2023, **24** (1), 37–54. doi 10.26089/NumMet.v24r104

doi 10.26089/NumMet.v24r104 УДК 519.688

Моделирование методом частиц на GPU с использованием языка GLSL

А. В. Озерицкий
Яндекс, Москва, Российская Федерация
Университетская гимназия (школа-интернат) МГУ имени М. В. Ломоносова, Москва, Российская Федерация
ORCID: 0000-0002-8194-6855, e-mail: aozeritsky@gmail.com

Аннотация: Рассмотрено моделирование гравитационной задачи N тел с использованием алгоритмов PM и P^3M. Реализация алгоритмов для GPU осуществлена с применением вычислительных шейдеров. Предложенный подход использует CPU-код только для синхронизации и запуска шейдеров и не содержит вычислительных частей, реализуемых на CPU; в том числе полностью отсутствует копирование данных между CPU и GPU. Приводятся параллельный алгоритм размещения частиц по ячейкам сетки и параллельный алгоритм распределения масс по узлам сетки. Основой алгоритмов является параллельное построение списков, соответствующих ячейкам сетки. Алгоритмы полностью распараллелены и не содержат частей, исполняемых в один поток. Для расчета одновременно с визуализацией часть вычислений сделана в вершинном шейдере. Выполнить их позволило использование буферных объектов в вершинном шейдере и специально подготовленных данных вместо вершин в качестве входа. Приведены результаты численных расчетов на примере образования галактических скоплений в расширяющейся согласно модели Фридмана плоской вселенной. В качестве модели вселенной брался куб с периодическими краевыми условиями по всем осям. Максимальное число частиц, с которым проводились расчеты, — 10^8. Для моделирования использовались современный кроссплатформенный API Vulkan и язык GLSL. Результаты расчетов на процессорах Apple M1 и Ryzen 3700X сравниваются с результатами расчетов на обычных видеокартах Apple M1 и NVIDIA RTX 3060. Параллельный алгоритм для CPU реализован с помощью OpenMP. Проведено сравнение производительности алгоритма с результатами других авторов, причем делаются качественные сравнения самих результатов вычислений и сравнение времени работы алгоритмов. Также приведено сравнение времени работы программы для GPU и похожей программы для кластера из многих узлов.

Ключевые слова: моделирование методом частиц, параллельное программирование, задача N тел, язык GLSL, API Vulkan, уравнение Пуассона.

Для цитирования: Озерицкий А.В. Моделирование методом частиц на GPU с использованием языка GLSL // Вычислительные методы и программирование. 2023. **24**, № 1. 37–54. doi 10.26089/NumMet.v24r104.

2023, **24** (1), 37–54. doi 10.26089/NumMet.v24r104

Computational simulation using particles on GPU and GLSL language

Aleksei V. Ozeritskii'
Yandex, Moscow, Russia
Gymnasium of Moscow State University, Moscow, Russia
ORCID: 0000-0002-8194-6855, e-mail: aozeritsky@gmail.com

Abstract: The N-body problem simulation using PM and P^3M algorithms is provided. A GPU implementation of an algorithm using compute shaders is provided. This algorithm uses the CPU for synchronizing and launching the shaders only, whereas it does not contain computational parts implemented on the CPU. That also includes no data copying between the GPU and CPU. Parallel algorithms for placing particles in grid cells and mass distribution in grid nodes are presented. The algorithms are based on parallel construction of linked lists corresponding to grid cells. The algorithms are completely parallel and do not contain sequential parts. Some calculations are done in a vertex shader to compute simultaneously with visualization. This was done with the help of shader buffer objects as well as specially prepared data instead of vertices as vertex shader input. The results of the numerical calculations using galaxy cluster formation based on a flat expanding Friedmann model universe are presented as an example. A cube with periodic boundary conditions on all axes was used as an example of a model universe. The maximum particle amount used in calculations is 10^8. The modern cross platform API Vulkan and GLSL language were used for simulation purposes. The numerical calculations are compared using the Apple M1 and Ryzen 3700X processors, with the results using regular video cards — Apple M1 and NVIDIA RTX 3060. The parallel algorithm for the CPU is implemented using OpenMP. Algorithmic efficiency is compared to the results by other authors. The results of the algorithm are compared to the results of other authors, and the qualitative results and execution time are also compared. A comparison of the running time of programs for the GPU with a similar cluster program with many nodes is given.

Keywords: particles simulation, parallel programming, N body problem, GLSL language, Vulkan API, Poisson equation.

For citation: A. V. Ozeritskii', "Computer simulation using particles on GPU with GLSL," Numerical Methods and Programming. **24** (1), 37–54 (2023). doi 10.26089/NumMet.v24r104.

1. Введение. Во многих прикладных областях возникает потребность в исследовании поведения большого числа взаимодействующих частиц. С математической точки зрения существуют два подхода для описания такого взаимодействия. Подход Эйлера не различает сами частицы, но вместо этого строит полевое описание. В каждой геометрической точке пространства задаются такие величины, как скорость, ускорение, температура и т.д. Затем прослеживается эволюция этих величин с течением времени. Подход Лагранжа состоит в описании параметров самих точек среды. Описываются координаты, скорости, ускорения и т.д., но уже для каждой "частицы" среды, а не для геометрической точки пространства.

Подход Лагранжа стал приобретать популярность с развитием вычислительной техники. На этом подходе основано моделирование методом частиц. Метод частиц применяется в различных областях, в частности при моделировании плазмы, моделировании полупроводников, в астрофизике, в молекулярной динамике. Метод частиц прост для реализации на компьютере и очень производителен.

Одной из первых монографий по указанной теме является работа 1981 г. Хокни и Иствуда [1]. В работе описываются алгоритмы PM и P^3M, а также постановки физических задач, которые можно решать с помощью данных алгоритмов. Работа фактически заложила основы моделирования по методу частиц. Суть алгоритма PM состоит в том, что заряды или массы частиц распределяются по сетке, потом решаются уравнения поля (например, уравнение Пуассона), после чего обновляются положения и скорости частиц. В алгоритме P^3M сила расщепляется на короткодействующую и сеточную. Расчет сеточной си-

2023, **24** (1), 37–54. doi 10.26089/NumMet.v24r104

лы осуществляется как в алгоритме PM, а короткодействующая сила считается с помощью стандартного решения задачи N тел в ограниченной области.

В статье будет описан расчет взаимодействия очень большого числа частиц по законам ньютоновской гравитации. Данная задача обычно возникает в областях астрофизики и космологии. Приведем основные работы по указанной тематике.

В работе [2] описана программа для компьютеров с распределенной памятью, предназначенная для расчета крупномасштабной структуры вселенной. В частности, проводились расчеты взаимодействия миллиарда частиц на кластере из 4096 ядер. Работа [3] также описывает алгоритм для компьютеров с распределенной памятью. Приведены результаты расчета для 40000000 частиц, анализируется прирост производительности при увеличении числа ядер.

В [4] рассматривается эволюция вселенной по модели λCDM. Приведены результаты численных расчетов, в том числе есть иллюстрации с полученной крупномасштабной структурой.

Работа [5] применяет алгоритм P^3M для решения плоской задачи N тел. В частности, изучается гравитационное линзирование. Приводятся результаты расчета и код на языке программирования Python.

Особо стоит отметить работу [6]. Данная работа описывает крупный open-source проект ENZO, который применяется в моделировании методом частиц во многих областях и в астрофизике в частности. Описывается астрофизическая модель λCDM, учитываются не только гравитационные поля, но также и магнитные. Уравнения поля решаются на адаптивных сетках. Проект поддерживает параллельное исполнение, в том числе на NVIDIA GPU с помощью технологии CUDA.

