$$

算法 5: 计算任务网格范围计算算法

```
foreach 计算任务 do
    pid1 ← 计算任务第一个粒子的哈希值
    pid2 ← 计算任务最后一个粒子的哈希值
    通过公式4.4计算当前计算任务的一级网格序号
    lx ← 通过公式4.5计算第一个粒子二级网格序号
    rx ← 通过公式4.5计算最后一个粒子二级网格序号
    zz1 ← 通过公式4.6计算第一个粒子三级网格序号
    zz2 ← 通过公式4.6计算最后一个粒子三级网格序号
    if lx == rx then
        lz = zz1    //lz 为三级网格范围的左边界
        rz = zz2    //rz 为三级网格范围的右边界
    else
        lz = 0
        rz = 3
    end
end
```

4.3　多层级网格数据加载方法

根据邻居范围的缩减方法 (网格裁剪方法) 和相应的邻居粒子的加载策略，在进行空间缩减的过程中，涉及到不同连续性的粒子数据的加载，本节给出了一种动态多层级的邻居查找策略，通过该策略，可以减少无效邻居粒子的加载计算，尽可能避免增加无效的循环迭代计算，保证空间数据加载的连续性。本节将对动态多层级邻居查找方法进行详细介绍。

因为本章中的方法仅在 x 和 z 两个坐标轴方向上进行空间裁剪，为了说明方便，本节主要基于 xz 平面进行说明。如上所述，仅在 x 轴方向上进行网格裁剪是不影响剩余空间中粒子数据原本的连续性，故此，存储合并策略依然适用。如图4.8中的 a 图所示，在进行 x 轴方向上的网格裁剪以后，剩余的空间依然可以合并为三个

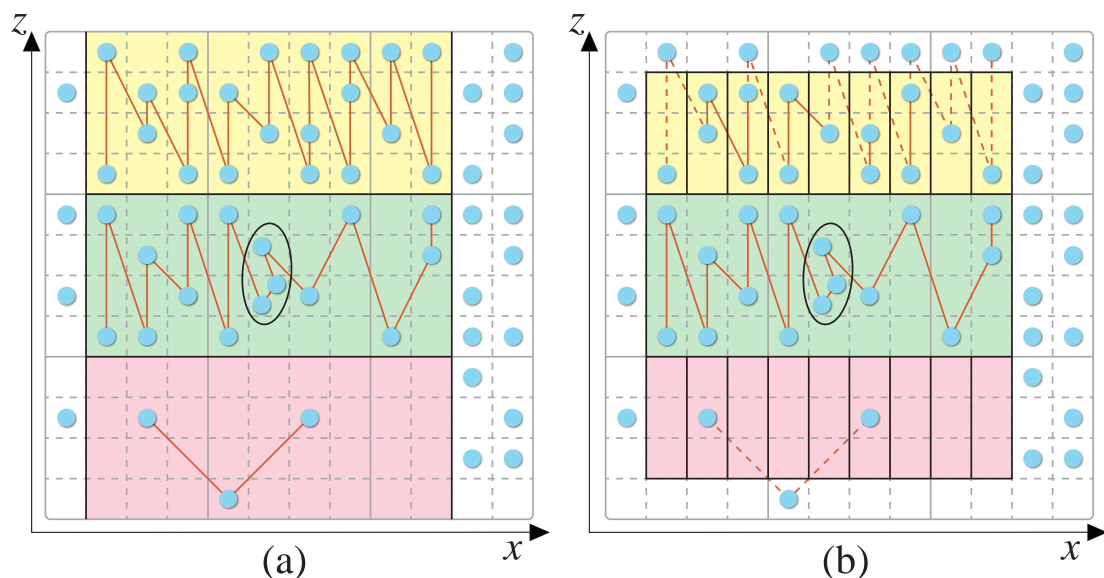

图 4.8: 各层级邻居粒子数据加载连续性示意图

较大的邻居搜索空间。在每个合并后的邻居空间中，粒子之间数据的连续性用粒子之间的连线进行表示，在进行邻居数据加载的时候，可以在较少的空间分区中连续加载数据。当在 z 轴方向上也进行网格裁剪后，如 b 图所示，因为部分粒子被从搜索空间中剔除，导致部分连续的数据发生中断，使得数据的连续性被破坏。由于被剔除的粒子存储位置未知，难以确定数据具体是从哪个粒子开始中断，所以，为了能够准确加载剩余的有效邻居粒子数据，需要依次连续加载每个重新划分后的小长方形子空间中的数据。在 z 轴方向上未被进行网格裁剪的邻居空间，其粒子数据的连续性未发生变化，对于这部分邻居空间，作为一个邻居空间进行连续的数据加载。相比在每一个四级网格中依次加载邻居数据的邻居空间缩减方法，本章的方法充分利用了数据的连续性，减少了在搜索空间中进行切换所需的循环迭代次数。

以图4.8中 b 图红色区域的搜索过程为例，为了减少一个无效的邻居粒子，需要在九个子空间中进行循环迭代，从而加载有效邻居粒子的数据。显然，此时三级网格的邻居空间缩减方法不适用，如果在红色区域上只使用二级网格的邻居缩减方法，在增加一个无效粒子数据加载计算的代价下，可以得到减少 8 次网格间的循环迭代次数以及连续加载粒子数据的优势，采用二级网格的邻居空间缩减方法更为合理。故此，如图4.9所示，本章最后采用一种动态网格裁剪的方式：在完成二

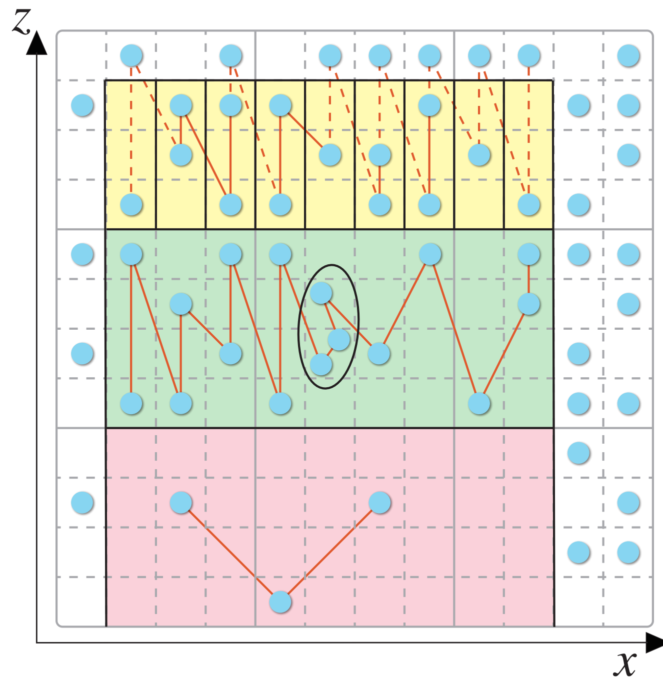

图 4.9: 动态三级网格裁剪示意图

级网格裁剪的基础上，根据合并空间中粒子的分布情况决定是否进一步执行三级网格裁剪策略。具体的实现过程如算法6所示。

算法 6: 计算任务网格范围计算算法

foreach 合并邻居空间 **do**
  根据当前任务二级空间范围进行二级网格裁剪
  $N_h \leftarrow$ 二级网格裁剪后的有效空间的粒子总数
  //TA 是粒子密集度关于三级网格裁剪的阈值
  if $\overline{NA} > TA$ ***And*** 当前空间可进行三级网格裁剪 **then**
    根据计算任务的三级空间范围进行三级空间裁剪
    if 裁剪后的空间存在粒子 **then**
      记录裁剪后二级空间的层数
    else
      标记该空间作为无效空间
    end
  end
end

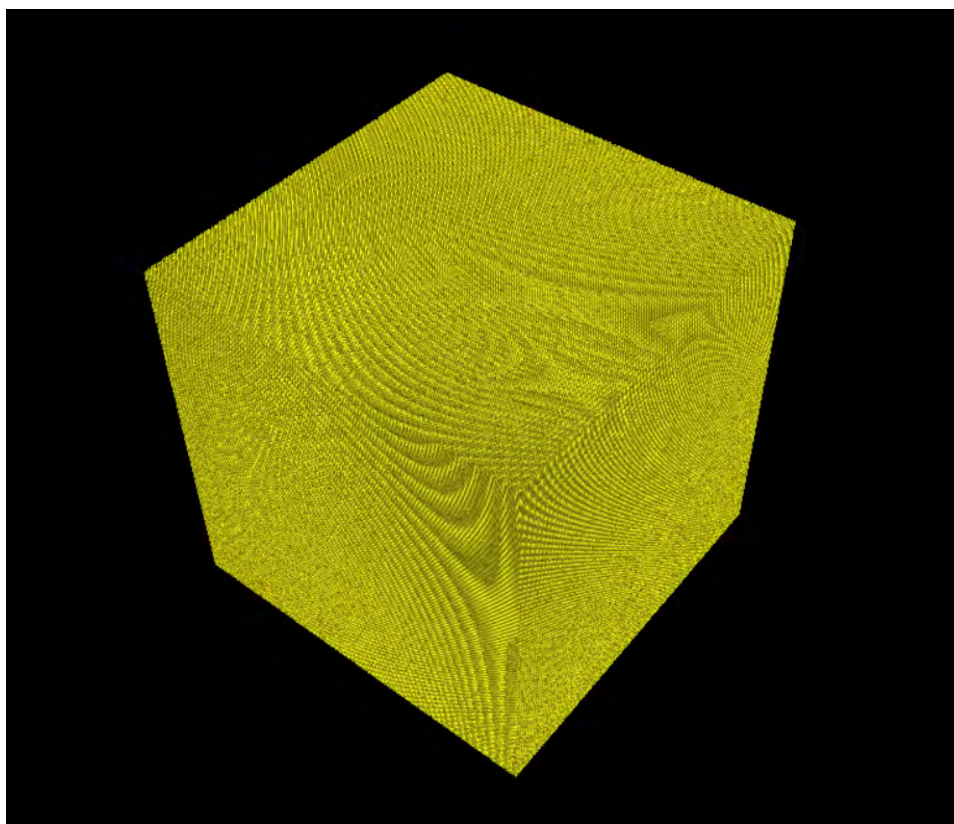

图 4.10: 参数测试场景效果图

4.4 实验分析

本节将重点介绍针对多层级邻居查找算法的实验测试情况，首先为了给阈值 TA 设定合适的参数值，本节需要在不同的粒子密集度下对仅采用二级网格裁剪方法和进一步使用三级网格裁剪方法这两种情况进行测试分析。为了保证在仿真过程中粒子密集度能够保持恒定，关于阈值 TA 的测试实验不进行粒子空间位置的更新计算。测试用例的关键参数如表4.1所示，考虑到邻居范围缩减方法适用于一级网格中粒子数较多的情况，本章实验不测试粒子数稀疏的情况。

本节的对比对象和改进对象是阮的单一任务调度算法 RSMS，实验场景是基于溃坝场景的仿真算法省去粒子位置更新计算的步骤得到，场景如图4.10所示。该测试场景的相关参数易于调整，相关变量易于控制，适合进行参数测试分析。在进行比较的时候，本节亦采用第三章实验部分的单位化后的比较方式，基于原本的

