计算机科学
COMPUTER SCIENCE

基于SYCL的多相流LBM模拟跨平台异构并行计算研究

丁越, 徐传福, 邱昊中, 戴未希, 汪青松, 林拥真, 王正华

引用本文

丁越, 徐传福, 邱昊中, 戴未希, 汪青松, 林拥真, 王正华. 基于SYCL的多相流LBM模拟跨平台异构并行计算研究[J]. 计算机科学, 2023, 50(11): 32-40.

DING Yue, XU Chuanfu, QIU Haozhong, DAI Weixi, WANG Qingsong, LIN Yongzhen, WANG Zhenghua. Study on Cross-platform Heterogeneous Parallel Computing for Lattice Boltzmann Multi-phase Flow Simulations Based on SYCL [J]. Computer Science, 2023, 50(11): 32-40.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.230300123

基于 SYCL 的多相流 LBM 模拟跨平台异构并行计算研究

丁 越 徐传福 邱昊中 戴未希 汪青松 林拥真 王正华
国防科技大学计算机学院 长沙 410073
(dingyue@nudt.edu.cn)

摘 要 异构并行体系结构是当前高性能计算的重要技术趋势。由于各种异构平台通常支持不同的编程模型,跨平台性能可移植异构并行应用开发非常困难。SYCL 是一个基于 C++语言的单源跨平台并行编程开放标准。目前针对 SYCL 的研究主要集中于与其他并行编程模型的性能比较,对 SYCL 中提供的不同并行内核实现及其性能优化研究得较少。针对这一现状,基于 SYCL 编程模型对开源多相流数值模拟软件 openLBMmflow 实现跨平台异构并行模拟,通过对比基础并行版本、细粒度调优的 ND-range 并行版本以及计算到工作项多对一映射方法,系统总结了 SYCL 并行应用的性能优化方法。测试结果表明,在 Intel Xeon Platinum 9242 CPU 以及 NVIDIA Tesla V100 GPU 上,相比优化后的 OpenMP 并行实现,在不需要额外调优的情况下,基础并行版本在 CPU 上获得了 2.91 的加速比,表明了 SYCL 的开箱即用性能具备一定优势。以基础并行版本为基准,ND-range 并行版本通过改变工作组大小及形状,在 CPU 与 GPU 上分别取得了最高 1.45 以及 2.23 的加速比。通过优化计算到工作项的多对一映射改变每个工作项处理的格子数量以及形状,与基础并行版本相比,在 CPU 与 GPU 上分别取得了最高 1.57 以及 1.34 的加速比。结果表明,SYCL 并行应用在 CPU 上更适合采用计算到工作项多对一映射的优化方法,在 GPU 上更适合采用 ND-range 并行内核,以提高性能。

关键词:SYCL;格子玻尔兹曼方法;多相流模拟;异构并行计算;跨平台并行编程模型
中图法分类号 TP391

Study on Cross-platform Heterogeneous Parallel Computing for Lattice Boltzmann Multi-phase Flow Simulations Based on SYCL

DING Yue, XU Chuanfu, QIU Haozhong, DAI Weixi, WANG Qingsong, LIN Yongzhen and WANG Zhenghua
College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China

Abstract Heterogeneous parallel architecture is an important technology trend in current high-performance computing. Since various heterogeneous platforms usually support different programming models, the development of cross-platform performance portable heterogeneous parallel application is difficult. SYCL is a single-source cross-platform parallel programming open standard based on C++ language. The current research on SYCL mainly focuses on the performance comparison with other parallel programming models, but there are few researches on the different parallel kernel implementations provided in SYCL and their performance optimization. To address this situation, the open source multi-phase flow simulation software openLBMflow is implemented based on the SYCL programming model for cross-platform heterogeneous parallel simulation. The performance optimization methods of SYCL parallel applications are systematically summarized by comparing the basic parallel version, the fine-grained tuned ND-range parallel version and many-to-one mapping computation to work-items method. The results show that on Intel Xeon Platinum 9242 CPU and NVIDIA Tesla V100 GPU, the basic parallel kernel achieves a speedup of 2.91 on CPU without additional tuning compared to the optimized OpenMP parallel implementation, indicating the out-of-the-box performance advantage of SYCL. Using the basic parallel version as a baseline, the ND-range parallel version achieves up to 1.45x speedup on the CPU and 2.23x speedup on the GPU respectively by changing the work-group size and shape. By changing and optimizing the number and shape of lattices processed per work-item, the many-to-one mapping computation to work-items method achieves up to 1.57x speedup on the CPU and 1.34x speedup on the GPU respectively compared to the basic parallel version. The results show that SYCL parallel applications are more suitable for many-to-one mapping computation to work-items method on the CPU and ND-range parallel kernels on the GPU to improve performance.

Keywords SYCL, Lattice Boltzmann method, Multi-phase flows imulation, Heterogeneous parallel computing, Cross-platform parallel programming model

到稿日期:2023-03-14 返修日期:2023-06-12
通信作者:徐传福(xuchuanfu@nudt.edu.cn)

1　引言

格子玻尔兹曼方法（Lattice Boltzmann Method，LBM）基于分子运动理论，由介观运动方程发展而来，从微观角度将流场分割为一块块的规则格子（质点），通过格子的流动和碰撞来模拟流体运动[1-2]。相比其他传统计算流体力学（Computational Fluid Dynamics，CFD）模拟方法，LBM 方法算法实现简单、易于并行，并且能够处理复杂的边界以及不规则的结构，逐渐被广泛用于各种理论研究以及工程实践中[3-5]。

随着 CFD 研究和应用的不断发展，CFD 所模拟问题的复杂度、网格规模及求解精度都在不断提升，对大规模并行 CFD 模拟提出了迫切需求[6]。随着摩尔定律的逐步失效，异构体系结构已成为当前高性能计算机的主流趋势[7]。近年来，GPU，MIC，DSP 和 FPGA 等加速器迅速发展，异构高性能计算机广泛采用“CPU＋加速器”架构。为了能够充分利用异构高性能计算机性能，必须采用异构并行编程模型开发异构并行应用，如目前流行的 CUDA[8] 异构并行编程模型，但 CUDA 只能在 NVIDA GPU 上使用。异构体系结构的多样性导致不同的加速器支持不同的编程模型，应用程序需要为不同的异构平台开展异构并行移植和优化，大幅提升了异构并行应用开发和优化的难度。近年来，学术界和工业界提出了一系列跨平台并行编程模型，其支持开发人员通过一套代码为不同体系结构生成可执行文件。例如，通用异构编程语言 OpneCL[9] 将硬件抽象为一个统一的平台模型[10]；OpenACC[11] 则是在 C 或 Fortran 代码中加入制导语句来表明并行代码区域的位置[12]。Kokkos[13] 和 RAJA[14] 等并行编程框架基于 C＋＋模板元技术，上层提供统一接口用于描述数据和并行结构，具体实现由框架软件在下层提供面向各种体系结构的支持[15]。领域特定语言（Domain Specific Language，DSL）编程框架针对特定领域应用提供简洁抽象的高层框架，使用 DSL 进行代码编写，由系统为各个平台生成并行代码[16]。目前典型的 DSL 有牛津大学的 OPS/OP2[17]、斯坦福大学的 Liszt[18] 以及 ETH 的 STELLA[19] 等。