В настоящей работе рассматривается адаптация алгоритмов для видеокарт. Сейчас даже встроенные видеокарты в подобных расчетах могут обладать большей производительностью, чем CPU, поэтому адаптация алгоритмов для этих видеокарт и использование технологий, отличных от CUDA, для GPU-вычислений являются перспективными. Стоит отметить, что на рынке дискретных видеокарт кроме видеокарт фирмы NVIDIA есть и видеокарты фирмы AMD, которые обладают отличной производительностью, но при этом не поддерживают CUDA. Для GPU-расчетов также может применяться язык OpenCL. Недавно Apple признала технологию OpenCL устаревшей для своей продукции. Поэтому написание программы на OpenCL исключит продукцию Apple. Выпущенные в 2020 г. процессоры M1 обладают производительностью в 2.6 Tflops в GPU-режиме, поэтому не хотелось бы исключать их поддержку. В работе использовались язык GLSL и новый API Vulkan. В отличие от OpenGL, Vulkan является более гибким API, в частности он поддерживает режим работы с видеокартой без создания окон, что хорошо подходит для вычислений. Была написана программа, которая позволяет запускать GLSL-код как с помощью OpenGL, так и с помощью Vulkan. Программа протестирована на следующих ОС: MacOS, Windows, Linux. Программу можно скачать по ссылке [7], вариант для CPU по ссылке [8].

2. Алгоритмы *PM* и P^3M. Рассмотрим два основных алгоритма, которые используются для моделирования методом частиц, а именно алгоритмы PM (particle-mesh, частица-сетка) и P^3M (частица-частица, частица-сетка) [1]. Для частиц, гравитационное взаимодействие между которыми описывается законами классической механики, алгоритм PM состоит из следующих шагов:

1) распределение массы частиц по сетке;
2) решение уравнения Пуассона на потенциал и вычисление напряженности в узлах сетки;
3) вычисление с помощью интерполяции напряженности поля в положениях частиц, на основе чего производится вычисление новых положений и скоростей частиц;
4) переход к шагу 1.

Шаг 1 сделаем с помощью линейной интерполяции (алгоритм “cloud in cell”, CIC); существуют и другие методы распределения массы, например можно поместить массу в ближайший узел сетки (подробнее о других методах [1, 5]). Решить уравнение Пуассона можно множеством способов, для простоты возьмем метод разделения переменных [9]. На шаге 3 применяется интерполяция тем же методом, что и на шаге 1, после чего выполняется интегрирование, которое в данной работе осуществляется методом Стермера–Верле [10] в связи с его высокой точностью ($O(h^2)$, где h — шаг сетки) и простотой реализации.

С точки зрения производительности самый сложный шаг метода — решение уравнения Пуассона. Сложность данного шага зависит от числа узлов сетки и от выбранного алгоритма решения уравнения. Например, для метода разделения переменных это $O(N^3 \log(N))$, где N — число шагов сетки по каждому направлению. Таким образом, алгоритм PM может применяться для сколь угодно большого числа частиц.

2023, **24** (1), 37–54. doi 10.26089/NumMet.v24r104

Алгоритм PM хорошо работает для случая, когда силы меняются достаточно гладко, но он не способен учесть должным образом взаимодействие между близкими частицами, силы между которыми могут быть велики и могут быстро меняться. В алгоритме P^3M сила $\boldsymbol{F}_i$, действующая на частицу с индексом i, расщепляется на короткодействующую силу $\boldsymbol{F}_i^s$, которая действует на ограниченном расстоянии r_s, и на длиннодействующую силу $\boldsymbol{F}_i^l$: $\boldsymbol{F}_i = \boldsymbol{F}_i^s + \boldsymbol{F}_i^l$. Сила $\boldsymbol{F}_i^l$ вычисляется по алгоритму PM, сила $\boldsymbol{F}_i^s$ вычисляется путем суммирования сил между всеми парами частиц (i, j), попадающих в шар радиуса r_s с центром в частице i.

Один из способов расщепления силы — расщепить гравитационный потенциал на длиннодействующий и короткодействующий в частотном пространстве [4, 11]:

$$\begin{aligned}
\psi_{\boldsymbol{k}} &= -\frac{4\pi G\rho_{\boldsymbol{k}}}{k^2} = -\frac{4\pi G\rho_{\boldsymbol{k}}}{k^2}\left(e^{-k^2r_s^2} + \left(1 - e^{-k^2r_s^2}\right)\right) = \psi_{\boldsymbol{k}}^l + \psi_{\boldsymbol{k}}^s,\\
\psi_{\boldsymbol{k}}^l &= -\frac{4\pi G\rho_{\boldsymbol{k}}}{k^2}e^{-k^2r_s^2},\\
\psi_{\boldsymbol{k}}^s &= -\frac{4\pi G\rho_{\boldsymbol{k}}}{k^2}\left(1 - e^{-k^2r_s^2}\right),
\end{aligned}$$

где $\psi_{\boldsymbol{k}}, \rho_{\boldsymbol{k}}$ — фурье-образы потенциала и плотности, $\boldsymbol{k} = (k_1, k_2, k_3)$ — вектор индексов, $k = |\boldsymbol{k}|$, G — гравитационная постоянная. В этом случае сила $\boldsymbol{F}_{ij}^s$, действующая на частицу i со стороны частицы j, в обычном пространстве задается формулой

$$\boldsymbol{F}_{ij}^s = -\frac{Gm_im_j(\boldsymbol{r}_i - \boldsymbol{r}_j)}{|\boldsymbol{r}_i - \boldsymbol{r}_j|^3}\left(\operatorname{erfc}\left(\frac{|\boldsymbol{r}_i - \boldsymbol{r}_j|}{2r_s}\right) + \frac{|\boldsymbol{r}_i - \boldsymbol{r}_j|}{r_s\sqrt{\pi}}\exp\left(-\frac{|\boldsymbol{r}_i - \boldsymbol{r}_j|^2}{4r_s^2}\right)\right), \tag{1}$$

в которой m_i, m_j — массы частиц, r_i, r_j — радиус-векторы этих частиц. Такой метод расщепления был выбран из-за более простотой реализации по сравнению с другими методами [1, 5].

Для того чтобы избавиться от близких к нулю величин в знаменателе (1), добавим небольшую положительную величину ε к расстоянию $|\boldsymbol{r}_i - \boldsymbol{r}_j|$.