表 4.1: 邻居空间缩减测试用例关键参数表

测试用例	粒子总数	粒子密集度	非空一级网格数	一级网格总数
用例 1	1030301	35	29791	300763
用例 2	1367631	46	29791	300763
用例 3	1815848	61	29791	300763
用例 4	2460375	82	29791	300763
用例 5	3511808	118	29791	300763
用例 6	5268024	177	29791	300763
用例 7	8365427	281	29791	300763

RSMS 的测试结果，对集成二级网格裁剪和进一步集成三级网格裁剪的 RSMS 的测试结果进行单位化。测试结果如图4.11所示，L1 代表仅使用二级网格裁剪的改进算法，L2 代表进一步使用三级网格裁剪的改进算法；在用例 1 的测试结果中，L1 和 L2 的计算效率低于 RSMS，这是因为场景中的粒子相对较少，邻居空间缩减后过滤的无效邻居粒子数相对较少，而改进算法增加的预处理时间超过优化后数值求解计算中节约的时间，导致性能下降；在用例 1 到用例 4 的测试结果中，L1 的效率均高于 L2，这说明粒子密集度没有达到一定阈值的时候，进一步采用三级网格裁剪会导致的循环迭代所需的开销大于实际的性能收益，根据实验结果，本文设定 TA 的值为 90。

在实验分析得到阈值 TA 后，则可对基于动态三级网格裁剪的多层级邻居粒子查找算法进行真实仿真测试分析，测试的实验用例依然采用表4.1中的测试用例。测试结果如图4.12所示，L3 代表的是基于 RSMS 使用动态三级网格裁剪方法的改进算法，和图4.12相比，L1 和 L2 在用例 1 上相对 RSMS 的加速比都有下降，尤其是 L2 的下降幅度较为明显，这是因为在真实的仿真场景中，粒子的移动会导致仿真过程中粒子密集度下降，而粒子密集度的降低会进一步突出三级网格裁剪的缺点，这也是 L2 在用例 5 上的加速比小于 L1 的原因；在整体的测试结果中，L3 在用例 1，用例 2 和用例 3 上的加速比和 L1 基本接近，这是因为在该测试用例上，L3 通过阈值的判断，基本采用 L1 的计算方式；从用例 4 开始，L3 的加速比均高

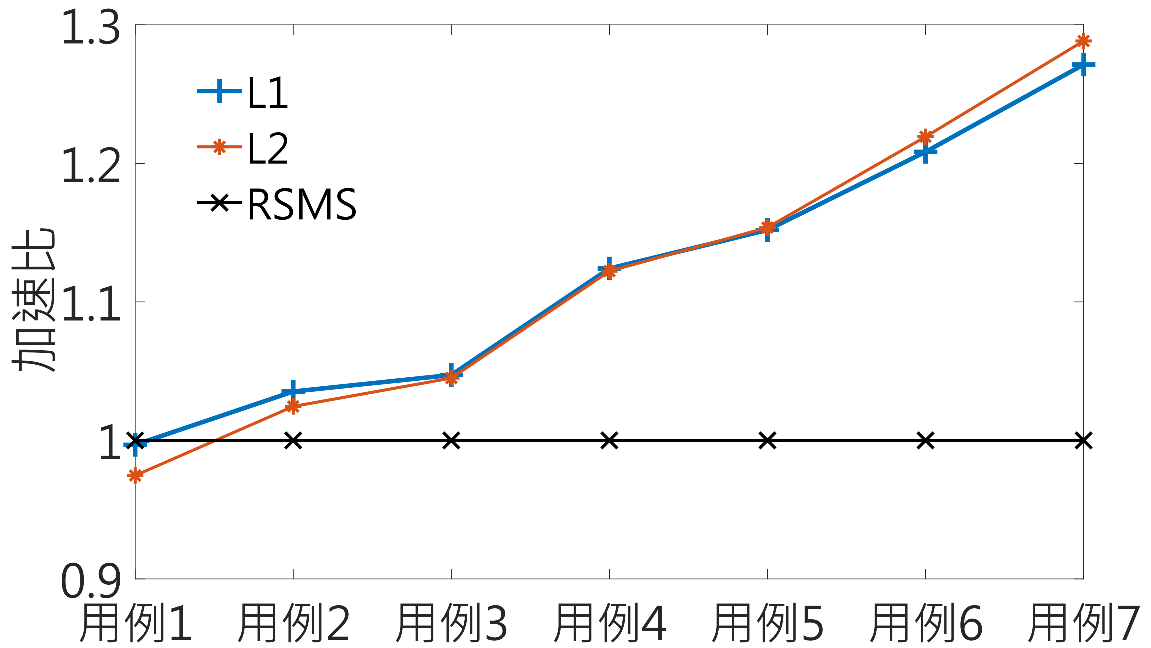

图 4.11: 不同层级网格裁剪方法实验测试结果图

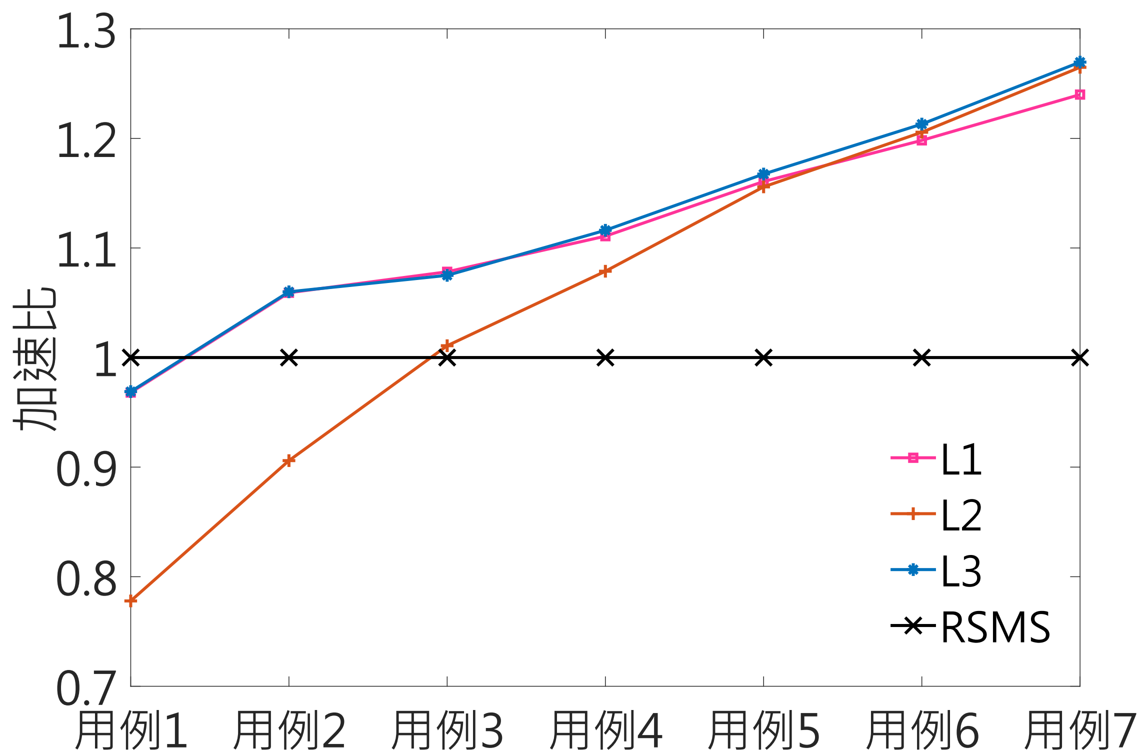

图 4.12: 动态网格裁剪方法实验测试结果图

于 L2 和 L3，这是因为 L3 可以根据不同网格中的粒子分布动态采用 L1 和 L2 的计算方式，使得整体的计算性能有所提升。测试结果表明，动态三级网格划分的方式进一步使得多层级邻居查找方法更具普适性和高效性。

4.5　本章小结

本章提出的多层级网格查找算法的核心目标是减少无效的邻居粒子数据的加载计算，并且尽量避免过多的网格间的循环迭代次数。为此，本章基于数据连续性的考虑，设计了一种多层级垂直网格划分的方法，并根据该网格的划分特点设计了相应的网格编码方法。基于本章的网格划分方法，又进一步设计了一种在多层级的网格上进行邻居查找的算法，从而达到减少无效邻居粒子，且尽量避免过多的循环迭代次数的目的。最后，通过实验测试分析，表明了本章方法的高效性。

第五章 基于混合加速架构的流体仿真

本章对第三章和第四章提出的加速算法进行了合并，从而实现了一个针对 SPH 算法，具有良好通用性和高效性的加速框架，同时为了进一步提高仿真框架的仿真能力，在该仿真框架中集合了多种 SPH 算法，从而使得仿真框架能够进行多种物理现象的仿真计算。通过对多种不同的物理现象的仿真测试，对框架的仿真计算效率进行了性能测试分析。

5.1 混合加速框架和效率分析

如上文所述，多层级任务调度算法是基于固定网格方法设计实现的，为了在多层级网格方法中实现多层级任务调度算法，关键是明确进行任务划分的空间层级，以及统一两种算法中的哈希编码方式。

在多层级任务划分算法中，任务划分是基于固定网格方法中子网格的粒子数进行划分，而固定网格的子网和多层级垂直网格的一级网格一致。故此，为了实现一级任务的划分，关键是确定每个一级网格中粒子的数量，可根据公式3.2进行计算。在计算得到每个一级网格的粒子数量 N_I 和粒子偏移量 O_I 后，结合多层级网格方法中的每个四级网格的偏移量 O 以及每个四级网格中的粒子的相对偏移量 s，可以计算出一级网格中每个粒子在一级网格中的相对偏移量，计算公式为：

$$s_I = O - O_I + s_I. \tag{5.1}$$

完成一级网格中粒子的相对偏移量计算后，则得到了公式3.4、公式3.5和公式3.6中所需的全部变量值，至此可以实现在多层级垂直网格中实现多层级任务调度算法。根据第四章的实验测试分析可知，在粒子密集度较小的时候，考虑到预处理时间对算法效率的影响，不适合使用网格裁剪方法，故此，为了保证系统框架的普适性，当初始粒子密集度小于一定阈值 TF(根据第四章的测试结果，本文将 TF 设定为 40) 的时候，完全使用第三章的多层级任务调度算法的进行计算，本文总体框架

的计算逻辑如算法7所示。

算法 7: 整体框架算法

输入: 仿真场景的初始化的所有变量值

```
1  进行粒子空间插值
2  repeat
3      if NA > TF then
4          if NA > TB then
5              根据多层级网格编码方法进行计数排序预处理
6              通过公式5.1和公式3.6进行任务划分
7              通过公式3.5和公式3.4计算粒子新的一级网格编码值
8              对新的一级网格编码值进行第二次计数排序
9              确定稀疏粒子和密集粒子的边界
10             根据每个子空间的计算任务数进行任务队列的组织
11             确定每个任务的二级网格和三级网格的编码范围
12             结合网格裁剪方法和算法4进行求解计算
13         else
14             完全执行传统主流方法的运算流程
15         end
16     else
17         使用算法4进行求解计算
18     end
19 until 仿真结束;
```