2014 年，Khronos Group[20] 制定并发布了并行编程模型 SYCL[21]。Khronos Group 是一个由超过 150 家主流的硬软件公司所组成的开放、非盈利、会员驱动的工业协会。SYCL 是一个基于 OpenCL 基本概念的跨平台抽象层，允许开发人员在不同硬件目标（CPU，GPU 和 FPGA 等）之间重用代码，并且可以为特定加速器执行自定义调优。SYCL 的主机端代码和设备端代码位于单个源文件中，代码遵循现代 C＋＋规范，将数据并行性添加进 C＋＋中，支持与其他 C＋＋软件的自然合成。Intel 于 2019 年推出了名为 oneAPI[22] 的工具包，旨在为不同计算硬件提供统一的编程模型和应用程序接口。oneAPI 的核心是名为 DPC＋＋（Data Parallel C＋＋）的编程语言。DPC＋＋建立在 SYCL 和现代 C＋＋语言之上，可以视为 SYCL 规范的一个实现。使用 DPC＋＋语言编写的同一份代码可以在经过 DPC＋＋编译器编译后，在 Intel 的 CPU，GPU 以及 FPGA 等硬件上运行。随后，Codeplay[23] 基于开源 DPC＋＋编译器实现了对 NVIDIA GPU 的支持，扩展了 DPC＋＋的目标硬件范围。

近年来，很多研究人员在异构超级计算机上开展了 LBM 模拟的异构并行和性能优化工作研究。例如，Feichtinger 等[24] 在使用 GPU 进行加速的 Tsubame 2.0 超级计算机上实现了 LBM 的异构并行加速，并在 1 000 个以上的 GPU 上进行了强可扩展性以及弱可扩展性测试。Li 等[25] 在使用 Intel Xeon Phi 协处理器（即 MIC）进行加速的天河 2 号超级计算机上，对 LBM 开源代码 openLBMflow 进行了 CPU＋MIC 的异构协同并行实现及优化，并使用 2048 个结点进行可扩展性测试。目前针对 LBM 的异构并行在 GPU 上主要采用 CUDA 或 OpenCL 进行实现，在 MIC 上主要使用 Intel Offload 进行实现。

在 SYCL 推出后，一些研究人员对其可用性以及性能进行了初步探索和测试分析。Volokitin 等[26] 使用 SYCL 对 Boris 粒子推进算法进行实现，在 Intel CPU 以及 GPU 上均获得了预期性能。Marinelli 等[27] 将高度优化的 GPU 哈希连接算法移植为 SYCL 实现，使用 Intel CPU 以及 GPU 进行测试，并利用 SYCL 中工作组的概念，使用不同工作组大小对性能进行细粒度调优，体现了 SYCL 在性能以及可移植性之间具有较好的权衡。上述工作表明，使用 SYCL 进行软件的跨平台并行开发是切实可行的。

目前针对 SYCL 以及 DPC＋＋的研究工作主要围绕着与其他实现（CUDA，OpenMP 等）进行比较，对 SYCL 应用程序的优化和提升关注较少。针对这一现状，本文的贡献主要包括：

1）使用跨平台并行编程模型 SYCL 实现 LBM 模拟程序，使 LBM 模拟可以在多种计算硬件上并行执行，有效利用不同架构的计算资源。

2）在 CPU 上将 SYCL 基础并行内核实现与优化后的 OpenMP 实现进行性能对比，SYCL 实现的加速比达到了 2.91，证明了 SYCL 在不进行额外调优工作的情况下，可以提供良好的开箱即用性能。

3）通过调整 ND-range 并行内核中工作组大小与形状、计算到工作项多对一映射内核中每个工作项处理的格子数量以及形状，对 SYCL 实现进行细粒度调优，研究了不同参数配置对计算内核性能的影响。

4）对 SYCL 中优化方法可获得的性能提升进行量化与系统总结。与 SYCL 基础并行内核版本相比，ND-range 并行内核在 CPU 和 GPU 上分别取得了最高 1.45 与 2.23 的加速比，计算到工作项多对一映射内核分别在 CPU 和 GPU 上得到的最高加速比为 1.57 与 1.34。该结果表明，不同硬件架构适用不同的优化方法：CPU 适合使用计算到工作项多对一映射内核，GPU 适合采用 ND-range 并行内核。

2　LBM 多相流方法和实现流程

本文基于开源代码 openLBMflow[28] 进行 SYCL 版本的开发。openLBMflow 使用 D3Q19 离散模型以及 Shan-Chen BGK 单松弛时间碰撞模型来实现三维多相流 LBM 模拟[29]，可以处理有重力情况下具有周期边界或者反弹边界的三维单相流或多相流问题。

2.1 格子 BGK 模型

LBM 中粒子的分布函数遵循格子玻尔兹曼方程，openLBMflow 中使用 BGK 模型模拟粒子碰撞，其单松弛时间的格子玻尔兹曼演化方程如下：

$$f_i(\boldsymbol{x}+\boldsymbol{c}_i\Delta t, t+\Delta t)-f_i(\boldsymbol{x},t)=-\frac{1}{\tau}[f_i(\boldsymbol{x},t)-f_i^{eq}(\boldsymbol{x},t)] \tag{1}$$

其中，τ 为松弛因子，$\boldsymbol{c}_i$ 为离散速度，f_i 为粒子分布函数，f_i^{eq} 为平衡态分布函数，其中 $i=0,1,\cdots,N$ 为离散速度方向，由 D3Q19 离散模型决定。D3Q19 离散模型的平衡态分布函数如下：

$$f_i^{eq}=\omega_i\rho\left[1+\frac{\boldsymbol{c}_i\cdot\boldsymbol{u}}{c_s^2}+\frac{(\boldsymbol{c}_i\cdot\boldsymbol{u})^2}{2c_s^4}-\frac{\boldsymbol{u}^2}{2c_s^2}\right] \tag{2}$$