3. Физическая задача. Описанную методику применим для следующей физической задачи. Рассмотрим вселенную, представляющую собой куб со стороной L, заполненный точечными массами. Количество точек в кубе — N. Массы взаимодействуют в соответствии с законами Ньютона. По всем осям используются периодические краевые условия, таким образом, вселенная в некотором смысле “бесконечная”. Также будем использовать модель Фридмана расширения плоской вселенной с масштабным коэффициентом $a(t)$, который меняется со временем по закону $a(t) = C(t + t_s)^{\frac{2}{3}}$, где C, t_s — константы [12]. Константу t_s удобно задавать так, чтобы выполнялось $a(0) = 1$. Также из уравнений Фридмана следует уравнение, связывающее $\ddot{a}$ с $\bar{\rho}(0)$ — средней плотностью вещества в начальный момент времени:

$$\ddot{a} = -\frac{4}{3}\pi G\frac{\bar{\rho}(0)}{a^2(t)}. \tag{2}$$

При моделировании космологического расширения удобно работать с сопутствующими расстояниями и координатами. Сопутствующее расстояние не меняется со временем при расширении вселенной. Обозначим сопутствующие координаты $\boldsymbol{r'}$, тогда связь с собственными координатами будет выражаться так: $\boldsymbol{r'}(t) = \dfrac{\boldsymbol{r}(t)}{a(t)}$, $\boldsymbol{r'}(0) = \boldsymbol{r}(0)$ [1, 6, 13]. Данный подход подробно описан в [1, 6], приведем основные формулы, которые используются при расчете. Весь расчет будет проводиться в штрихованных переменных, поэтому определим $\boldsymbol{v'} = \dot{\boldsymbol{r}'}$, $\rho'(\boldsymbol{r'}, t) = a^3(t)\rho(\boldsymbol{r}, t)$, $\nabla' = a\nabla$, $\Delta' = a^2\Delta$. Штрихованные дифференциальные операторы работают в терминах сопутствующих координат. Выпишем уравнения Ньютона:

$$\begin{aligned}
\ddot{\boldsymbol{r}} &= \dot{\boldsymbol{v}} = -\nabla\psi,\\
\dot{\boldsymbol{r}} &= \boldsymbol{v},\\
\Delta\psi &= 4\pi G\rho(\boldsymbol{r}, t).
\end{aligned}$$

Подстановкой штрихованных переменных получим: $\ddot{\boldsymbol{r}} = \ddot{\boldsymbol{r}}'a + \dot{a}\dot{\boldsymbol{r}}' + \ddot{a}\boldsymbol{r'} + \dot{a}\dot{\boldsymbol{r}}'$, откуда

$$\dot{\boldsymbol{v}}' + \frac{2\dot{a}}{a}\dot{\boldsymbol{r}}' = -\frac{1}{a}\nabla\psi - \frac{\ddot{a}}{a}\boldsymbol{r'},$$

2023, **24** (1), 37–54. doi 10.26089/NumMet.v24r104

здесь $\dfrac{\dot a}{a}$ не что иное, как параметр (“постоянная”) Хаббла. Заметим, что $\nabla' \dfrac{r'^2}{2} = \boldsymbol{r'}$. Это позволяет ввести обозначение $\psi' = a\psi + \dfrac{a^2\ddot a}{2} r'^2$ и переписать уравнение:

$$\dot{\boldsymbol{v}}' + \frac{2\dot a}{a}\dot{\boldsymbol{r}}' = -\frac{1}{a^3}\nabla'\psi'.$$

Правую часть можно раскрыть, используя определение ψ', изначальное выражение для $\nabla\psi$ из уравнений Ньютона и уравнение Фридмана (2):

$$\Delta'\psi' = 4\pi G\rho a^3 + 3\ddot a a^2 = 4\pi G(\rho'(\boldsymbol{r'}, t) - \bar\rho(0)), \tag{3}$$

именно это уравнение мы будем решать для определения сеточной силы. Заметим, что интеграл по всему пространству от правой части равен нулю. Это условие необходимо для того, чтобы уравнение Пуассона с периодическими краевыми условиями имело решение [14].

Для расчета короткодействующих сил будем использовать стандартную формулу Ньютона. Выпишем ее вид для штрихованных переменных:

$$\boldsymbol{F}_{ij} = -\frac{Gm_im_j}{|\boldsymbol{r}_i - \boldsymbol{r}_j|^3}(\boldsymbol{r}_i - \boldsymbol{r}_j) = -\frac{1}{a^2}\frac{Gm_im_j}{|\boldsymbol{r'}_i - \boldsymbol{r'}_j|^3}(\boldsymbol{r'}_i - \boldsymbol{r'}_j),$$

здесь $\boldsymbol{F}_{ij}$ — сила, действующая на массу m_i со стороны массы m_j. Итоговая расчетная формула выписывается с помощью естественной модификации (1), где появится множитель, зависящий от $a(t)$.

4. Алгоритм для GPU. Ниже будут приведены примеры кода на языке программирования GLSL, который очень похож на язык C, но имеет некоторые особенности. В языке GLSL отсутствуют указатели, присутствуют векторные типы данных, нельзя работать с динамической памятью. Вся память, необходимая для работы, должна быть выделена заранее. Если требуется работать с подмассивами большого массива, то вместо указателей нужно использовать целочисленные индексы. Отличное описание языка GLSL с примерами есть в книге [15]. В листинге 1 приведены входные данные, которые используются во всех программах, а также важные константы и макросы.

Листинг 1. Входные данные для всех программ
Listing 1. Input data for all programs

```
layout(std140, binding=0) uniform Settings {
    vec4 origin; // coordinates of the left bottom edge of the cube
    int particles; // the number of the particles
    int nn;  // grid parameter
    int n;   // log(nn)
    float h; // cell size, h=l/nn
    float l; // cube side length
};

uint xsize = gl_NumWorkGroups.x*gl_WorkGroupSize.x;
uint ysize = gl_NumWorkGroups.y*gl_WorkGroupSize.y;
uint zsize = gl_NumWorkGroups.z*gl_WorkGroupSize.z;

uint globalIndex =
    gl_GlobalInvocationID.z*xsize*ysize+
    gl_GlobalInvocationID.y*xsize+
    gl_GlobalInvocationID.x; // global thread id

// index in 3d-array
#define off(i,k,j) ((i)*nn*nn+(k)*nn+(j))
// index in 3d-array with periodicity
#define poff(i,k,j) (((i+nn)%nn)*nn*nn+((k+nn)%nn)*nn+((j+nn)%nn))
```

2023, **24** (1), 37–54. doi 10.26089/NumMet.v24r104

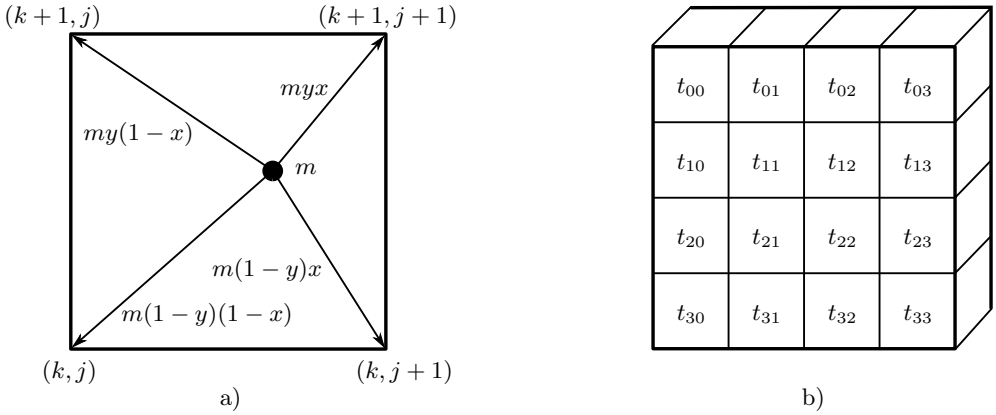

Рис. 1. Распределение массы: a) CIC метод; b) работа потоков с подобластями

Fig. 1. Mass distribution: a) CIC method; b) GPU threads and subdomains

4.1. Распределение массы по узлам. Для распределения масс частиц по узлам сетки будем использовать метод CIC (cloud in cell). О других методах можно прочитать в [1, 5]. Суть метода состоит в том, что масса, находящаяся в кубической ячейке сетки, линейно распределяется по вершинам куба. Для этого рассматриваются барицентрические координаты точки, сумма которых, как известно, равна единице. Барицентрическая координата, соответствующая вершине куба, умножается на массу точки и полученное число добавляется к узлу сетки, соответствующему данному узлу куба. Рис. 1 иллюстрирует данную процедуру для двумерного случая. Здесь (x, y) — координаты точки в квадрате, они принимают значения от 0 до 1. В трехмерном случае добавится координата z и множители z и $1 - z$ перед массой. В листинге 2 приведен наивный параллельный алгоритм. Используемые входные данные приведены в листинге 1, в дополнение к ним: `cur` — индекс точки, с которой работает текущий поток, `Position` — массив точек (w координата содержит массу), `Rho` — плотность в узлах сетки.