（第3行条件：$\overline{NA} > TF$；第4行条件：$\overline{NA} > TB$）

在得到整体的加速框架后，需要进行相关的性能分析测试。为了充分说明框架的有效性，本章在不同架构的 GPU 平台上对系统框架进行了相关的性能测试，本章采用的 GPU 有 GTX 970(麦克斯韦架构)，GTX 1080(帕斯卡架构) 和 RTX 2070(图灵架构)。本章的实验对比对象是阮的单一任务调度算法 RSMS，首先探究的内容是本文的仿真框架相对 RSMS 的加速比与粒子数量的关系，采用的初始粒子密集度为 297。为了保证粒子密集度的恒定，以达到控制变量的目的，关于粒子总数的

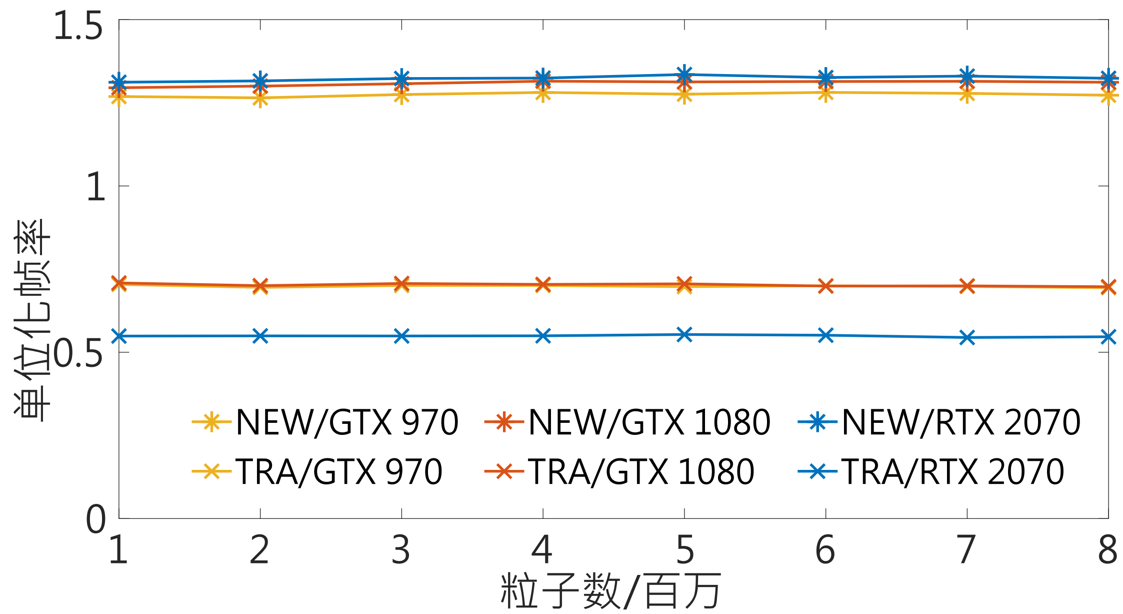

图 5.1: 整体框架加速效率与粒子数关系测试结果图

表 5.1: 粒子密集度测试用例关键参数以及测试结果统计表.

GPU		粒子数	粒子密集度	TRA			RSMS			NEW		
				Pre	Dfor	Tot	Pre	Dfor	Tot	Pre	Dfor	Tot
GTX 970	用例 1	8,000,000	64	8.9	510.3	519.3	13.5	331.4	344.9	15.0	326.3	341.3
	用例 2		125	9.2	990.7	999.9	14.1	665.1	679.2	15.0	576.4	591.4
	用例 3		297	10.0	2315.2	2325.2	15.6	1626.0	1641.6	15.7	1264.4	1280.1
GTX 1080	用例 1		64	8.2	227.7	235.9	10.8	147.2	158	11.6	139.7	151.3
	用例 2		125	7.9	439.9	447.8	11.1	293.6	304.7	11.8	246.1	257.9
	用例 3		297	8.3	1015.6	1023.9	12.0	705.3	717.3	12.3	532.5	544.8
RTX 2070	用例 1		64	15.9	247.6	263.5	18.5	122.4	140.9	16.6	121.0	137.6
	用例 2		125	15.8	501.7	517.5	18.8	262.0	280.8	17.6	220.1	237.7
	用例 3		297	18.3	1171.9	1190.2	21.2	632.8	654.0	17.2	475.0	492.2

测试实验也不更新粒子的空间位置。实验结果如图5.1所示，为了更好地体现不同框架的相对速率，本章以 RSMS 的帧率为基准，对本文框架以及其他加速框架进行单位化处理。图中的 NEW 代表的是本文的框架，TRA 代表传统主流方法。在不同粒子总数下，系统的加速比在三种不同的硬件平台上基本保持稳定，由此可见，本文的系统框架相对 RSMS 的加速情况受粒子总数的影响较小，同时也可以看出本文的框架相对传统主流方法的加速效率更为明显。

在得到粒子总数对系统加速效率影响较小的结果后，本节通过固定粒子的位置进一步测试粒子密集度对本文整体框架加速效率的影响。表5.1展示了本文加速

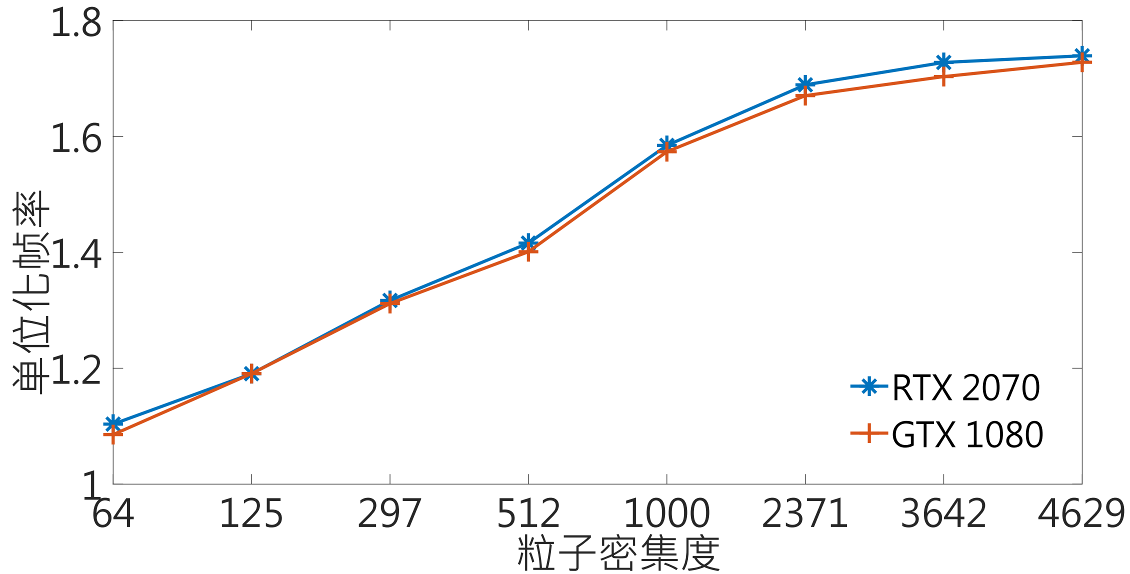

图 5.2: 真实物理仿真整体框架与粒子密集度关系性能测试结果图

框架在不同测试用例以及不同 GPU 上的性能测试结果，**Pre** 代表算法单帧的预处理时间，**Dfor** 代表算法单帧求解粒子受力所需的时间，**Tot** 代表算法单帧总时间。从中可以看出粒子密集度越大，系统的加速比越高；传统主流方法在 GTX 1080 上的运算效率比 RTX 2070 的运算效率高，这是因为测试所用的 GTX 1080 的显存的访问效率比 RTX 2070 要高；GTX 970 的运算效率在所有的测试结果中均比其他两种显卡低；基于共享内存的方法在所有的测试结果中，运算效率均比传统主流方法高。根据表5.1中的实验结果，在后面的实验中，只对 RSMS 和本文的框架进行测试，测试平台为 GTX 1080 以及 RTX 2070。

固定粒子的位置只能反映在特定粒子密集度下框架的加速效率，而真实的仿真场景中，随着粒子的移动，场景中的粒子密集度常常会随之改变。所以，考虑到真实仿真场景中粒子密集度的变化性，本节进一步探究了整体框架在真实仿真场景中的加速效率，测试结果如图5.2所示，框架在不同 GPU 上的加速效率和初始粒子密集度存在正相关关系，可以达到近 1.8 倍的效率提升。由于更大的粒子密集度在真实的仿真场景中并不多见，故此本章仅给出了 0 到 5000 之间的粒子密集度的测试结果，如若进一步加大测试场景的初始粒子密集度，系统的加速比可以进一步提高，最大可以达到接近两倍的效率提升。

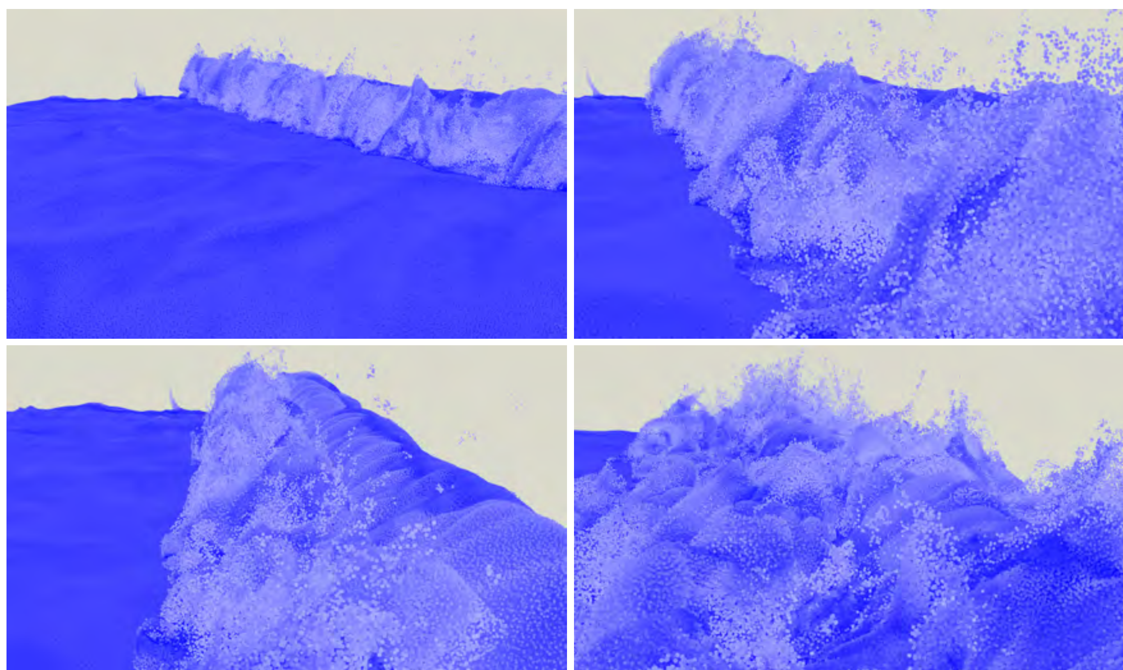

图 5.3: 海浪仿真效果图

根据算法7的运算流程可知，当粒子密集度较低的时候，本文的系统框架会采用第三章中的多层级任务调度算法进行相关的求解计算，第三章的实验部分已经给出了较低的粒子密集度下的多层级任务调度算法与 RSMS 的实验对比结果，从实验结果中可知，在更低的粒子密集度下，本文计算框架的计算效率要高于 RSMS。由此可见，本文的整体计算框架在实际的应用中具有更好的普适性和高效性，考虑到本文的加速框架未对 SPH 算法的数值方法进行简化，本文的框架可直接应用于 SPH 算法的各种实际应用中，具有较好的实际应用价值。