其中，ω_i 为权系数，c_s 为无量纲声速。对于 D3Q19 离散模型，有：

$$c_s^2=c^2/3 \tag{3}$$

$$\omega_i=\begin{cases}1/3, & i=0\\ 1/18, & i=1,2,\cdots,6\\ 1/36, & i=7,8,\cdots,18\end{cases} \tag{4}$$

$$c_i=\begin{cases}(0,0,0)c, & i=0\\ (\pm1,0,0)c,(0,\pm1,0)c,(0,0,\pm1)c, & i=1,2,\cdots,6\\ (\pm1,\pm1,0)c,(\pm1,0,\pm1)c,(0,\pm1,\pm1)c, & i=7,8,\cdots,18\end{cases} \tag{5}$$

其中，$c=\Delta x/\Delta t$ 为格子速度，Δx 与 Δt 分别为格子步长与时间步长。

格子玻尔兹曼方程中流体的宏观密度及动量分别由式(6)与式(7)得到：

$$\rho=\sum_i f_i \tag{6}$$

$$\rho u=\sum_i c_i f_i \tag{7}$$

粘性系数 γ 与松弛因子 τ 的关系可以表示为：

$$\gamma=\left(\tau-\frac{1}{2}\right)c_s^2\Delta t \tag{8}$$

2.2 Shan-Chen 模型

openIBMflow 使用 Shan-Chen BGK 模型，引入粒子间相互作用力以及流体与固体间相互作用力。粒子间相互作用力为：

$$\boldsymbol{F}(x)=-G\varphi(\boldsymbol{x},t)\sum_i w_i\varphi(\boldsymbol{x}+c_i\Delta t,t)\boldsymbol{c}_i \tag{9}$$

流体与固体间相互作用力为：

$$\boldsymbol{F}_{\text{ads}}(x)=-G_{\text{ads}}\varphi(\boldsymbol{x},t)\sum_i w_i s(\boldsymbol{x}+\boldsymbol{c}_i\Delta t,t)\boldsymbol{c}_i \tag{10}$$

其中，G 为两相流体相互作用的强度；G_{ads} 为吸附参数；s 表示所在位置是否为固体；w_i 为权函数，由式(11)得出：

$$w_i=\begin{cases}0, & i=0\\ 1/18, & i=1,2,\cdots,6\\ 1/36, & i=7,8,\cdots,18\end{cases} \tag{11}$$

$\varphi(\boldsymbol{x},t)$ 表示相互作用势能，可表示为：

$$\varphi(\boldsymbol{x},t)=\varphi_0 e^{-\frac{\rho_0}{\rho}} \tag{12}$$

重力计算公式为：

$$\boldsymbol{F}_g(\boldsymbol{x})=\rho(\boldsymbol{x})g \tag{13}$$

其中，g 为重力加速度。

因此，多相流模拟中的体积力为：

$$\boldsymbol{F}_a(x)=\boldsymbol{F}+\boldsymbol{F}_{\text{ads}}+\boldsymbol{F}_g \tag{14}$$

平衡态速度为：

$$\boldsymbol{u}^{eq}=\boldsymbol{u}+\frac{\tau\boldsymbol{F}_a}{\rho} \tag{15}$$

流体真实速度为：

$$\boldsymbol{U}=\boldsymbol{u}+\frac{\boldsymbol{F}_\rho}{2\rho} \tag{16}$$

2.3 边界条件

模拟过程中周期网格被视为正常内部网格，固壁边界使用标准反弹规则。图 1 给出了二维反弹示例，其中箭头表示分布函数在不同方向上的分量。可以看到，固体边界上向外的分量会变为完全相反的方向，以模拟固体表面的反弹；其余分量则迁移到相邻格子上。

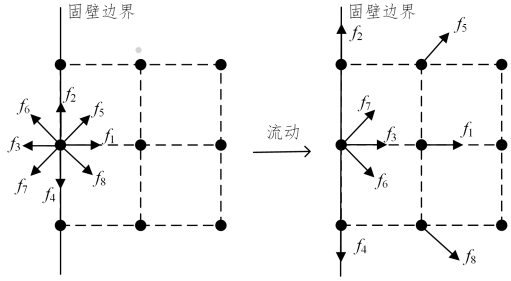


图 1 二维固壁边界反弹示例

Fig. 1 Illustration of 2D bounce-back boundary

2.4 实现流程

openLBMflow 的主要计算流程可以分为 3 个部分(见图 2)：流场初始化、时间迭代以及后处理。其中，时间迭代部分包含 3 个计算密集部分：1)计算粒子速度 u、密度 ρ 以及粒子间势能 φ(*calculate_phi*)；2)计算粒子分布函数 f_n(*calculate_fn*)；3)边界反弹更新(*update_boundary*)。上述 3 个计算部分使用多重嵌套 for 循环的方式来实现，其中 *calculate_phi* 以及 *calculate_fn* 为典型模板计算。由于上述 3 个计算部分占据了模拟的绝大多数时间，因此对它们的并行化处理对于程序的加速是至关重要的，故接下来的实验测试主要针对这 3 个计算部分展开。

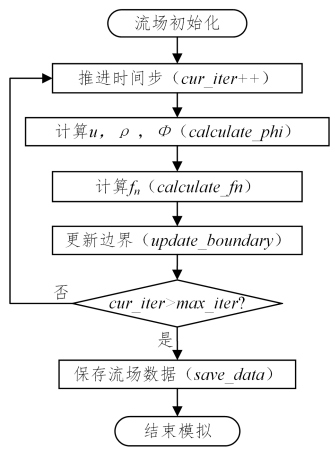


图 2 openLBMflow 模拟流程

Fig. 2 Simulation process of openLBMflow

3　SYCL 实现和性能优化方法

本章对实现过程中所使用到的 SYCL 特性进行概述，并且给出使用示例。由于对不同的嵌套 for 循环采用了相同的 SYCL 实现方法，因此仅使用 2.4 节中的 *calculate_fn* 计算部分作为示例。

3.1　SYCL 编程模型

SYCL 将不同类型的硬件设备统一抽象为设备(Device)，并将其作为加速卸载的目标。程序员可以根据具体情况手动选择所需设备类型(CPU，GPU 或 FPGA 等)，或者使用启发式方法进行设备选择(默认选择计算性能最强的硬件)。SYCL 程序的第一步需要创建一个队列(Queue)，队列是一个将工作提交给设备的机制。一个队列只能和一个设备相关联，一个设备可以有多个相关联的队列。SYCL 通过将计算任务提交给一个队列，来将计算转移给一个设备执行。