Данный код отлично отражает суть алгоритма, но содержит проблему — запись в узел ρ_{ikj} осуществляется без синхронизации из нескольких потоков. Данная проблема возникает из-за того, что разные потоки работают с точками, которые могут находиться в произвольных местах области, в том числе в одной ячейке или в смежных ячейках. Проблему можно решить, сделав кластеризацию точек — область расчета разбивается на подобласти с точками, каждый поток работает только со своей подобластью. На рис. 1 изображено разбиение куба на 4×4 подобласти, с которыми работает группа потоков размера 4×4. В реальной программе используется разбиение 32×32.

В листинге 3 приведен алгоритм построения разбиения. Для каждой подобласти создается односвязный список точек. Голова каждого списка хранится в разделяемой памяти `lists`, связи между узлами

Листинг 2. Распределение массы по методу CIC

Listing 2. Mass distribution using the CIC method

```
float mass = Position[cur].w;
vec3 x = vec3(Position[cur]) - vec3(origin);
ivec3 index = ivec3(floor(x/h));
x = (x-index*h)/h;
for (int i = 0; i < 2; i++) {
  for (int k = 0; k < 2; k++) {
    for (int j = 0; j < 2; j++) {
      Rho[poff(index.z+i,index.y+k,index.x+j)] +=
          4*M_PI*G*mass*
          abs((1.0-i-x.z)*(1.0-k-x.y)*(1.0-j-x.x));
    }
  }
}
```

2023, **24** (1), 37–54. doi 10.26089/NumMet.v24r104

Листинг 3. Построение разбиения
Listing 3. Partitioning

```
layout(std430, binding=7) buffer ListBuffer {
  int list[];
};

uint work_size = (particles + threads - 1) / threads;
uint from = globalIndex * work_size;
uint to = min(particles, from+work_size);

shared int lists[32][32];
float h32 = l/32;

lists[gl_LocalInvocationID.y][gl_LocalInvocationID.x] = -1;
barrier();

for (uint i = from; i < to; i++) {
  vec2 x = vec2(Position[i])-vec2(origin);
  ivec2 index = ivec2(floor(x/h32));

  list[i] = atomicExchange(lists[index.y][index.x], int(i));
}
```

списка содержатся в массиве `list`, сами данные в массиве `Position`. Обходить список для узла (x, y) можно так: `cur = lists[y][x]; while (cur != -1) { pos = Position[cur]; cur = list[cur]; }`. После построения разбиения можно применить алгоритм из листинга 2, параллельно обрабатывая точки из подобластей. Делать это надо с учетом того, что нельзя одновременно работать со смежными областями, т.е. сначала работаем с $t_{00}, t_{02}, t_{20}, t_{22}$, потом с $t_{01}, t_{03}, t_{21}, t_{23}$ и т.д.

4.2. Уравнение Пуассона. Уравнение (3) будем решать методом разделения переменных [9]. Обозначим $f = 4\pi G(\rho - \rho_0)$. Значения функций f и ψ в узле сетки (i, k, j): f_{ikj} и ψ_{ikj}. Используются периодические краевые условия по всем направлениям, поэтому по каждому направлению можно сделать разложение в ряд Фурье по синусам и косинусам. Пусть $N = 2^n$ — число узлов сетки по каждому направлению. Тогда f_{ikj} представимо в виде [9]:

$$f_{ikj} = \sum_{l=0}^{2^{n-1}} \tilde{f}_{ikl} \cos\frac{2j\pi l}{2^n} + \sum_{l=1}^{2^{n-1}-1} \bar{f}_{ikl} \sin\frac{2j\pi l}{2^n}, \quad i, k, j = 0, 1, \ldots, 2^n - 1, \tag{4}$$

$$\begin{aligned} \tilde{f}_{ikj} &= \sum_{l=0}^{2^n-1} f_{ikl} \cos\left(\frac{2j\pi l}{2^n}\right), \quad i, k = 0, 1, \ldots, 2^n - 1, \quad j = 0, 1, \ldots, 2^{n-1}, \\ \bar{f}_{ikj} &= \sum_{l=1}^{2^n-1} f_{ikl} \sin\left(\frac{2j\pi l}{2^n}\right), \quad i, k = 0, 1, \ldots, 2^n - 1, \quad j = 1, 2, \ldots, 2^{n-1} - 1, \end{aligned} \tag{5}$$

где $(\tilde{f}_{ikl}, \bar{f}_{ikl})$ — коэффициенты Фурье для f_{ikj}. Таким образом, имеем разложение f_{ikj} в ряд Фурье по направлению j. Данное разложение осуществляется с помощью алгоритма FFT быстрого преобразования Фурье. Далее можно рассмотреть пару $(\tilde{f}_{ikl}, \bar{f}_{ikl})$ как единое целое и сделать ее разложение по направлению k, а получившийся результат в свою очередь разложить по направлению i. В итоге мы получим набор коэффициентов Фурье $\hat{f}_{ikj}$. Используя коэффициенты Фурье и собственные значения оператора Лапласа сеточной периодической функции, можно найти коэффициенты Фурье для ψ_{ikj}:

$$\hat{\psi}_{ikj} = -\frac{L^2 \hat{f}_{ikj}}{4N^2\left(\sin^2\left(\frac{i\pi}{N}\right) + \sin^2\left(\frac{k\pi}{N}\right) + \sin^2\left(\frac{j\pi}{N}\right)\right)}.$$

2023, **24** (1), 37–54. doi 10.26089/NumMet.v24r104

Листинг 4. Преобразование Фурье по направлению j
Listing 4. FFT along the j-direction

```
i = gl_GlobalInvocationID.x;
k = gl_GlobalInvocationID.y;
for (j = 0; j < nn; j++) {
  W[ioff+j] = Rho[off(i,k,j)];
}
pFFT_1(ooff,ioff,h*slh,nn,n);
for (j = 0; j < nn; j++) {
  Psi[off(i,k,j)] = W[ooff+j];
}
```

Коэффициент $\hat{\psi}_{000}$ можно положить равным любой константе, например нулю. Данное допущение соответствует тому, что решение уравнения Пуассона с периодическими краевыми условиями по всем осям определено с точностью до константы. Используя (4), можно найти ψ_{ikj} по $\hat{\psi}_{ikj}$. На GPU последовательно делается преобразование Фурье сначала по i, потом по k, потом по j. Каждое такое преобразование делается параллельно за счет независимости направлений. Например, сделать преобразование по j необходимо для всех возможных значений (i, k). Для этого можно запустить $i \times k$ GPU-потоков (листинг 4). Здесь `W` — вре́менная рабочая область памяти, `ioff, ooff` — адреса в рабочей памяти, уникальные для каждого потока, `Rho` — правая часть уравнения Пуассона, `Psi` — решение, `pFFT_1` — функция, делающая вычисление по формулам (5) с помощью алгоритма FFT. Полный код программы можно посмотреть по ссылке [7].