5.2　物理仿真

为了充分说明系统框架整体加速的效率，本节在整体的仿真框架上进行了一系列复杂的物理场景的仿真，并在本文的框架上对多种仿真算法的运算效率进行了较为细致的测试分析。仿真的场景主要包括海浪的仿真，液体模型下落的仿真，高速流仿真，流体耦合仿真，弹性固体仿真以及弹性固体和液体交互仿真。

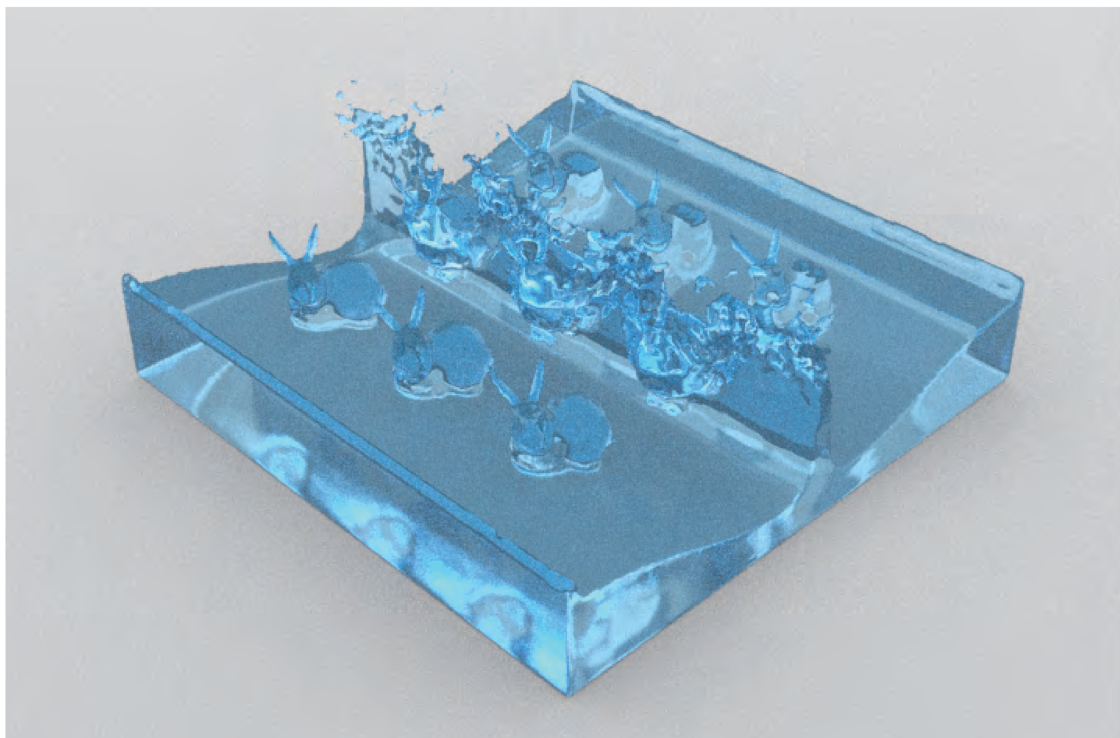

图 5.4: 液体模型下落仿真效果图

5.2.1 海浪仿真

为了实现较为逼真的海浪场景仿真，需要较大的仿真规模，粒子数要求也较多。如果采用复杂的仿真计算方法，拥有大量粒子的大规模场景仿真的效率较低，难以在单机上进行仿真。同时为了较好维持液体的不可压缩性，海浪的仿真采用的是 WCSPH[38] 算法，本小节将对该仿真方法进行概述。

WCSPH 算法与基础的 SPH 仿真方法类似，算法过程大致相同，主要区别在于对纳维-斯托克方程离散求解的计算公式存。首先，对于压力场项产生的加速度项 $-\frac{\nabla p}{\rho}$，WCSPH 采用的计算公式为：

$$-\frac{\nabla p(\mathbf{x}_i)}{\rho_i} = -\sum_j m_j(\frac{p_i}{{\rho_i}^2} + \frac{p_j}{{\rho_j}^2})\nabla W(\mathbf{x}_i - \mathbf{x}_j, h), \tag{5.2}$$

压强的计算公式同公式2.7。对于粘滞力项产生的加速度 $\upsilon\nabla^2\mathbf{u}$，WCSPH 引入人工

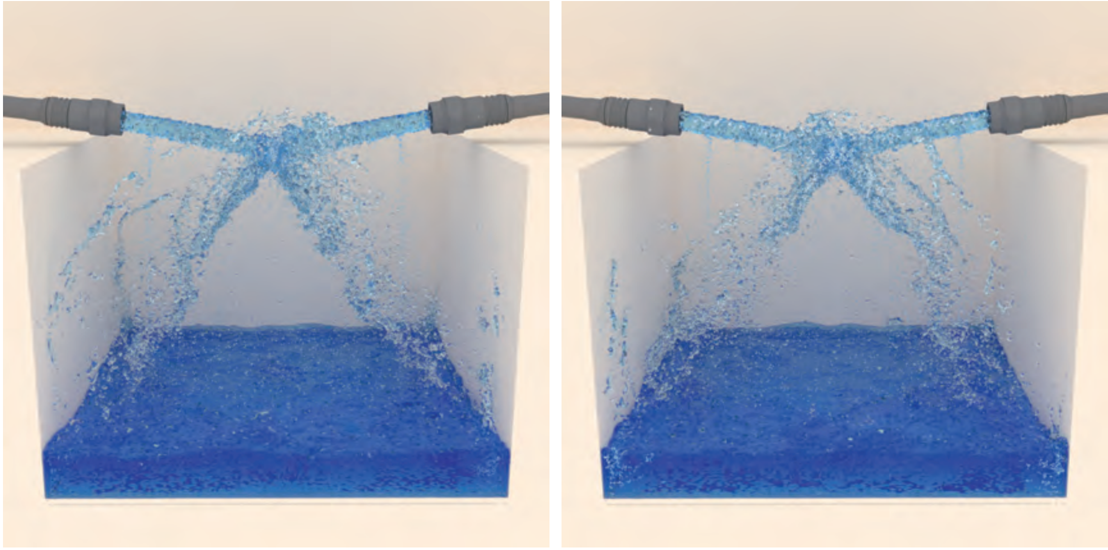

图 5.5: 柱状高速流体对冲仿真效果图

粘滞力的计算公式：

$$\nu\nabla^2\mathbf{u}(\mathbf{x}_i) = \begin{cases} \sum_j m_j \Pi_{ij} \nabla W(\mathbf{x}_i - \mathbf{x}_j, h), & (\mathbf{u}_i - \mathbf{u}_j)(\mathbf{x}_i - \mathbf{x}_j) < 0 \\ 0, & (\mathbf{u}_i - \mathbf{u}_j)(\mathbf{x}_i - \mathbf{x}_j) \geq 0 \end{cases}, \tag{5.3}$$

其中 Π_{ij} 的计算公式为：

$$\Pi_{ij} = \frac{2\varphi h u_s}{\rho_i + \rho_j}\left(\frac{(\mathbf{u}_i - \mathbf{u}_j)(\mathbf{x}_i - \mathbf{x}_j)}{|\mathbf{u}_i - \mathbf{u}_j|^2 + \varepsilon h^2}\right), \tag{5.4}$$

公式中的 φ 是常数，取值范围是 0.08 到 0.5，ε 也是常数，值为 0.01，u_s 是声音在液体中的速度大小。此外，WCSPH 提出了一种新的表面张力产生的加速度 $\mathbf{a}_s$ 的计算方法，计算公式为：

$$\mathbf{a}_s(\mathbf{x}_i) = -\frac{\xi}{m_i}\sum_j m_j(\mathbf{x}_i - \mathbf{x}_j)W(\mathbf{x}_i - \mathbf{x}_j, h), \tag{5.5}$$

公式中的 ξ 为表面张力系数。在 WCSPH 算法中，为了进一步保证算法的稳定性和收敛性，每一帧的时间步长 Δt 需满足公式：

$$\Delta t = \min\left(0.25 \cdot \min_i\left(\frac{h}{|\mathbf{f}_i|}\right), 0.4 \cdot \frac{h}{u_s \cdot (1 + 0.6\varphi)}\right), \tag{5.6}$$

公式中的 $\mathbf{f}_i$ 代表每个粒子受到的外力。

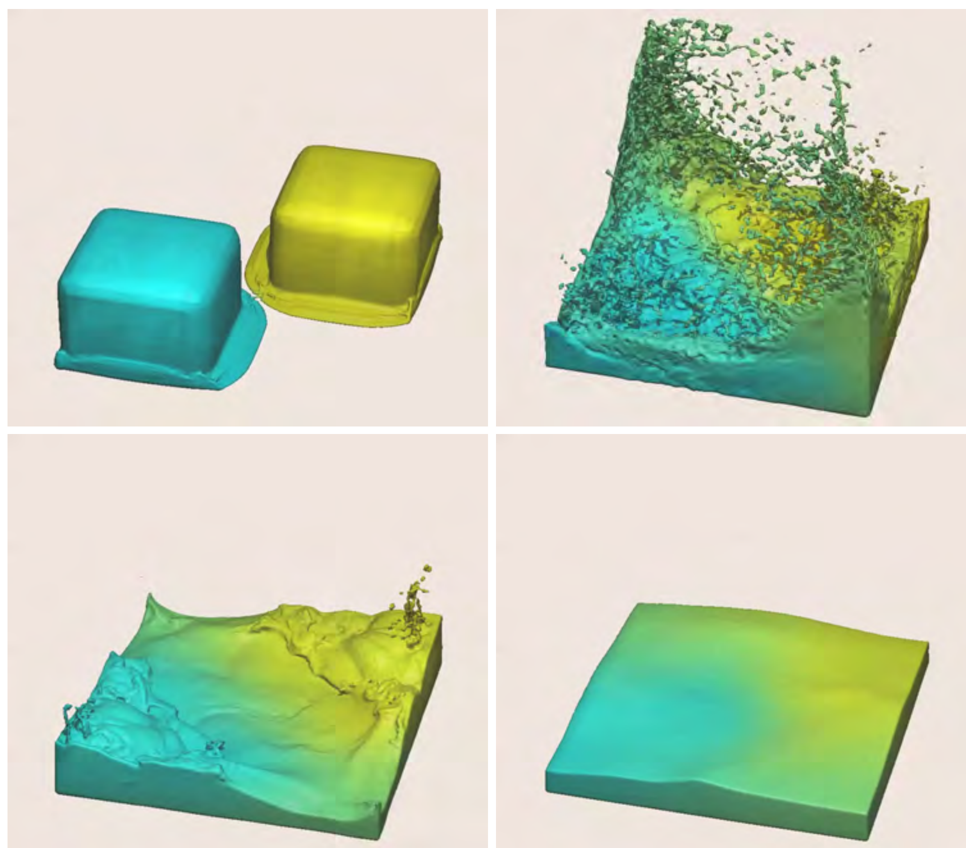

图 5.6: 多流体耦合仿真效果图

根据本小节中的公式，实现了将 WCSPH 算法集成到本文的框架中，为了进一步实现海浪场景的模拟，需要进行场景设计。在进行场景设计的时候，场景的下边界设计为斜坡，使得粒子在重力作用下可以沿斜坡向下移动。在海浪回落的方向，周期性施加适当推力，使得粒子可以涌向斜坡上方，基于该方式可以实现海浪场景的仿真，仿真效果如图5.3所示。在未对时间步长进行限制的情况下，WCSPH 的采用的计算公式已经具有较好的稳定性，在一定程度上可以让仿真场景具有较好的不可压缩性，考虑到海浪场景不需要较为严格的不可压缩性以及对仿真效率的要求较高，本节在实现过程中未按照公式5.6对时间步长进行限制。