SYCL 中提供了两种内存管理的方式：1)缓冲区与访问器(Buffer&Accessor)模式；2)基于指针的统一共享内存(Unified Shared Memory，USM)模式。由于 openLBMflow 中的数据访存是基于指针进行的，因此选择使用基于指针的统一共享内存实现方式来进行实现。这种方式提供类似 C 语言的内存接口，只需要替换 malloc 或 new 即可，便于移植现有的C/C++代码到 SYCL。统一共享内存模型提供了管理内存的显式和隐式模型，出于性能考量，本文在主机端和设备端使用 memcpy 方法进行显式内存复制。在 SYCL 实现的全部计算流程中，数据始终驻留在设备端，只有用户选择输出模拟结果时才将数据显式复制到主机端进行保存。

设备端代码被称为内核(Kernel)。内核允许一个操作的多个实例并行执行，以表达数据并行性，通过编译器和运行时系统将实例映射到硬件执行。基于内核的方法能够跨设备提供可移植性和较高的开发效率。SYCL 支持 3 种并行内核：1)基础并行内核；2)ND-range 并行内核；3)层次并行内核。由于层次并行内核与 ND-range 并行内核仅仅是书写表达方式不同，在性能上没有差异，因此本文仅对基础并行内核和 ND-range 并行内核进行实现与测试。需要注意的是，提交给队列的不同内核之间默认以异步方式执行，因此为了保证模拟结果的正确性，需要使用 depends_on 方法来显式指明内核间的依赖关系。

3.2　基础并行内核实现

基础并行内核是 SYCL 中最为简单的并行内核，目的是给开发人员提供一种快速高效的实现方法。在基础并行内核中，计算实例到硬件的映射全部由编译器和运行时系统自动决定，不需要任何开发人员的干预和决策，这是基础并行内核的最大特点。

图 3 给出了使用基础并行内核对 *calculate_fn* 计算的抽象实现。首先使用 depends_on 函数声明 *calculate_fn* 依赖于 *calculate_phi* 的计算，*calculate_fn* 内核需要等到 *calculate_phi* 计算内核全部完成才可以开始执行。基础并行内核所使用的 parallel_for 函数包括两个参数：第一个参数使用 range 类描述并行执行的迭代空间，指明每个维度的大小；第二个参数使用 item 类，表示内核函数的单个实例。通过它使用 get_id 方法获得当前实例在 range 中的索引，结合指针来访问对应位置的数据以完成计算。对于内部格子，即 solid_device[x][y][z]=0 的格子，进行碰撞与流动。

```
/*基础并行内核实现 */
1. event calculate_fn=myqueue.submit([&](handler& h){
2.    h.depends_on(calculate_phi);//指明所依赖内核
3.    h.parallel_for(range{x_range,y_range,z_range},//指明迭代空间
4.                   [=](item<3> it){ //内核实例
5.      //通过实例获取索引
6.      size_t x=it.get_id(0);
7.      size_t y=it.get_id(1);
8.      size_t z=it.get_id(2);
9.      //通过索引进行对应计算
10.     if(solid_device[x][y][z]==0){
11.         collision(x,y,z);
12.         streaming(x,y,z);
13. }});});
```

图 3　基础并行内核的实现

Fig. 3　Implementation of basic parallel kernel

可以看到使用基础并行内核表示数据并行性十分简单，但这种易于实现的特性也牺牲了对底层软硬件的控制，例如映射与调度等。因此，对于较为复杂的计算，基础并行内核的性能难以判断。

3.3　ND-range 并行内核实现

基础并行内核无法支持硬件级别的优化，SYCL 提供了 ND-range 并行内核来提供对底层性能调优的支持。ND-range 并行内核可以将实例分为不同类型的分组，并将它们精确地映射到硬件平台上，适合在内核中利用一定的局部性。在 ND-range 并行内核中，全部迭代空间被划分为工作组(Work-group)，工作组又被划分为一维的子组(Sub-group)，子组由若干个工作项(Work-item)构成。图 4 为 2 维 8*8 大小的迭代空间被划分为 4*4 大小的工作组、1*4 大小的子组以及 1*1 大小的工作项的示意图。

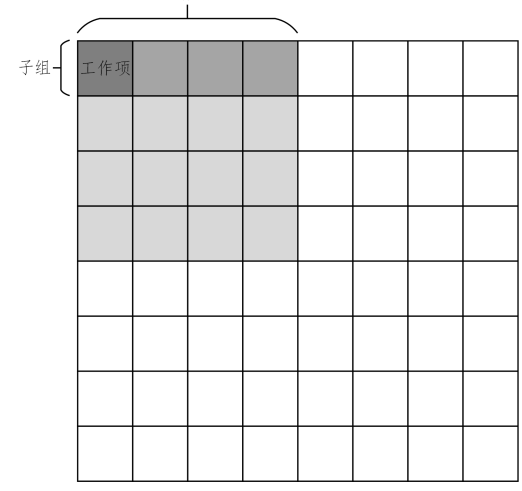


图 4　工作组、子组、工作项关系图

Fig. 4　Relation of work-group，sub-group and work-item

对于不同硬件设备，ND-range 并行内核会将上述分组映射到不同层次的计算硬件上。对于多核 CPU 而言，编译器和运行时系统会将不同工作组映射到不同计算核心上，子组则

使用 SIMD 运算单元进行计算，子组中不同工作项则对应矢量计算单元中的每个通道。在 GPU 上，一个工作组将会被映射到一个流处理器上，每个子组为一个线程束，工作项则被映射到一个个计算核心上。上述映射关系由编译器和运行时系统自动完成，无法通过编程控制。除了上述对硬件的映射外，工作组以及子组还支持组内的通信和同步，本文 LBM 模拟的实现中没有涉及，因此不加以赘述。

图 5 给出了 ND-range 并行内核的抽象实现。ND-range 并行内核同样使用 parallel_for 函数表示。第一个参数使用 nd_range 类分别指出全局迭代空间 global 以及局部迭代空间 local 的大小，也可以称 local 为工作组大小。nd_item 类与 item 类功能类似，表示内核函数的单个实例，提供查询全局以及局部索引功能。与基础并行内核不同的是，此处需要使用 get_global_id 方法获得全局索引。由于子组被映射到 SIMD 硬件上，子组大小与向量宽度有关，由系统自动根据硬件平台确定。本文主要针对工作组大小和形状进行调整，以测试工作组配置对 LBM 性能的影响。

```
/* ND-range 并行内核实现 */
1. event calculate_fn=myqueue.submit([&](handler& h){
2.   h.depends_on(calculate_phi);//指定所依赖内核
3.   range<3> global{x_range,y_range,z_range};//全局迭代空间
4.   range<3> local{x_local,y_local,z_local};//给定工作组大小
5.   h.parallel_for(nd_range<3>{global,local},//给定全局与局部大小
6.                  [=](nd_item<3> it) {//内核实例
7.     //通过实例获取索引
8.     size_t x=it.get_global_id(0);
9.     size_t y=it.get_global_id(1);
10.    size_t z=it.get_global_id(2);
11.    //通过索引进行对应计算
12.    if(solid_device[x][y][z]==0){
13.      collision(x,y,z);
14.      streaming(x,y,z);
15. }});});
```