Данный подход отлично работает на CPU, на GPU он работает не очень хорошо из-за необходимости обмениваться информацией с буферной вре́менной памятью `W`. Если вместо массива `W` использовать разделяемую (shared) память, то можно получить улучшение производительности. Но в этом случае нужно распараллеливать реализацию преобразования Фурье, чтобы все потоки группы работали с одной и той же разделяемой памятью. Такое распараллеливание осуществляется естественным образом, с помощью распараллеливания внутренних циклов алгоритма. Напомним, что внешний цикл алгоритма имеет $\log(N)$ шагов, а внутренние в среднем по N шагов, при этом результат следующего шага нетривиальным образом по индексам массива зависит от результата предыдущего, поэтому распараллеливать надо именно внутренние циклы. Из-за громоздкости кода приведем только пример подготовки данных запуска преобразования Фурье в листинге 5. Программу из листинга 5 надо запускать на $N \times N$ группах — по группе на узел сетки. В качестве размера группы используется только одно измерение, при этом оно обязательно должно делить N.

Листинг 5. Преобразование Фурье по направлению j (улучшенный вариант)
Listing 5. Enhanced FFT along the j-direction

```
shared float S[512]; // input at offset 0, output at offset nn
size = gl_WorkGroupSize.x; // use 1-d group of size (x,1,1)
thread_id = gl_LocalInvocationID.x;
work = nn/size; // work size of current thread
id = thread_id*work; // data offset for thread

i = gl_GlobalGroupID.x;
j = gl_GlobalGroupID.y;
for (i = id; i < id+work; i++) {
  S[i] = Rho[off(i,k,j)];
}
pFFT_1(nn,0,h*slh,nn,n); // parallel version of FFT
for (i = id; i < id+work; i++) {
  Psi[off(i,k,j)] = S[nn+i];
}
```

2023, **24** (1), 37–54. doi 10.26089/NumMet.v24r104

Надо дождаться выполнения преобразования по какой-либо оси перед запуском преобразования по следующей оси. Если используется одна вычислительная группа, то синхронизацию различных этапов можно сделать с помощью функции `barrier()` в самой GLSL-программе. Программа из листинга 5 существенным образом зависит от исполнения на нескольких вычислительных группах, поэтому синхронизацию в данном случае надо обеспечивать внешними средствами. Синхронизацию можно сделать из программы на языке C с помощью вызова `glMemoryBarrier` для OpenGL или с помощью `vkCmdPipelineBarrier` для Vulkan.

4.3. Напряженность. Для аппроксимации напряженности $E = -\nabla\psi$ используется схема второго порядка точности:

$$E_x(x,y,z) = \frac{\psi(x+h,y,z) - \psi(x-h,y,z)}{2h}, \tag{6}$$

для направлений y, z используются аналогичные выражения. В листинге 6 приведена программа для GPU.

4.4. Короткодействующие силы. Для расчета короткодействующих сил

$$\boldsymbol{F}_i^s = \sum_{j:|\boldsymbol{r}_i - \boldsymbol{r}_j| < r_s} \boldsymbol{F}_{ij}^s,$$

действующих на частицу i, нужно рассмотреть все пары частиц, расстояние между которыми меньше r_s. Разобьем область на кубы со стороной длины не меньше r_s. Для каждой такой ячейки будем поддерживать массив точек, входящих в ячейку. При расчете на видеокарте нет возможности динамически выделять память, вся память должна быть выделена заранее, поэтому на каждую ячейку области надо зарезервировать память, например на 1024 индекса точки. Если мы разбили область на 64^3 ячейки, то понадобится $64^3 \cdot 1024 \cdot 4$ байт памяти для хранения индексов.

Для параллельного заполнения массивов можно воспользоваться разбиением, построенным в разделе 4.1. Алгоритм из листинга 7 использует списки, построенные алгоритмом из листинга 3. Массив `cell_data` содержит индексы точек для ячейки. Данные для ячейки с номером `cell_id` начинаются с

Листинг 6. Расчет напряженности
Listing 6. Gravitational field strength

```
uint i = gl_GlobalInvocationID.x;
uint k = gl_GlobalInvocationID.y;
for (j = 0; j < nn; j++) {
  E[off(i,k,j)].x = -(Psi[poff(i,k,j+1)]-Psi[poff(i,k,j-1)])/2.0/h;
  E[off(i,k,j)].y = -(Psi[poff(i,k+1,j)]-Psi[poff(i,k-1,j)])/2.0/h;
  E[off(i,k,j)].z = -(Psi[poff(i+1,k,j)]-Psi[poff(i-1,k,j)])/2.0/h;
}
```

Листинг 7. Заполнение массивов точек для каждой ячейки
Listing 7. Filling arrays of points for each cell

```
int cur = lists[gl_LocalInvocationID.y][gl_LocalInvocationID.x];
while (cur != -1) {
  vec3 x = vec3(Position[cur]) - vec3(origin);
  ivec3 index = ivec3(floor(x/h));
  int cell_id = off(index.z,index.y,index.x);
  uint n = cell_data[cell_id*cell_size];
  if (n < max_per_cell) {
    cell_data[cell_id*cell_size+n+1] = cur;
    cell_data[cell_id*cell_size] = n+1;
  }

  cur = list[cur];
}
```

2023, **24** (1), 37–54. doi 10.26089/NumMet.v24r104

индекса `cell_id*cell_size`, где `cell_size` — константа, например 1024. В первой ячейке каждого подмассива содержится число точек, содержащихся в ячейке `cell_id`.

После построения массивов остается для каждой ячейки обойти 3^3 соседних ячеек и для всех точек, входящих в эти ячейки, решить стандартную задачу N тел. Данную задачу можно эффективно решить на GPU с помощью алгоритма, предложенного в [16]. Суть алгоритма состоит в том, что точки обрабатываются параллельно, а точки, которые обходятся во внутреннем цикле алгоритма, кешируются в разделяемой памяти и далее внутренний цикл работает именно с данными из разделяемой памяти, а не из исходного массива точек. В [17] показано, что такой трюк с кешированием позволяет увеличить производительность решения задачи N тел в 8 раз.

4.5. Расчет положений. Новые координаты и скорости частиц рассчитываются с помощью метода Стермера–Верле [10]. Для частицы i имеем:

$$\boldsymbol{r}_i^{n+1} = \boldsymbol{r}_i^n + \tau \boldsymbol{v}_i^n + \frac{1}{2}\tau^2 \dot{\boldsymbol{v}}_i^n,$$
$$\dot{\boldsymbol{v}}_i^{n+1} = \frac{1}{m_i}\left(\boldsymbol{F}_i^l + \boldsymbol{F}_i^s\right) - \frac{2\tau \dot{a}}{a}\boldsymbol{v}_i^n,$$
$$\boldsymbol{v}_i^{n+1} = \boldsymbol{v}_i^n + \frac{1}{2}\tau(\dot{\boldsymbol{v}}_i^{n+1} + \dot{\boldsymbol{v}}_i^n),$$

здесь τ — шаг по времени, $\boldsymbol{F}_i^l$ вычисляется с помощью напряженности (6), значение которой в точке нахождения частицы определяется с помощью метода CIC (раздел 4.1), с единственным отличием — теперь надо не значение распределить по узлам сетки, а собрать значение в точке из значений в узлах сетки.