5.2.2 液体模型仿真

在进行液体模型仿真的时候，需要保证液体模型形状的稳定性，即不可在仿真过程中模型中的液体发生明显膨胀亦或是收缩。液体模型的仿真对不可压缩性的

要求较高，若严格按照 WCSPH 的计算方法，受到时间步长的限制，完成整个仿真过程的时间相对 PCISPH 较长，而不可压缩性的表现二者相近，故此，液体模型的仿真采用 PCISPH 算法 [43]，本小节将对该方法进行概述。

算法 8: PCISPH 基本算法

输入: 仿真场景的初始化的所有变量值

1 **repeat**
2 　**foreach** 粒子 **do**
3 　　根据初始粒子密度 ρ_0 计算除压力项外的所有加速度
4 　　预测压强和压力项加速度为零
5 　**end**
6 　**while** $\rho_e > E$ ***Or*** 迭代次数小于迭代次数阈值 **do**
7 　　**foreach** 粒子 **do**
8 　　　根据预测压力项加速度预测粒子速度 $\mathbf{u}^*$ 和位置 $\mathbf{x}^*$
9 　　**end**
10 　　**foreach** 粒子 **do**
11 　　　预测粒子密度 ρ^*
12 　　　预测粒子密度误差 $\rho_e = \rho^* - \rho_0$
13 　　　预测粒子压强 $p^* = p^* + \varrho \cdot \rho_e$
14 　　**end**
15 　　**foreach** 粒子 **do**
16 　　　根据 ρ_0 和 p^* 预测压力项加速度
17 　　**end**
18 　**end**
19 　**foreach** 粒子 **do**
20 　　计算更新粒子速度 $\mathbf{u}$ 和位置 $\mathbf{x}$
21 　**end**
22 **until** 仿真结束;

PCISPH 是通过预测矫正的方式来提高仿真流体的不可压缩性，故而其计算方

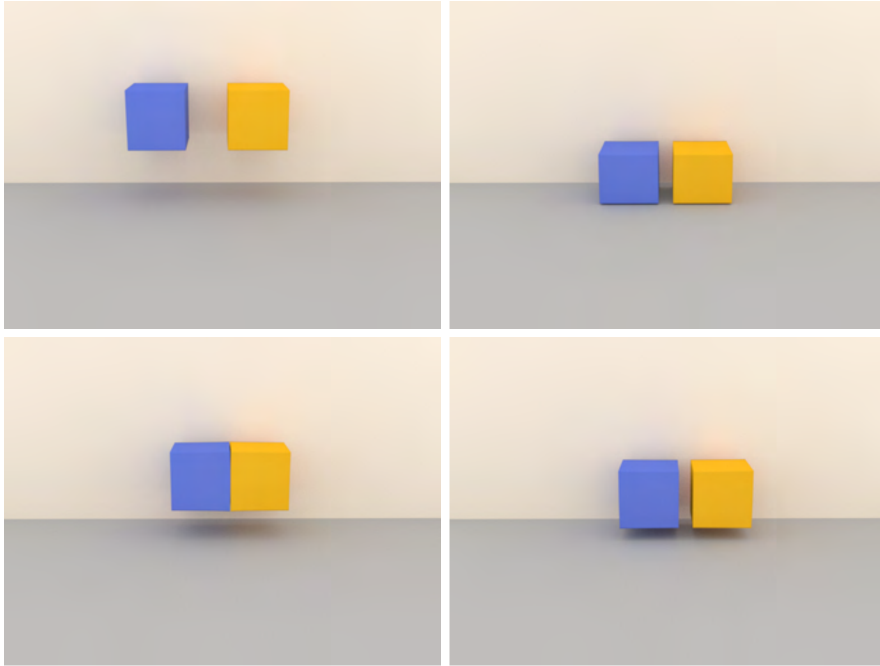

图 5.7: 弹性固体碰撞仿真效果图

式与传统的 SPH 算法以及 WCSPH 算法不同，大致计算流程如算法8所示。

在算法8中，涉及到的预测求解的公式与基础的 SPH 算法大多一致，主要差别在于压强的预测计算方式不同，ϱ 的计算公式为：

$$\varrho = \frac{2m^2\Delta t^2}{\rho_0^2(\sum_j \nabla W_{ij} \cdot \sum_j \nabla W_{ij} + \sum_j (\nabla W_{ij} \cdot W_{ij}))}. \tag{5.7}$$

结合 PCISPH 的算法流程和相关的计算公式，实现了 PCISPH 在本文框架中的集成，最终实现了液体模型的仿真，图5.4展示了液态兔子悬空落入坍塌的水体中，图5.5展示了柱状高速流体的对冲效果。

5.2.3　多流体仿真

为了实现较为复杂的流体仿真效果，如多种液体之间的混合，本小节采用了 Ren 等人提出的多种流体仿真模型 [7]，本文中称 Ren 等人的方法为 MFSPH，本小节将对该技术的核心思想和理论基础做简要概述。

MFSPH 是基于 WCSPH 扩展得到的，该方法也可以集成到 PCISPH 中，不同于传统的 WCSPH 算法，MFSPH 算法中的粒子涉及到多种材质属性。故此，该方

法可以模拟多种不同液体之间的交互效果，而交互的关键在于每一个粒子所代表的各个材质比例之间变化值的计算，即相场的计算。不同的材质粒子之间的比例关系满足如下公式：

$$\sum_k \alpha_k = 1, \quad \alpha_k \geq 0. \tag{5.8}$$

由于一个粒子可能代表多种材质属性，故此一个粒子的质量不一定恒定，粒子密度的计算公式为：

$$\rho = \sum_k \alpha_k \rho_k, \tag{5.9}$$

为了保证流体的质量守恒，需要引入连续性方程来约束粒子各材质占比的变化，求解相场变化的计算模型为：

$$\frac{D\alpha_k}{Dt} = -\alpha_k \nabla \cdot \mathbf{u} - \nabla \cdot (\alpha_k \mathbf{u}_{mk}), \tag{5.10}$$

其中 $\mathbf{u}_{mk}$ 是该粒子中材质 k 的漂移速度。漂移速度的计算公式为：

$$\mathbf{u}_{mk} = \tau(\rho_k - \sum_{k'} c_{k'} \rho_{k'})\mathbf{a}' - \tau(\nabla p_k - \sum_{k'} c_{k'} \nabla p_{k'}) - \sigma(\frac{\nabla \alpha_k}{\alpha_k} - \sum_{k'} c_{k'} \frac{\nabla \alpha_k}{\alpha_k}), \tag{5.11}$$

τ 和 σ 通常为常量系数，$c_k = (\alpha_k \rho_k)/\rho$，$\mathbf{a}' = \mathbf{g} - D\mathbf{u}/Dt$。通过上述公式可以实现对 MFSPH 相场变化的计算。在 MFSPH 算法中，考虑到不同材质流体间力的作用与单一材质流体存在差异，MFSPH 对纳韦-斯托克方程进行了变形，得到混合流体速度场的计算模型：

$$\frac{D(\rho \mathbf{u})}{Dt} = \nabla \cdot \mathbf{T}_m + \nabla \cdot \mathbf{T}_{dm} - \nabla p + \rho \mathbf{a}, \tag{5.12}$$

其中 $\mathbf{T}_m$ 是粘性应力张量，$\mathbf{T}_{dm}$ 是弥散张量，Ren 等人给出了 MFSPH 算法中弥散张量的具体推导过程，本文不再赘述。通过求解相场和速度场的变化，即可实现多种流体之间的耦合效果。该方法大大简化了传统多流体耦合的计算步骤，易于实现，且仿真效果较好。具体的仿真效果如图5.6所示，展示了两种不同液体之间的碰撞混合过程，两种液体分别用蓝绿色和黄色进行标注，在发生混合的交界处附近，粒子根据混合的程度表现出介于两种不同颜色之间的不同程度的渐变颜色，通过在框架中集成 GVDB[44]，实现该场景的实时仿真渲染。

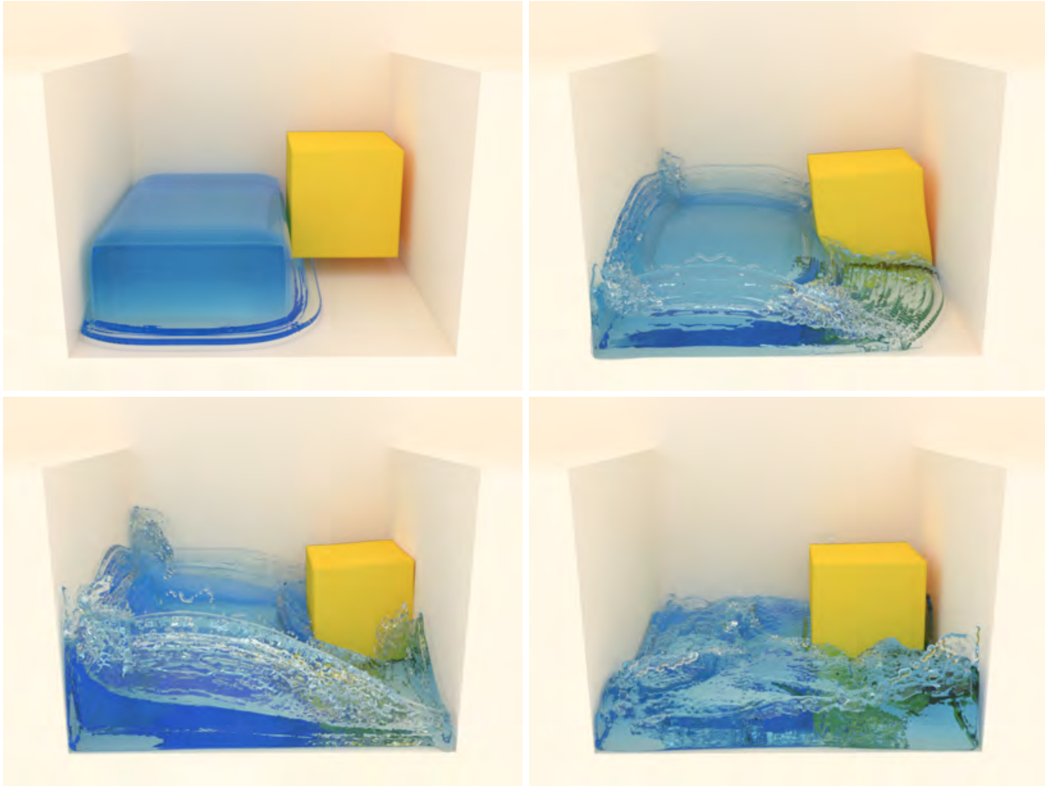

图 5.8: 弹性固体与液体交互效果图

5.2.4　弹性固体液体交互仿真

为了进一步实现固体的模拟仿真，如弹性固体以及弹性固体与液体之间的交互仿真，本小节采用的是 Yan 等人提出的多相流模型 [8]，该模型是在 Ren 等人的 MFSPH 的基础上扩展得到，本文中称该方法为 MPSPH。本小节将对该技术做简要的概述。