图 5 ND-range 并行内核的实现

Fig. 5 Implementation of ND-range parallel kernel

3.4 计算到工作项的多对一映射实现

在上述两种并行内核中，并行内核的迭代空间与 LBM 中的格子是一一对应的，这种情况被称为计算与工作项的一对一映射。Reinders 等[30]提出了一种计算与工作项多对一的映射方式，用于提高性能，我们使用基础并行内核对其进行了实现，如图 6 所示。这种实现方式与 3.2 节和 3.3 节中实现的最大不同是在内核中又包含了一层 for 循环，此时每个并行实例需要处理多个格子，以共同实现 LBM 中全部格子的计算。在该种实现中，我们通过改变工作项数量来考查每个工作项处理格子的数量对性能的影响。注意循环增量为工作项的数量，此时每个工作项所处理的格子并不连续，在 CPU 中会导致缓存命中率低，因此这种实现方法不适用于 CPU 硬件。后续实验测试也验证了上述分析。

通过对上述算法的修改实现了更加适用于 CPU 版本的代码，如图 7 所示。这里并行内核中加入了三重 for 循环，且每层循环的增量均为 1，使得每个工作项能够处理在空间中连续的格子块(代码中省略边界检查部分)，具备更好的局部性，有效利用了 CPU 缓存，提高缓存命中率。第 4 章的实验结果也验证了上述实现在 CPU 上的性能优势。

```
/* GPU 上计算到工作项的多对一映射 */
1. event calculate_fn=myqueue.submit([&](handler& h){
2.   h.depends_on(calculate_phi);//指定所依赖内核
3.   size_t N=x_range * y_range * z_range;//计算格子总数
4.   size_t W=worker;//给定工作项数量
5.   h.parallel_for(range{W},[=](item<1> it){
6.     size_t id=it.get_id(0);//通过实例获取索引
7.     //循环计算并行实例对应的多个格子
8.     for(size_t i=id;i<N;i+=W){
9.       //计算格子索引
10.      size_t x=i/(y_range * z_range);
11.      size_t y=i %(y_range * z_range) / z_range;
12.      size_t z=i % z_range;
13.      //通过索引进行对应计算
14.      if(solid_device[x][y][z]==0){
15.        collision(x,y,z);
16.        streaming(x,y,z);
17. }}});});
```

图 6 GPU 上计算到工作项的多对一映射

Fig. 6 Many-to-one mapping computation to work-items on GPU

```
/* GPU 上计算到工作项的多对一映射 */
1. event calculate_fn=myqueue.submit([&](handler& h){
2.   h.depends_on(calculate_phi);//指定所依赖内核
3.   range<3> worker{x_worker,y_worker,z_worker};//给定工作项数量
4.   h.parallel_for(worker,[=](item<3> it){
5.     //获取实例索引
6.     size_t x_id=it.get_id(0);
7.     size_t y_id=it.get_id(1);
8.     size_t z_id=it.get_id(2);
9.     //使用三重循环计算并行实例对应的多个格子
10.    for(size_t x=x_id * x_size;x<(x_id+1) * x_size;x++){
11.      for(size_t y=y_id * y_size;y<(y_id+1) * y_size;y++){
12.        for(size_t z=z_id * z_size;z<(z_id+1) * z_size;z++){
13.          //通过索引进行对应计算
14.          if(solid_device[x][y][z]==0){
15.            collision(x,y,z);
16.          streaming(x,y,z);
17. }}}}});});
```

图 7 CPU 上计算到工作项的多对一映射

Fig. 7 Many-to-one mapping computation to work-items on CPU

4 实验测试

4.1 实验配置

本文在 CPU 和 GPU 上分别对基于 SYCL 的 LBM 并行模拟及其优化进行了测试分析。CPU 平台采用了中国科技云超算 BSCC-T 分区的单个计算节点，包含 2 个 Intel Xeon Platinum 9242 CPU，频率为 2.30 GHz，关闭超线程，共 96 个核。CPU 平台的 SYCL 版本所使用的编译器为 Intel oneAPI 2022.1 版本的 Intel oneAPI DPC++/C++ Compiler 2022.0.0(2022.0.0.20211123)，编译参数为-O3 -fsycl -std=c++17。作为 CPU 平台性能基准的 OpenMP 版本使用的编译器为 Intel oneAPI 2022.1 版本的 icc(ICC)2021.5.020211109，编译参数为-O3 -qopenmp。GPU 平台采用了

中国科技云超算 N26 分区的单个 GPU 卡进行,该 GPU 为 NVIDIA Tesla V100-SXM2 GPU,频率为 1.26GHz,具有 80 个 SM,32GB 显存。在 GPU 上使用 Codeplay 扩展的编译器实现[31],使用该项目的 sycl 分支 20dbc70 提交版本进行编译。编译参数为-O3 -fsycl -std=c++17 -fsycl-targets=nvptx64-nvidia-cuda -Xsycl-target-backend --cuda-gpu-arch=sm_70,基于 CUDA 后端进行编译。LBM 并行模拟的测试网格规模为 512 * 512 * 256,对第 2 章所述的 3 个主要计算部分进行 10 次迭代,执行 5 次并取平均值。