Заметим, что если производятся вычисления с одновременным выводом на экран, то может быть удобно вычислять новые положения сразу в вершинном шейдере и сразу пропускать координаты дальше по графическому конвейеру. Поточечную параллельность в этом случае за нас сделает GPU. Чтобы продемонстрировать идею, в листинге 8 приведен фрагмент шейдера. На вход в качестве вершин передается целочисленный массив `idx`. Шейдер запускается параллельно для каждого элемента массива `idx`. В этом массиве хранятся номера частиц, то есть `idx[i] = i` (эта инициализация была сделана заранее). Отметим, что если не нужна совместимость с OpenGL, то можно вообще использовать вершинный шейдер без входа, а вместо целочисленного массива использовать встроенную переменную `gl_VertexIndex`, которая будет содержать номер вершины. Сами координаты, скорости, ускорения, напряженность "прикрепляются" к шейдеру в виде буферных объектов, как и в случае расчетов в вычислительном шейдере. Следует отметить, что в вершинном шейдере можно делать более сложные вычисления. Еще одно из преимуществ такого подхода — можно запускать расчеты на старом оборудовании, которое не поддерживает API Vulkan или OpenGL старше 4.3. В [7] есть пример кода решения задачи N тел с помощью стандартного алгоритма в вершинном шейдере.

Листинг 8. Вершинный шейдер для расчета новых положений
Listing 8. Calculation of new positions in vertex shader

```
layout (location = 1) in int idx; // index of current particle: 0,1,2,...
layout (location = 0) out vec4 color;

int main() {
  float mass = Position[idx].w;
  gl_Position = MVP * vec4(vec3(Position[idx]), 1); // vertex for visualization
  color = vec4(0,1,1,1); // input for fragment shader (particle color)
  vec4 A = vec4(0); vec4 r = vec4(0);
  r = Position[idx] + tau*Velocity[idx] + 0.5 * tau*tau*Accel[idx];
  A = ... // CIC interpolation + Short Force
  Velocity[idx] = Velocity[idx] + 0.5 * tau * (vec4(vec3(A),0)+Accel[idx]);
  Accel[idx] = vec4(vec3(A), 0); Position[idx] = vec4(vec3(r), mass);
}
```

2023, **24** (1), 37–54. doi 10.26089/NumMet.v24r104

Таблица 1. Ошибка решения уравнения Пуассона

Table 1. Poisson equation solving error

Grid n^3, n	CPU		GPU	
	Relative error	Error decrease	Relative error	Error decrease
32	9.501655e-03	—	9.501261e-03	—
64	2.401749e-03	3.96	2.401266e-03	3.96
128	6.022543e-04	3.99	6.019362e-04	3.99
256	1.508816e-04	3.99	1.505103e-04	4.00
512	3.790651e-05	3.98	3.770563e-05	3.99

4.6. О расчете на CPU. Для CPU была написана аналогичная программа, с естественными отличиями. Так, для поддержания массива частиц в каждой ячейке сетки использовались динамические массивы `std::vector<int>`, которые заново заполнялись на каждом шаге алгоритма в один поток. Все циклы по точкам и по ячейкам распараллеливались естественным образом с помощью OpenMP. В целом, программа для CPU гораздо проще, так как можно использовать динамическую память, а в ручной синхронизации с помощью механизмов вида `atomicExchange` или `barrier` нет необходимости. Программа для CPU выложена на GitHub [8].

5. Результаты.

5.1. Корректность. Корректность программы, решающей уравнение Пуассона, проверялась на тестовой функции $\psi = \sin^3 x + \cos^5 y - \sin z$. Правая часть уравнения Пуассона для этой функции: $f = 1/16(-12\sin x + 36\sin 3x - 10\cos y - 45\cos 3y - 25\cos 5y + 16\sin z)$. Задача решалась в кубе со стороной 2π, использовались периодические краевые условия по всем осям. В табл. 1 приведены относительная погрешность решения и падение погрешности при измельчении сетки. Приведены результаты для версий решателя для CPU и для GPU.

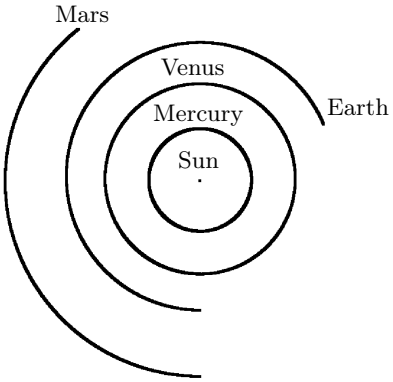

Рис. 2. Траектории тел планетарной системы

Fig. 2. Trajectories of bodies of planetary system

Чтобы убедиться, что орбиты получаются правдоподобными, использовалась планетарная система из пяти тел солнечной системы. Тест хорош тем, что любые ошибки в программе приводили к разлету тел или к образованию странных траекторий, отличных от эллиптических, также этот тест удобен для отладки программы. Параметры теста можно посмотреть в табл. 2. За единицу массы принята масса Земли, за единицу расстояния — расстояние от Земли до Солнца, скорость Земли принята за единицу. В этой нормировке гравитационная постоянная $G = 2.96\text{e}{-6}$. В начальный момент времени тела выстроены по оси Y, скорости ортогональны оси Y (задана X-компонента скорости). Солнце помещено в начало системы координат. Расстояния от Солнца для всех планет и скорости взяты из Википедии с применением соответствующей нормировки. Траектории тел изображены на рис. 2. При счете продолжительное время качественно картина не меняется — тела продолжают вращаться вокруг

Таблица 2. Параметры планетарной системы из пяти тел

Table 2. The parameters of a planetary system composed of five bodies

Body name	Y-coordinate	X-component of velocity	Mass
Sun	0	0	333333
Mercury	0.39	1.58	0.038
Venus	0.72	1.17	0.82
Earth	1	1	1
Mars	1.51	0.8	0.1

2023, **24** (1), 37–54. doi 10.26089/NumMet.v24r104

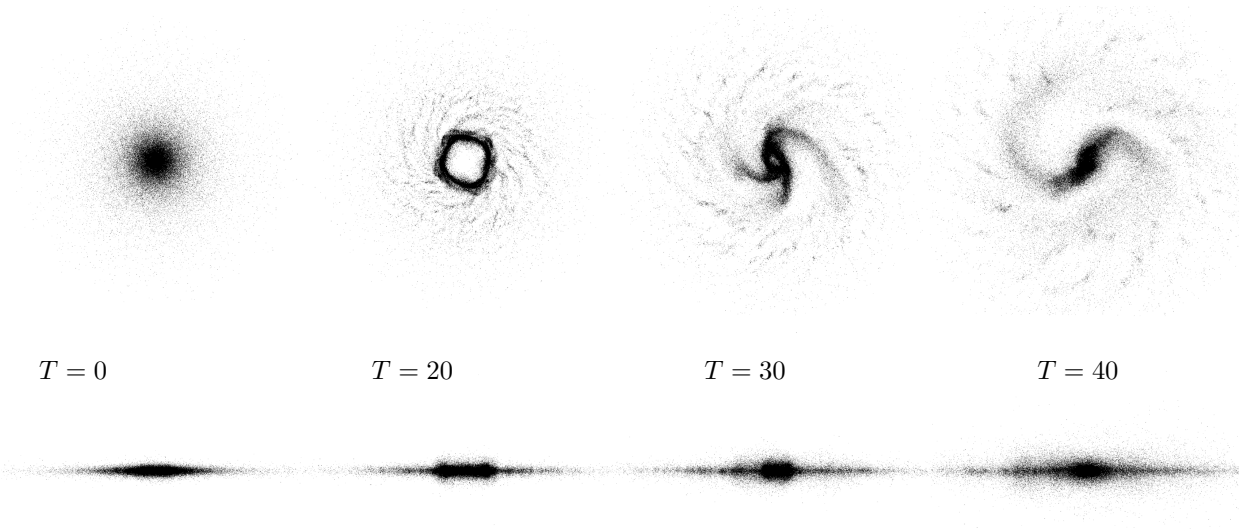

Рис. 3. Эволюция галактического диска

Fig. 3. Evolution of a galactic disk

Солнца. В этом и следующем тестах масштабный коэффициент брался равным тождественно единице, краевые условия на потенциал брались периодическими.