在 MPSPH 算法中，为了实现弹性固体的仿真，其主要方法是改变 MFSPH 中速度场的求解模型。在原 MFSPH 的速度场求解模型上，MPSPH 增加了偏应力张量 $\mathbf{T}_{sm}$，从而得到速度场的求解模型：

$$\frac{D(\rho\mathbf{u})}{Dt} = \nabla\cdot\mathbf{T}_{sm} + \nabla\cdot\mathbf{T}_{m} + \nabla\cdot\mathbf{T}_{dm} - \nabla p + \rho\mathbf{a}. \tag{5.13}$$

故此，相比于 MFSPH，MPSPH 的主要扩展技术在于求解偏应力张量 $\mathbf{T}_{sm}$，而偏应力张量的求解涉及到物体材质的特性，故此对于不同的仿真材质，偏应力张量的求解方式可能不同，Yan 等人较为详细地介绍了弹塑性固体偏应力张量的计算模型，

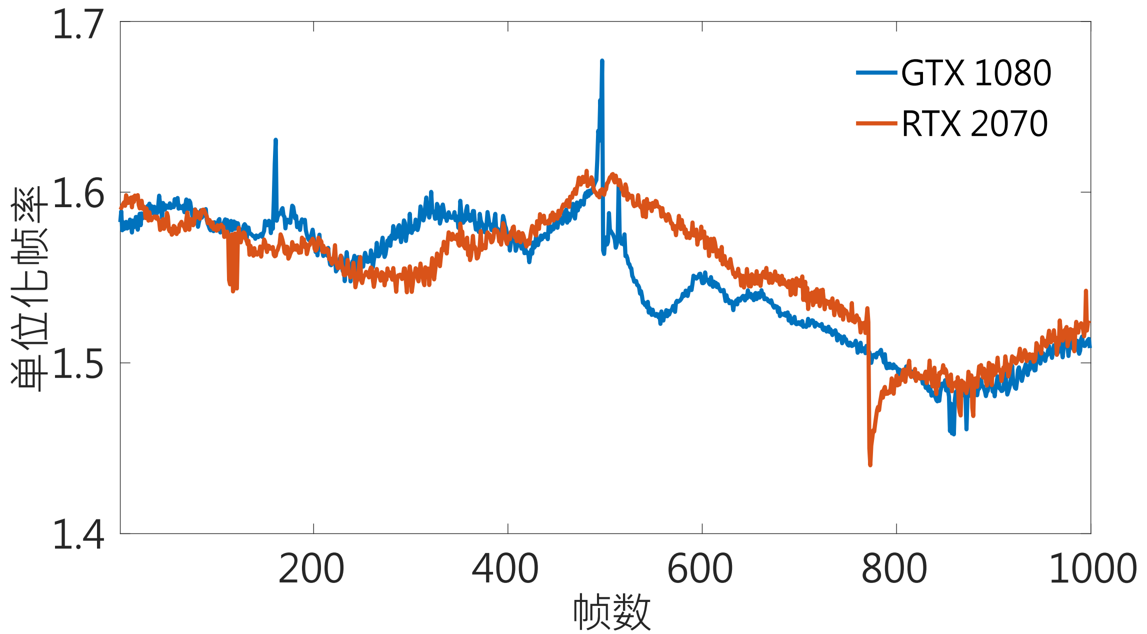

图 5.9: 海浪仿真性能测试结果图

本文不再赘述。为了较好实现偏应力张量模型的离散求解，MPSPH 采用的离散求解模型为：

$$(\nabla\cdot A)_i=\sum_j\frac{m_j}{\rho_j}\sum_k\frac{2(\alpha_k)_i(\alpha_k)_j}{(\alpha_k)_i+(\alpha_k)_j}(A_j\pm A_i)\nabla W_{ij},\tag{5.14}$$

通过该离散求解模型和偏应力张量的求解模型，MPSPH 可以根据相应的本构模型求解计算出不同材质固体的偏应力张量，由此完成多种不同材质固体的仿真。

因为 MPSPH 是基于 MFSPH 扩展而来，一个粒子可以同时表征多种不同属性的材质对象，如在一个粒子上可以存在固体的材质，与此同时也能存在液体的材质对象，由此可以实现固体到液体的渐变或者是液体到固体的渐变，该过程可以通过对相场的求解实现。此外 MPSPH 也可实现变形固体与液体之间的交互，即部分粒子的液体参数值为 1，部分粒子的弹性固体参数值为 1，且保持该参数比例不变，即可实现不同材质之间的交互仿真。弹性固体间的交互仿真效果如图5.7所示，展示了两个弹性固体之间的碰撞，固液交互的仿真效果如图5.8所示，展示了弹性固体和液体之间的变形交互。

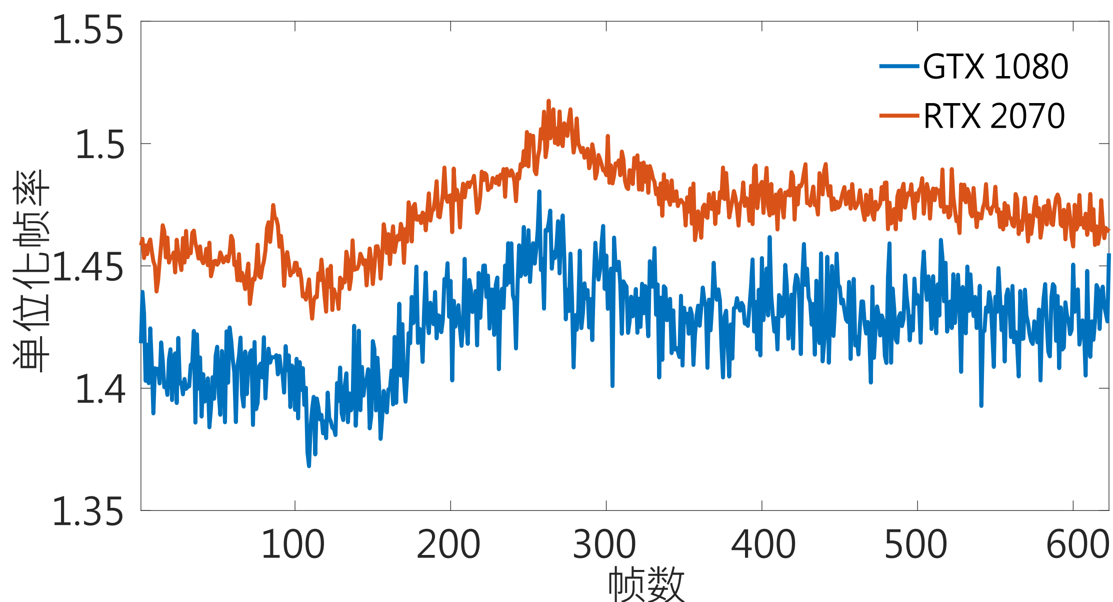

图 5.10: 液体模型下落仿真测试结果图

5.3　仿真测试分析

为了证实本文的框架对集成的多种 SPH 算法均具有有效的加速作用，本节对 WCSPH，PCISPH，MFSPH 以及 MPSPH 在本文框架上的运行效率进行了相应的测试分析。

针对 WCSPH 算法，基于海浪的仿真场景进行测试，测试场景的粒子数为 17,970,160，场景中的一级网格数为 6000000，初始粒子密约为 1000。海浪仿真场景在不同 GPU 上的测试结果如图5.9所示，图中的曲线是不同平台上仿真过程中每一帧的帧率相对 RSMS 方法帧率的加速比。测试结果表明该实验中本文的框架在不同 GPU 平台上可以实现平均单帧 1.55 倍的性能提升，加速效果明显。该实验中，帧率变化趋势的幅度较为明显，这是因为仿真过程中持续不断的粒子飞溅和挤压导致粒子密集度持续发生增大和减小。

针对 PCISPH 算法，基于图5.4中的仿真场景进行测试，测试场景的粒子数为 1,516,563，场景中的一级网格数为 39304，初始粒子密集度约为 500。从算法8中可以看出，PCISPH 需要通过迭代矫正来保证算法的不可压缩性，这个过程相比基础的 SPH 算法以及 WCSPH 算法涉及到较多的求解步骤，这意味着需要多次重

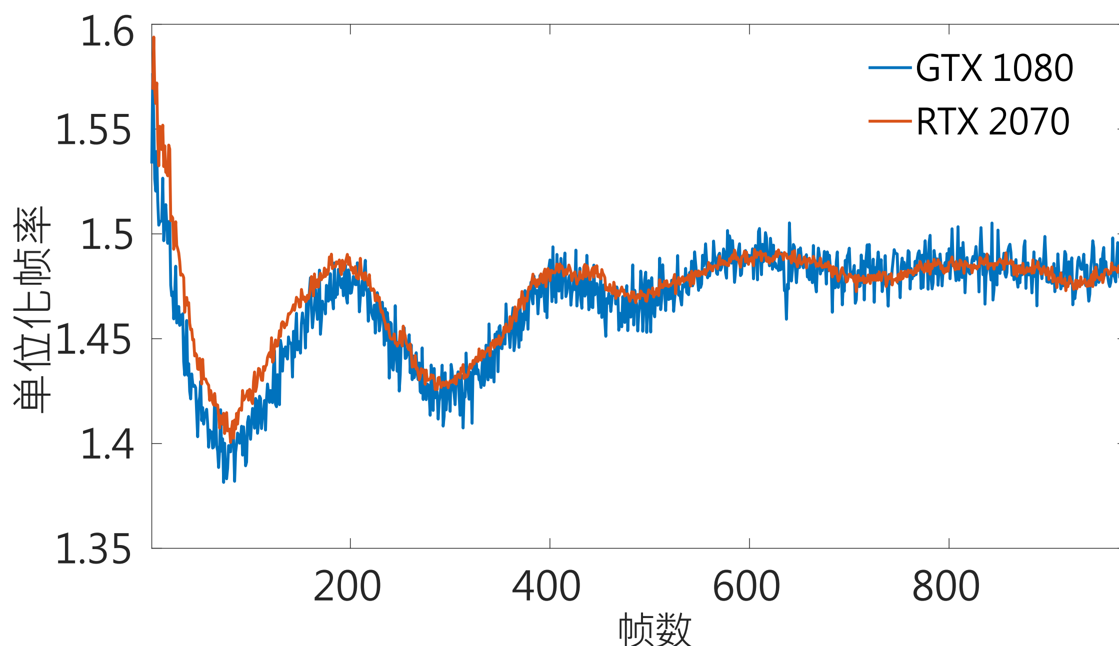

图 5.11: 多流体耦合仿真测试结果图

启 CUDA 核函数。该实验在不同 GPU 上的测试结果如图5.10所示，在该实验中，PCISPH 算法在 RTX 2070 上的相对帧率要比 GTX 1080 的相对帧率高，本文的框架在两种 GPU 上分别可以达到 1.47 和 1.43 倍的性能提升。算法相对帧率的整体变化趋势相对较小，这是因为 PCISPH 可以较好地保持算法的不可压缩性，从而使得粒子的分布相对较为均匀，粒子密集度的变化相对较小。