4.2　结果分析

4.2.1　基础并行内核版本

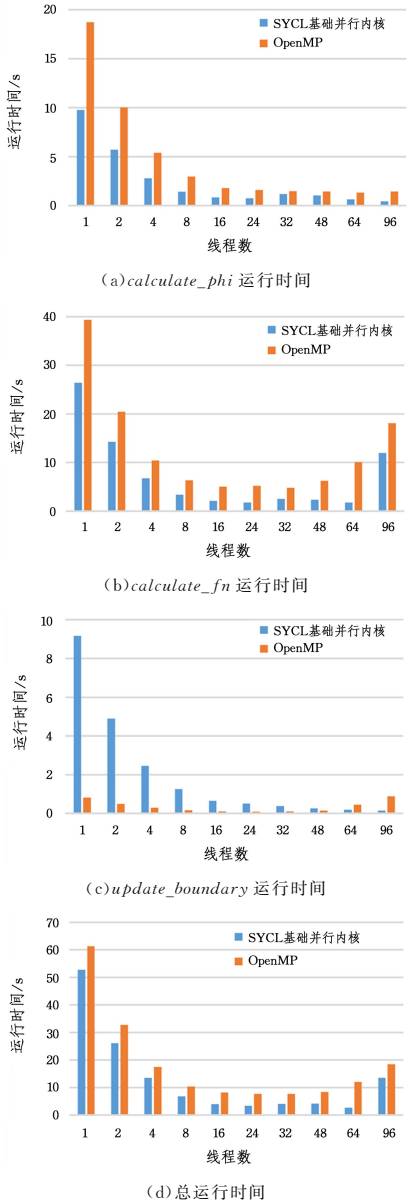
在基础并行内核实现测试中,使用了 Wang 等[32-33]优化后的 OpenMP 并行版本的 LBM 程序作为基准,该版本在主要的 3 个计算部分嵌套 for 循环上直接加入 OpenMP 制导语句,将内部格子与边界格子拆分计算,并且使用循环分块方式进行优化。如图 8 所示,通过分块将 $X*Y*Z$ 大小的块切分为 $TX*TY*TZ$ 大小的块,通过修改 TX,TY 和 TZ 的大小,使得数据能够较好地适应缓存。该方法是层次存储系统中提高数据重用的常见方法。

```
/*OpenMP 版本实现 */
1. #pragma omp parallel for
2. for(xs=0,xs < X;xs +=TX){
3.   xe=min(xs + TX,X);
4.   for(ys=0;ys < Y;ys +=TY){
5.     ye=min(ys + TY,Y);
6.     for(zs=0;zs < Z;zs +=TZ){
7.       ze=min(zs + TZ,Z);
8.       #pragma omp for
9.       for(x=xs;x < xe;x++)
10.        for(y=ys;y < ye;y++)
11.          #pragma simd
12.          #pragma ivdep
13.          for(z=zs;z < ze;z++){
14.            //通过索引 x,y,z 进行计算
15.          }}}}
```

图 8　OpenMP 版本中循环分块的实现

Fig. 8　Implementation of loop tiling in OpenMP version

图 9 分别展示了 OpenMP 版本与 SYCL 基础并行内核版本在 3 个计算部分的运行时间以及总体的运行时间,由于还包含其他计算步骤,因此总体运行时间并不等于 3 个计算部分运行时间之和。在计算量最大的 *calculate_phi* 与 *calculate_fn* 计算部分,SYCL 基础并行内核的性能在不同线程数配置下均优于 OpenMP 版本。*calculate_phi* 中 SYCL 基础并行内核在 96 线程时获得最短运行时间 0.43 s,OpenMP 在 64 线程时获得最短运行时间 1.34 s。*calculate_fn* 中 SYCL 基础并行内核在 64 线程时获得最短运行时间 1.73 s,OpenMP 在 32 线程时获得最短运行时间 4.80 s。*update_boundary* 计算部分则相反,OpenMP 版本明显优于 SYCL 基础并行内核。其原因可能是 DPC++默认线程模型 Intel oneAPI Threading Building Blocks(Intel TBB)所使用的动态调度开销高于 OpenMP 静态调度开销,且 *update_boundary* 只在边界格子上进行计算,相比前两者计算量小,导致并行加速无法抵消掉并行开销。总体性能表现与计算时间占比最大的 *calculate_fn* 部分类似。CPU 上 OpenMP 版本在使用 24 个线程时获得的最优性能为 7.73 s;SYCL 在仅使用 8 个线程时即可获得比 OpenMP 最佳性能更短的执行时间,为 6.80 s。SYCL 在 CPU 上使用 64 线程时获得的最佳性能为 2.66 s,在 GPU 上的执行时间为 1.98 s,相比 OpenMP 版本分别实现了 2.91 与 3.90 倍的加速且不需要任何额外的优化工作。该实验的结果表明,SYCL 最为基础的并行内核实现,在没有开发人员手工优化的情况下,开箱即用性能较传统 OpenMP 实现具有明显优势。



(a)*calculate_phi* 运行时间

(b)*calculate_fn* 运行时间

(c)*update_boundary* 运行时间

(d)总运行时间

图 9　OpenMP 与 SYCL 基础并行内核版本的运行时间

Fig. 9　Runtime of OpenMP and SYCL basic parallel kernel

图 10 分别给出了使用 SYCL 基础并行内核实现的 3 个计算过程以及总体的强可扩展性结果。可以看出,*update_boundary* 具有较好的强扩展性,加速比随线程数增多而增大,最大加速比为60.3。*calculate_phi* 与 *calculate_fn* 的加速比在线程数达到 32 后先减小而后增大,*calculate_phi* 在 96 线程时获得最高加速比为 22.7,*calculate_fn* 在 64 线程时

获得最高加速比为 15.2。由于 *calculate_fn* 的计算占据总体计算时间比例最大，因此总体加速比变化趋势与 *calculate_fn* 相似，在 32 线程后先减小再增大，并在 64 个线程时获得最大 19.8 的加速比。

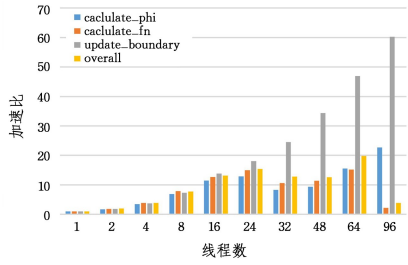


图 10 SYCL 基础并行内核的强可扩展性

Fig. 10 Strong scaling of SYCL basic parallel kernel

4.2.2 ND-range 并行内核

SYCL 的 ND-range 并行内核需要开发人员显式给定一个工作组大小作为参数，图 11 和图 12 分别给出了 CPU 和 GPU 上以基础并行内核的最佳性能为基准，采用不同工作组大小所获得的加速比。