Еще одним хорошим тестом является задача о моделировании динамики бесстолкновительной системы галактического диска. Эта задача позволяет продемонстрировать устойчивость системы, где галактика не должна разлетаться или коллапсировать. В качестве начальных параметров для расчета возьмем диск Серсика, сгенерированный программой MAGI [18]. В качестве настроек программы MAGI возьмем Listing 3 из [18], количество тел — 100000. Состояние галактического диска в двух проекциях в различные моменты времени T приведено на рис. 3.

5.2. Эволюция вещества, распределенного по кубу. На рис. 4 представлен расчет с двумя параметрами расширения — 2 и 2.5. Над каждым изображением обозначено время счета. Можно видеть, что вещество, равномерно размазанное по кубу, с течением времени начинает образовывать структуры.

На рис. 5 сравниваются расчеты, выполненные по алгоритмам P^3M и PM с одними и теми же начальными данными и параметрами. На рисунке слева границы у структур более четкие, почти отсутствуют филаменты между галактическими скоплениями. Похожие результаты есть в работе [4].

5.3. Производительность. Расчет выполнялся на CPU Apple M1 и AMD Ryzen 3700X, а также на GPU Apple M1 и NVIDIA RTX 3060. Для расчета использовались два компьютера: 1) ноутбук Macbook Air с 16 ГБ памяти (общая оперативная и видеопамять) и с 8-ядерным GPU; 2) PC с AMD Ryzen 3700X, видеокартой Gygabyte NVIDIA RTX 3060 12Gb, 32Gb RAM. Ноутбук Macbook Air использует пассивное охлаждение, при перегреве он может начать пропускать CPU циклы, чтобы снизить температуру. Тесты, имеющиеся в Интернете, показывают, что пропуск циклов начинается после 10 минут непрерывной нагрузки, чего более чем достаточно для тестов, рассмотренных ниже.

В табл. 3 приведена производительность GPU в Tflops для FP32 по данным, сообщаемым производителем. Для сравнения в таблицу добавлены видеокарты AMD и NVIDIA RTX 3080. Следует учитывать, что это пиковая производительность, достичь которую в реальных вычислениях сложно.

AMD и Apple не указывают производительность своих CPU, поэтому были выполнены замеры в тесте Linpack [19]. Параметры теста: размер задачи — 28672, размеры блоков — 64, 128, 256, 512, число процессов — 8 для M1 и 16 для Ryzen 3700X. Тест собирался с оптимизированной версией библиотеки blas. На Apple это Accelerate [20], на AMD — ATLAS [21]. В табл. 4 приведено максимальное число, которое удалось получить.

Из данных о производительности можно сделать вывод, что даже встроенной графики M1 достаточно для того, чтобы получить ускорение в 10 и более раз по сравнению с CPU версией программы. Производительность GPU AMD сравнима с производительностью GPU NVIDIA. Написание универсальной программы для всех видов GPU оправдано.

2023, **24** (1), 37–54. doi 10.26089/NumMet.v24r104

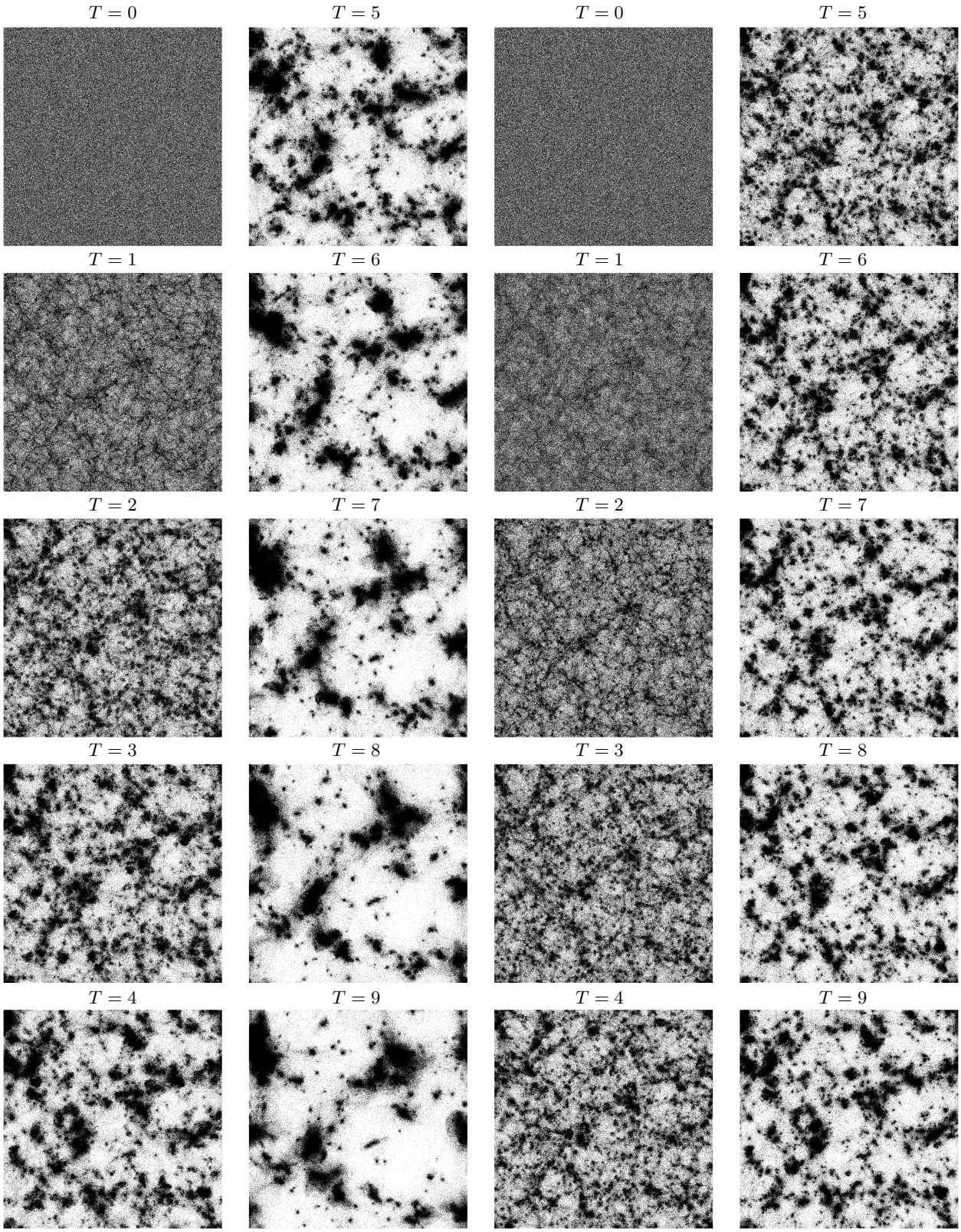

Рис. 4. Эволюция распределения вещества по кубу: a) расчет с параметром расширения 2; b) расчет с параметром расширения 2.5

Fig. 4. Evolution of matter distribution across a cube: a) expansion factor 2, b) expansion factor 2.5