针对 MFSPH 算法，对多种液体之间的交互场景进行测试，测试场景的一级网格总数为 156464，粒子总数为 3,169,044，初始粒子密集度约为 500。相比 WCSPH 和 PCISPH，MFSPH 的求解过程中涉及到多种材质属性的求解计算，并且粒子上的物理量较多，每一步的求解更为复杂。针对该实验的测试结果如图5.11所示，在液体下落和飞溅的过程中，算法帧率的变化趋势成波浪式变化，这也是粒子在运动过程中的相互挤压和分散造成的，随着液体趋于平稳，相对帧率的变化趋势也逐渐平稳。在该实验中，本文的框架可以达到平均单帧 1.47 倍的性能提升。

针对 MPSPH，基于弹性固体的交互碰撞场景进行测试，测试场景中一级网格个数为 77452，例子总数为 2,788,192，粒子密集度约为 500。相比 MFSPH，MPSPH 增加了偏应力张量相关的求解计算，计算复杂度在本文框架所集成的 SPH 算法中

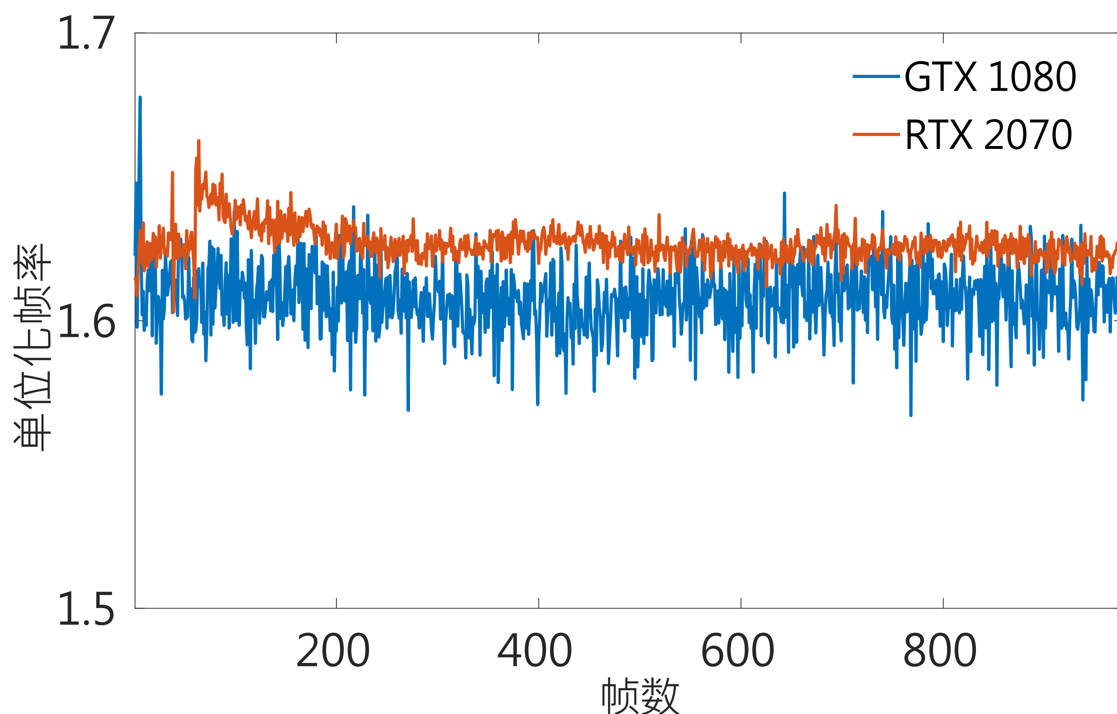

图 5.12: 固体碰撞仿真测试结果图

是最高的。实验的测试结果如图5.12所示，本文的框架在 GTX 1080 和 RTX 2070 上平均帧率的加速比分别为 1.61 和 1.63，性能提升依然明显，可见本文框架的高效性。该实验中粒子间相对运动的变化幅度较小，粒子密集度较为稳定，故而相对帧率的变化趋势较为平稳。

5.4 本章小结

本章对本文提出的算法进行了整合，实现了一个统一的加速计算框架，并用多个实验用例测试整体加速框架的加速比与粒子密集度和粒子总数之间的关系。测试表明，整体加速算法的加速比与粒子密集度存在正相关关系，受粒子总数的影响较小。通过在整体框架中集成多种 SPH 仿真方法，实现了多种物理现象的仿真，并针对框架中的每种仿真算法，在不同的 GPU 平台上进行了性能测试分析。实验表明，本文的系统框架的加速效果明显，在不同硬件平台下所有测试场景中均能达到较好的计算效率。

第六章 总结和展望

本章将对全文的工作进行一个总结，主要包括对本文提出的方法以及测试结果的总结，同时也指出了本文方法中存在的问题。在完成全文的总结后，结合本文的工作内容，对未来进一步的探究进行了展望。

6.1 本文工作总结

本文首先基于针对 SPH 算法的 GPU 单一任务调度算法设计了一种双重尺度的任务调度算法，该调度算法不仅可以在一定程度上减少不同任务之间冗余数据的重复加载，从而提高算法的计算效率，而且不会增加额外的闲置线程。考虑到空间中粒子分布的随机性，以及计算任务中粒子数量的合理性，设计了一种新的编码方法，通过该编码方法实现了一种混合算法调度策略，根据不同的粒子分布情况，分别发挥传统主流算法的优势以及任务调度算法的优势，使得算法更具普适性和高效性。

考虑到 SPH 的邻居查找方法涉及到较多的无效邻居粒子的加载计算，本文设计了一种四层结构的多层级垂直网格划分方法，并针对该网格划分方法设计了相应的编码方法，通过这种编码方法，可以提高粒子数据分布的一致性，这在一定程度上也能提高线程的并行效率。通过新的网格划分和编码方法，实现了一种在不同网格层级上进行邻居查找的方法。该方法通过利用粒子分布的连续性，减少了不同网格间切换所需的循环迭代的次数，从而实现了一种高效的邻居空间范围缩减方法，有效减少了无效粒子的加载计算。考虑到不同网格层级上的空间的裁剪对粒子连续性的影响，进一步设计了一种动态的空间缩减方法，尽可能保证空间缩减方法的高效性和有效性。

最后，根据本文提出的多层级任务调度方法和多层级空间网格方法，设计了在多层级网格空间上的多层级任务调度方法。针对该统一的加速算法，本文进行了细

致的性能分析，给出了统一加速方法的加速效率以及相关的影响因素，并给出了具体的运算时间。通过结合多种不同的 SPH 算法，进一步实现了完整的加速仿真框架，在该加速仿真框架上，进行了多种场景的物理仿真以及相关的性能测试分析，从而充分说明了本文中的计算框架的普适性和高效性。

6.2 研究展望

本文的计算框架对 SPH 的数值计算过程没有任何简化和修改，不影响具体的数值结果，理论上，所有的 SPH 算法均可集成到本文的计算框架中，从而可以进一步提高本文框架的仿真能力。在本文的仿真加速框架中，设计了多个阈值参数，而这些参数的值均是根据实验的测试结果进行人为设定，考虑到真实仿真中粒子分布的随机性，该参数设定方式存在一定的不合理性。为了进一步提高仿真框架对不同粒子分布情况的仿真场景的适应性，需要设计一种动态的参数设定方式。

在任务调度策略中，线程组的线程之间可以进行天然的并行协作，基于线程组之间的协作能力，可以设计一种高效的粒子合并策略，从而减少空间中的仿真粒子数量，进而实现算法的性能提升。显然这种方法对计算的数值结果存在影响，在设计合并策略的时候，需要考虑数值的稳定性。该数值简化的计算方式可以作为仿真框架的可选仿真方式，在进行一些对数值要求较低的仿真计算时，可以采用这种计算方式。

本文的加速框架较为充分地利用了 GPU 中的存储层级结构，有效地减少了全局显存的访问，但仍然需要进行较多的全局显存读写操作。为了进一步提高加速框架的加速效率，可以通过更为细致地组织数据的存储结构，使得在全局显存上数据的读写更为高效。此外，对于超大规模的仿真场景，单 GPU 的存储空间无法满足仿真的存储空间需求，需要将本文的框架扩展到多 GPU 的硬件平台上。

本文的方法同样适用于与 SPH 类似的算法，可以将本文中的方法进行迁移，扩展到其他类似的算法中，进而设计一种统一的计算框架，使其能够满足多种计算需求，从而提高计算框架的实用性。

参考文献

[1] GINGOLD R A, MONAGHAN J J. Smoothed particle hydrodynamics: theory and application to non-spherical stars[J]. Monthly Notices of the Royal Astronomical Society, 1977, 181(3) : 375 – 389.

[2] MONAGHAN J J. Smoothed particle hydrodynamics[J]. Annual Review of Astronomy and Astrophysics, 1992, 30(1) : 543 – 574.

[3] MÜLLER M, CHARYPAR D, GROSS M H. Particle-based fluid simulation for interactive applications[C] // Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, San Diego, CA, USA, July 26-27, 2003. 2003 : 154 – 159.

[4] IHMSEN M, ORTHMANN J, SOLENTHALER B, et al. SPH Fluids in Computer Graphics[C] // Eurographics 2014 - State of the Art Reports, Strasbourg, France, April 7-11, 2014. 2014 : 21 – 42.

[5] MÜLLER M, KEISER R, NEALEN A, et al. Point based animation of elastic, plastic and melting objects[C] // Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, Grenoble, France, August 27-29, 2004. 2004 : 141 – 151.

[6] GERSZEWSKI D, BHATTACHARYA H, BARGTEIL A W. A point-based method for animating elastoplastic solids[C] // Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, SCA 2009, New Orleans, Louisiana, USA, August 1-2, 2009. 2009 : 133 – 138.