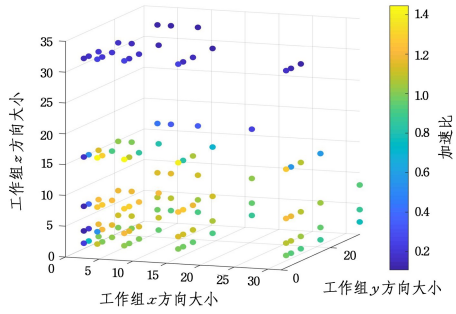


图 11 CPU 上 ND-range 并行内核的性能

Fig. 11 Performance of ND-range parallel kernel on CPU

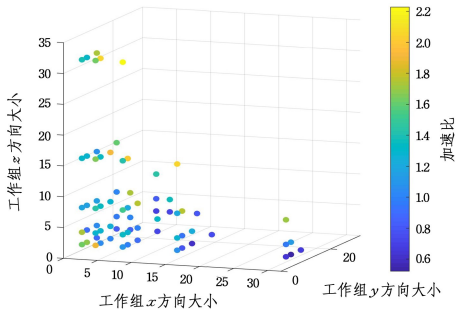


图 12 GPU 上 ND-range 并行内核性能

Fig. 12 Performance of ND-range parallel kernel on GPU

SYCL 中不同的设备在执行 ND-range 并行内核时的工作组大小受硬件限制。本文测试使用的 CPU 平台工作组大小不能超过 8192，即工作组 x,y,z 3 个方向的大小乘积不能超过 8192。因此针对 CPU 平台，我们在不同维度上分别取值 2,4,8,16 和 32 进行排列组合，跳过 16 * 32 * 32 和 32 * 32 * 32 等超出限制的工作组大小，设置线程数为 96，以针对不同大小的工作组进行测试。图 11 给出了 CPU 上 ND-range 并行内核的性能，其中颜色越深表示该工作组配置的性能越差，颜色越浅表示该工作组配置的性能越好，可以看到工作组大小对性能有明显影响。对于 CPU 测试的 121 个工作组大小，有 56 个工作组大小配置与基础并行内核相比实现了加速，其中工作组大小为 8 * 2 * 16 时运行时间为 1.84 s，获得了最大的加速比(1.45)。由图 11 可以看出，加速比低的配置主要集中在工作组 z 方向较大的部分，其中工作组配置 8 * 32 * 32 的加速比最低(0.107)，运行时间为 24.8s。使用 SYCL 所提供的 ND-range 并行内核可以简单地实现类似于循环分块的优化方法。

对于本文采用的 GPU 平台，最大工作组大小限制为 1024，但在实际测试中，我们发现工作组大小超过 600 就可能导致程序运行异常，给出的 CUDA 错误编码为 701。具体解释为 CUDA ERROR_LAUNCH_OUT_OF_RESOURCES，通常原因是启动内核时指定线程数量过多或寄存器的总数超过了可使用的最大值，因此本文 GPU 平台测试时限制工作组最大为 512。与 CPU 类似，在不同维度依旧选取 2,4,8,16 和 32 作为工作组大小，共对 72 个工作组大小进行了测试，结果如图 12 所示。其中有 56 个工作组配置相比 GPU 基础并行内核版本实现了加速，工作组配置为 8 * 2 * 32 时运行时间为 0.89 s，加速比为 2.23，实现了最大程度的性能提升。而最慢的工作组配置为 32 * 4 * 2，运行时间为 3.80 s，加速比为 0.52。由此可见，工作组大小的配置优化对 GPU 同样重要。

此外，通过测试发现，对于 CPU 而言，工作组大小不能整除完整迭代空间的配置是可以正常运行的，但 GPU 上的测试会报错，提示 PI_ERROR_INVALID_WORK_GROUP_SIZE。这表明目前 SYCL 对不同硬件架构的支持处于不同阶段，并不完全一致，针对 CPU 的支持相对而言更加完善。

4.2.3 计算到工作项的多对一映射

图 13 与图 14 分别给出了在 CPU 和 GPU 上使用 3.4 节中所描述的计算到工作项目的多对一映射方法进行实现的性能，同样以基础并行内核为基准。在 CPU 平台上同样使用 96 个线程，对每个工作项所处理的格子数量进行了指定，每个方向分别取值为 1,2,4 和 8，共进行了 64 组测试，结果有 54 组实现了加速，使用该种方法的绝大部分参数配置都可以获得性能的提升。其中，每个工作项处理 4 * 2 * 1 个格子时相对于基础并行内核获得了最大加速比，加速比为 1.57，运行时间为 1.69 s。该优化相比 ND-range 并行内核，在 CPU 上取得了更好的效果。得到这样的结果的原因是每个工作项处理空间上连续的格子，对于算法中所包含的典型模板计算，拥有与 ND-range 并行内核同样好的数据重用性以及局部性。同时考虑到 CPU 上进行线程切换需要保存和加载寄存器内存，导致开销较大，该方法减少了总工作项数量，从而减小了线程切换开销，进而加速程序运行，提高了性能。

图 14 给出了 GPU 上计算到工作项的多对一映射的结果，其中性能表现最好的是每个工作项处理两个不连续的格子，相比基础并行内核可以获得 1.34 的加速比，总运行时间为 1.48s。这个结果明显低于采用 ND-range 并行内核的最优性能。不同于 CPU，GPU 对每个线程使用不同的硬件寄存器，切换线程时不需要保存和加载寄存器内存，因此线程切换高效，故减少工作项数量以减少线程切换的方法在 GPU 上无法获得明显的性能收益，同时内核中所包含的循环计算

还会增加额外的计算成本。这就造成了该方法相比 ND-range 并行内核所得到的加速比更低。

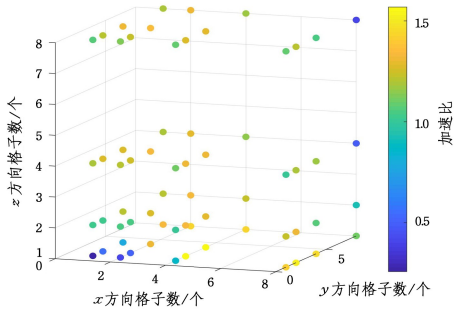


图 13　CPU 上计算到工作项多对一映射性能

Fig. 13　Performance of many-to-one mapping computation to work-items on CPU

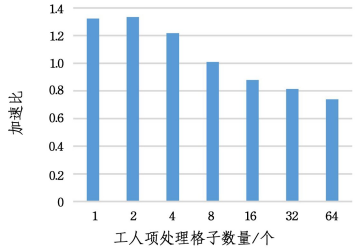


图 14　GPU 上计算到工作项多对一映射性能

Fig. 14　Performance of many-to-one mapping computation to work-items on GPU

结束语　本文基于 SYCL 实现了多项流 LBM 模拟的单源异构跨平台并行。整体而言，采用 SYCL 移植现有 C/C++代码开发难度小，移植速度快。测试表明，SYCL 基础并行内核版本相对于传统 OpenMP 优化实现，在 CPU 上获得了 2.91 倍的加速，表明了 SYCL 作为新型并行编程模型的性能优势。通过 ND-range 并行内核以及计算到工作项的多对一映射优化，CPU 上的性能相比基础并行内核版本提升了多达 1.57 倍，在 GPU 上性能提升了多达 2.23 倍。SYCL 作为新型跨平台编程模型，当前还处于发展完善过程，未来我们将重点研究 SYCL 相关的优化方法，并将 SYCL 应用于大型复杂数值模拟应用的异构并行。

参 考 文 献

[1] ANDERSON J D. Computational Fluid Dynamics: The Basics with Applications[M]. New York: McGraw-Hill, 1995: 1-30.

[2] SUCCI S, BENZI R, HIGUERA F. The Lattice Boltzmann Equation: A New Tool for Computational Fluid-Dynamics[J]. Physica D, 1991, 47(1/2): 219-230.

[3] MILIANI S, MONTESSORI A, ROCCA M L, et al. Dam-Break Modeling: LBM as the Way towards Fully 3D, Large-Scale Applications[J]. Journal of hydraulic engineering, 2021, 147(5): 1-17.

[4] SARITHA G, BANERJEE R. Development and Application of a High Density Ratio Pseudopotential Based Two-phase LBM Solver to Study Cavitating Bubble Dynamics in Pressure Driven Channel Flow at Low Reynolds Number[J]. European Journal of Mechanics B: Fluids, 2019, 75: 83-96.

[5] BUDINSKI L. Application of the LBM with Adaptive Grid on Water Hammer Simulation[J]. Journal of Hydroinformatics, 2016, 18(4): 687-701.

[6] ZHIS. Impact of Mesh Partitioning Methods in CFD for Large Scale Parallel Computing[J]. Computers & Fluids, 2014, 103: 1-5.

[7] LEE S, GOUNLEY J, RANDLES A, et al. Performance Portability Study for Massively Parallel Computational Fluid Dynamics Application on Scalable Heterogeneous Architectures[J]. Journal of Parallel and Distributed Computing, 2019, 129: 1-13.

[8] CUDA Official [OL]. [2023-02-01]. https://developer.nvidia.com/cuda-toolkit.

[9] OpenCL Official[OL]. [2023-02-01]. https://opencl.org/.

[10] TIAN W, SEVILLA T A, ZUO W. A Systematic Evaluation of Accelerating Indoor Airflow Simulations using Cross-Platform Parallel Computing[J]. Journal of Building Performance Simulation, 2017, 10(3): 243-255.

[11] OpenACC Official[OL]. [2023-02-01]. https://www.openacc.org/.

[12] MATSUFURU H, AOKI S, AOYAMA T, et al. OpenCL vs OpenACC: Lessons from Development of Lattice QCD Simulation Code[J]. Procedia Computer Science, 2015, 51: 1313-1322.

[13] Kokkos Official[OL]. [2023-02-01]. https://kokkos.org/.

[14] RAJA Documentation[OL]. [2023-02-01]. https://raja.readthedocs.io/.

[15] REGULY I Z, MUDALIGE G R. Productivity, Performance, and Portability for Computational Fluid Dynamics Applications [J]. Computers & Fluids, 2020, 199: 104425.

[16] MACIÀ S, MARTÍNEZ-FERRER P J, AYGUADÉ E, et al. Automated Generation of High-Performance Computational Fluid Dynamics Codes[J]. arXiv: 2204.12120v2, 2022.

[17] OPS/OP2 Official[OL]. [2023-02-01]. https://op-dsl.github.io/.

[18] DEVITO Z, JOUBERT N, PALACIOS F, et al. Liszt: A Domain Specific Language for Building Portable Mesh-Based PDE Solvers[C]// Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. California: IEEE Computer Society, 2011: 1-12.

[19] MOHR D, STEFANOVIC D. Stella: A Python-Based Domain-Specific Language for Simulations[C]// Proceedings of the 31st Annual ACM Symposium on Applied Computing. New York: Association for Computing Machinery, 2016: 1952-1959.

[20] Khronos Group Official [OL]. [2023-02-01]. https://www.khronos.org/.

[21] SYCL Official[OL]. [2023-02-01]. https://www.khronos.org/sycl/.

[22] oneAPI Official[OL]. [2023-02-01]. https://www.oneapi.io/.

[23] Codeplay Official[OL]. [2023-02-01]. https://codeplay.com/.

[24] FEICHTINGER C, HABICH J, KOESTLER H, et al. Performance Modeling and Analysis of Heterogeneous Lattice Boltzmann Simulations on CPU-GPU Clusters[J]. Parallel Computing, 2015, 46(7): 1-13.

[25] LI D, XU C, WANG Y, et al. Parallelizing andOptimizing Large-Scale 3D Multi-phase Flow Simulations on the Tianhe-2 Super-

computer[J]. Concurrency & Computation Practice & Experience, 2016, 28(5): 1678-1692.
[26] VOLOKITIN V, BASHINOV A, EFIMENKO E, et al. Parallel Computing Technologies [M]. Springer International Publishing, 2021: 288-300.
[27] MARINELLI E, APPUSWAMY R. XJoin: Portable, Parallel Hash Join across Diverse XPU Architectures with oneAPI [C]// Proceedings of the 17th International Workshop on Data Management on New Hardware. Virtual Event China: ACM, 2021: 1-5.
[28] openLBMflow Repository[OL]. [2023-02-01]. https://sourceforge.net/projects/lbmflow/.
[29] CHEN S, MARTINEZ D, MEI R. On Boundary Conditions in Lattice Boltzmann Methods[J]. Physics of Fluids, 1996, 8(9): 2527-2536.
[30] REINDERS J, ASHBAUGH B, BRODMAN J, et al. Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL[M]. Berkeley, CA: Apress, 2021: 125-126.
[31] Compiler Repository[OL]. [2023-02-01]. https://github.com/intel/llvm.
[32] XU C, WANG X, LI D, et al. Openmp4.5-Enabled Large-Scale Heterogeneous Lattice Boltzmann Multiphase Flow Simulations [C]// Proceedings of 2019 IEEE International Conference on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom). California: IEEE Computer Society, 2019: 1007-1016.
[33] WANG X. Parallel Collaborative Algorithm for Large-Scale LBM Multiphase Flow on Heterogeneous Many-Core Platform [D]. Changsha: National University of Defense Technology, 2018.

DING Yue, born in 1999, postgraduate. Her main research interests include parallel and high performance computing applications and so on.

XU Chuanfu, born in 1980, Ph.D, associate professor, is a member of China Computer Federation. His main research interests include parallel computing and applications and so on.

(责任编辑:柯颖)