[7] REN B, LI C, YAN X, et al. Multiple-Fluid SPH Simulation Using a Mixture Model[J]. ACM Transactions on Graphics, 2014, 33(5): 171:1 – 171:11.

[8] YAN X, JIANG Y, LI C, et al. Multiphase SPH simulation for interactive fluids and solids[J]. ACM Transactions on Graphics, 2016, 35(4): 79:1 – 79:11.

[9] YANG T, CHANG J, LIN M C, et al. A unified particle system framework for multi-phase, multi-material visual simulations[J]. ACM Transactions on Graphics, 2017, 36(6): 224:1 – 224:13.

[10] BENDER J, KOSCHIER D. Divergence-Free SPH for Incompressible and Viscous Fluids[J]. IEEE Trans. Vis. Comput. Graph., 2017, 23(3): 1193 – 1206.

[11] KOSCHIER D, BENDER J, SOLENTHALER B, et al. Smoothed Particle Hydrodynamics Techniques for the Physics Based Simulation of Fluids and Solids[C] // JAKOB W, PUPPO E. Eurographics 2019 - Tutorials. [S.l.]: The Eurographics Association, 2019.

[12] KIPFER P, SEGAL M, WESTERMANN R. UberFlow: a GPU-based particle engine[C] // Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware 2004, Grenoble, France, August 29-30, 2004. 2004: 115 – 122.

[13] KOLB A, LATTA L, REZK-SALAMA C. Hardware-based simulation and collision detection for large particle systems[C] // Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware 2004, Grenoble, France, August 29-30, 2004. 2004: 123 – 131.

[14] KOLB A, CUNTZ N. Dynamic particle coupling for GPUbased fluid simulation[C] // Proceedings of 18th Symposium on Simulation Technique. 2005: 722 – 727.

[15] HARADA T, KOSHIZUKA S, KAWAGUCHI Y. Smoothed Particle Hydrodynamics on GPUs[C] // Proceedings of computer graphic. 2007: 63 – 70.

[16] GREEN S. Cuda particles[J]. NVIDIA Whitepaper, 2008, 2(3.2) : 1.

[17] HÉRAULT A, BILOTTA G, DALRYMPLE R A. Sph on gpu with cuda[J]. Journal of Hydraulic Research, 2010, 48(S1) : 74 – 79.

[18] GOSWAMI P, SCHLEGEL P, SOLENTHALER B, et al. Interactive SPH Simulation and Rendering on the GPU[C] // Proceedings of the 2010 Eurographics/ACM SIGGRAPH Symposium on Computer Animation, SCA 2010, Madrid, Spain, 2010. 2010 : 55 – 64.

[19] CRESPO A J C, DOMÍNGUEZ J M, ROGERS B D, et al. DualSPHysics: Open-source parallel CFD solver based on Smoothed Particle Hydrodynamics (SPH)[J]. Computer Physics Communications, 2015, 187 : 204 – 216.

[20] DOMÍNGUEZ J M, CRESPO A J C, GÓMEZ-GESTEIRA M. Optimization strategies for CPU and GPU implementations of a smoothed particle hydrodynamics method[J]. Computer Physics Communications, 2013, 184(3) : 617 – 627.

[21] DOMíNGUEZ J M, CRESPO A J C, GóMEZ-GESTEIRA M, et al. Neighbour lists in smoothed particle hydrodynamics[J]. International Journal for Numerical Methods in Fluids, 2011, 67(12) : 2026 – 2042.

[22] WINKLER D, REZAVAND M, RAUCH W. Neighbour lists for smoothed particle hydrodynamics on GPUs[J]. Computer Physics Communications, 2018, 225 : 140 – 148.

[23] XIA X, LIANG Q. A GPU-accelerated smoothed particle hydrodynamics (SPH) model for the shallow water equations[J]. Environmental Modelling and Software, 2016, 75 : 28 – 43.

[24] WINKLER D, MEISTER M, REZAVAND M, et al. gpuSPHASE - A shared memory caching implementation for 2D SPH using CUDA[J]. Computer Physics Communications, 2017, 213 : 165 – 180.

[25] WINKLER D, REZAVAND M, MEISTER M, et al. gpuSPHASE - A shared memory caching implementation for 2D SPH using CUDA (new version announcement)[J]. Computer Physics Communications, 2019, 235 : 514 – 516.

[26] OHNO K, NITTA T, NAKAI H. SPH-based Fluid Simulation on GPU Using Verlet List and Subdivided Cell-Linked List[C] // Fifth International Symposium on Computing and Networking, CANDAR 2017, Aomori, Japan, November 19-22, 2017. 2017 : 132 – 138.

[27] 阮骥鸣. 面向 SPH 流体模拟的异构并行计算方法 [D]. [S.l.] : 华东师范大学, 2017.

[28] HUANG K, RUAN J, ZHAO Z, et al. A General Novel Parallel Framework for SPH-centric Algorithms[J]. Proc. ACM Comput. Graph. Interact. Tech., 2019, 2(1) : 7:1 – 7:16.

[29] ZHANG F, HU L, WU J, et al. A SPH-based method for interactive fluids simulation on the multi-GPU[C] // Proceedings of the 10th International Conference on Virtual Reality Continuum and its Applications in Industry, VRCAI 2011, Hong Kong, China, December 11-12, 2011. 2011 : 423 – 426.

[30] HU L, SHEN X, LONG X. Research on SPH Parallel Acceleration Strategies for Multi-GPU Platform[C] // Advanced Parallel Processing Technologies - 10th International Symposium, APPT 2013, Stockholm, Sweden, August 27-28, 2013, Revised Selected Papers. 2013 : 104 – 118.

[31] RUSTICO E, BILOTTA G, GALLO G, et al. Smoothed Particle Hydrodynamics Simulations on Multi-GPU Systems[C] // Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2012, Munich, Germany, February 15-17, 2012. 2012 : 384 – 391.

[32] RUSTICO E, BILOTTA G, HÉRAULT A, et al. Advances in Multi-GPU Smoothed Particle Hydrodynamics Simulations[J]. IEEE Transactions on Parallel and Distributed Systems, 2014, 25(1) : 43 – 52.

[33] VALDEZ-BALDERAS D, DOMÍNGUEZ J M, ROGERS B D, et al. Towards accelerating smoothed particle hydrodynamics simulations for free-surface flows on multi-GPU clusters[J]. Journal of Parallel and Distributed Computing, 2013, 73(11) : 1483 – 1493.

[34] DOMÍNGUEZ J M, CRESPO A J C, VALDEZ-BALDERAS D, et al. New multi-GPU implementation for smoothed particle hydrodynamics on heterogeneous clusters[J]. Computer Physics Communications, 2013, 184(8) : 1848 – 1860.

[35] VERMA K, SZEWC K, WILLE R. Advanced load balancing for SPH simulations on multi-GPU architectures[C] // 2017 IEEE High Performance Extreme Computing Conference, HPEC 2017, Waltham, MA, USA, September 12-14, 2017. 2017 : 1 – 7.

[36] 谭捷, 杨旭波. 基于物理的流体动画综述 [J]. 中国科学信息科学：中国科学, 2009, 39(5) : 499 – 514.

[37] J.J., MONAGHAN. Simulating Free Surface Flows with SPH[J]. Journal of Computational Physics, 1994.

[38] BECKER M, TESCHNER M. Weakly compressible SPH for free surface flows[C] // GLEICHER M, THALMANN D. Proceedings of the 2007 ACM SIG-

GRAPH/Eurographics Symposium on Computer Animation, SCA 2007. [S.l.] : Eurographics Association, 2007 : 209 – 217.

[39] SETALURI R, AANJANEYA M, BAUER S, et al. SPGrid: a sparse paged grid structure applied to adaptive smoke simulation[J]. ACM Trans. Graph., 2014, 33(6) : 205:1 – 205:12.

[40] GAO M, WANG X, WU K, et al. GPU optimization of material point methods[J]. ACM Trans. Graph., 2018, 37(6) : 254:1 – 254:12.

[41] MORTON G. A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing[J], 1966.

[42] HOLZMÜLLER D. Efficient Neighbor-Finding on Space-Filling Curves[J]. CoRR, 2017, abs/1710.06384.

[43] SOLENTHALER B, PAJAROLA R. Predictive-corrective incompressible SPH[J]. ACM Trans. Graph., 2009, 28(3) : 40.

[44] HOETZLEIN R K. GVDB: Raytracing Sparse Voxel Database Structures on the GPU[C] // ASSARSSON U, HUNT W. Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics. [S.l.] : The Eurographics Association, 2016.

致 谢

三年的研究生生涯转眼即逝，回想这三年的时光，难以忘记参加科研竞赛时的通宵达旦，难以忘记赶投论文的各种艰辛，难以忘记这三年中大家对我的帮助，正是这些种种让我这从一个稚嫩的本科生，成长为一个合格的硕士生。

首先我要感谢我的导师王长波教授。王老师日常工作认真负责，科研经验丰富，科研态度严谨，以身作则，对学生的指导从不放松，使得我们养成了良好的科研习惯，学会了科学的研究方法，让我们在自己的科研道路上打下坚实的基础，更重要的是让我们学会做一个认真负责的人。在生活中王老师非常关心我们的身心健康，为我们营造了一个团结友爱，乐观积极，互帮互助的学习氛围，这为我们的成长提供了很好的环境。正是在这样的环境以及王老师的帮助下，在研究生期间，取得了相对不错的成绩。这里，我要对我的导师王长波教授表示衷心的感谢！

在科研探究的道路上，我还要感谢秦洪教授，秦老师对科研的热情十分高昂，拥有在国内外顶级会议期刊上发表论文的丰富经验。在研究生撰写学术论文期间，秦老师给了我许多的英文写作上的宝贵经验和技巧，在日常的科研探究上也给了我很多鼓励和建议，对我在科研道路上的成长，起到了很大的帮助。这里，我也要衷心感谢秦洪教授！

我还要感谢我的实验室同学，感谢赵子鹏同学在我撰写论文期间给予了关键的协助，阮骥鸣同学为我的科研探究提供技术基础，李晨同学对我撰写的科研论文提供了宝贵的建议，还有实验室的其他同学陪我度过了三年难忘的研究生时光。

最后，对我的父母表示衷心的感谢！感谢他们一直以来对我的支持和帮助！

黄可蒙

二零二零年四月于理科大楼

攻读硕士学位期间发表论文和科研情况

■ 发表论文

[1] **Kemeng Huang**, Jiming Ruan, Zipeng Zhao, et al. A General Novel Parallel Framework for SPH-centric Algorithms[J]. **PACMCGIT**, 2019. (**I3D 2019, CCF B**)

[2] **Kemeng Huang**, Zipeng Zhao, Chen Li, et al. Novel Hierarchical Strategies for SPH-centric Algorithms on GPGPU[J]. Submitted to **Graphical Models**. (**CCF B**)

■ 其他科研成果

1. **黄可蒙**, 吴柏威, 董天文. 多无人机对组网雷达的协同干扰. **第十五届中国研究生数学建模竞赛国家一等奖**, 2018.