2023, **24** (1), 37–54. doi 10.26089/NumMet.v24r104

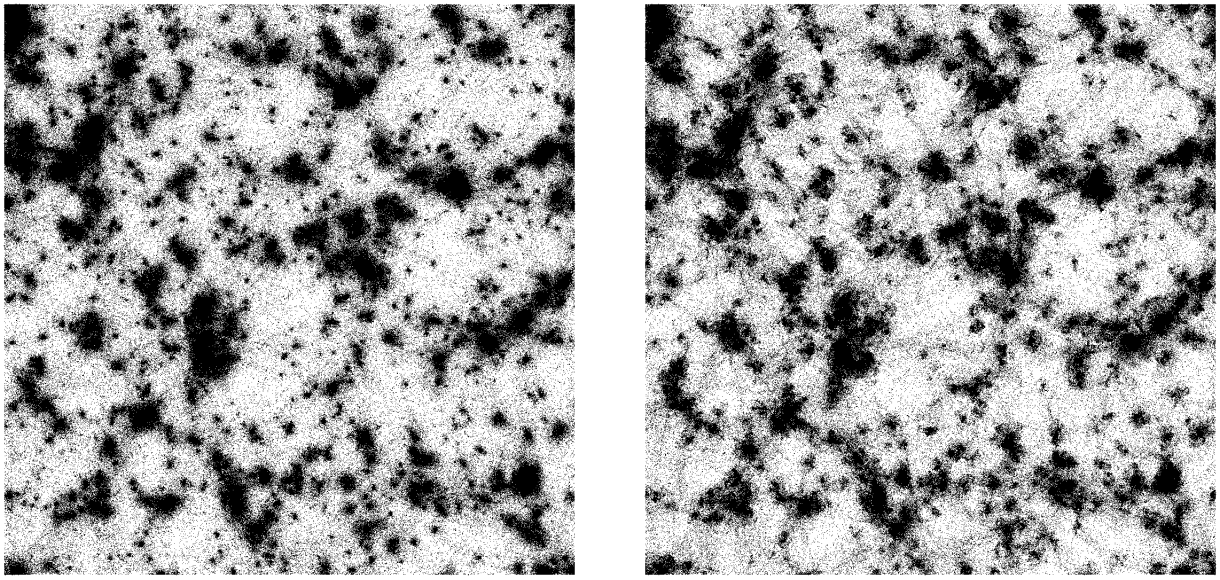

Рис. 5. Сравнение расчетов с помощью алгоритмов P^3M (слева) и PM (справа)

Fig. 5. Comparison of calculations using P^3M (left) and PM (right) algorithms

Таблица 3. Сравнение производительности различных GPU по техническим спецификациям

Table 3. Comparison of performance of different GPUs according to technical specifications

GPU	Tflops, FP32	Relative performance
Apple M1 [22]	2.6	1
AMD Radeon RX 6600 XT [23]	10.60	4
NVIDIA RTX 3060 [24]	12.7	4.88
AMD Radeon RX 6800 XT [25]	20.74	7.97
NVIDIA RTX 3080 [26]	29.77	11.45

Таблица 4. Сравнение производительности CPU в тесте Linpack [19]

Table 4. Comparison of CPU performance in the Linpack test [19]

CPU	Gflops, FP64	Relative performance
AMD Ryzen 3700X	116	1
Apple M1	190	1.63

Тестирование производительности осуществлялось на задаче из раздела 5.2 с параметром расширения 2. В табл. 5–7 приведено время работы в миллисекундах алгоритмов P^3M и PM, а также время работы каждого этапа по отношению к времени работы на процессоре M1. Количество тел — 1000000. Расчеты на видеокартах делались одновременно с выводом изображения на экран. Еще одно из преимуществ вычислений на видеокартах — можно сразу дешево выводить результаты счета и в режиме реального времени видеть траектории системы. На не очень мелких сетках (64^3 и меньше) это возможно даже на встроенном GPU.

Процессор Ryzen проигрывает M1 не только на синтетическом тесте (табл. 4), но и при реальных расчетах. Видеокарта RTX 3060 производительней M1 в 4.8 раз по спецификациям. На практике получилась разница от 3.5 до 4 раз.

В таблицах видна аномалия времени работы этапа вычисления сеточной силы $\boldsymbol{F}_i^l$ на видеокарте M1. По всей видимости, это связано с особенностями извлечения длительности конкретных этапов работы графического конвейера. Не на всех видеокартах можно это сделать достаточно точно.

2023, **24** (1), 37–54. doi 10.26089/NumMet.v24r104

Таблица 5. Абсолютное и относительное время работы алгоритма P^3M для числа частиц $N = 10^6$, сетка для потенциала: 32^3, сетка для локальных сил: 64^3. Абсолютное время в миллисекундах, относительное по отношению к M1 CPU

Table 5. The absolute and relative time of P^3M algorithm for $N = 10^6$ particles, grid resolution: 32^3, resolution of grid for local forces: 64^3. The absolute time in milliseconds. Base unit for the relative time is M1 CPU time

Stage	Apple M1 (CPU)		Apple M1 (GPU)		Ryzen 3700X		NVIDIA RTX 3060	
	t, ms	t, unit	t, ms	t, unit	t, ms	t, unit	t,ms	t, unit
ρ	9.63	1	5.36	1.80	6.01	1.60	1.36	7.08
ψ	1.70	1	1.21	1.41	1.26	1.35	0.25	6.80
$\boldsymbol{F}_i^l$	0.83	1	0.01	83	0.73	1.14	0.04	20.75
$\boldsymbol{F}_i^s$	159.02	1	103.06	1.54	123.91	1.28	28.84	5.51
$\boldsymbol{r}_i$	68.3	1	7.84	8.71	153.84	0.44	1.00	68.3
Total	239.66	1	117.48	2.04	285.75	0.84	31.49	7.61

Таблица 6. Абсолютное и относительное время работы алгоритма P^3M для числа частиц $N = 10^6$, сетка для потенциала: 64^3, сетка для локальных сил: 64^3. Абсолютное время в миллисекундах, относительное по отношению к M1 CPU

Table 6. The absolute and relative time of P^3M algorithm for $N = 10^6$ particles, grid resolution: 64^3, resolution of grid for local forces: 64^3. The absolute time in milliseconds. Base unit for the relative time is M1 CPU time

Stage	Apple M1 (CPU)		Apple M1 (GPU)		Ryzen 3700X		NVIDIA RTX 3060	
	t, ms	t, unit	t, ms	t, unit	t, ms	t, unit	t,ms	t, unit
ρ	9.45	1	11.61	0.81	6.55	1.44	2.26	4.18
ψ	8.87	1	5.69	1.56	6.90	1.27	0.86	10.31
$\boldsymbol{F}_i^l$	5.99	1	0.01	599	5.32	1.13	0.14	42.79
$\boldsymbol{F}_i^s$	313.93	1	103.09	3.04	209.32	1.50	28.87	10.87
$\boldsymbol{r}_i$	82.74	1	8.38	9.87	212.8	0.38	1.41	57.46
Total	420.98	1	128.78	3.27	440.89	0.95	33.54	12.55

Таблица 7. Абсолютное и относительное время работы работы алгоритма PM для числа частиц $N = 10^6$, сетка для потенциала: 128^3. Абсолютное время в миллисекундах, относительное по отношению к M1 CPU

Table 7. The absolute and relative time of PM algorithm for $N = 10^6$ particles, grid resolution — 128^3. The absolute time in milliseconds. Base unit for the relative time is M1 CPU time

Stage	Apple M1 (CPU)		Apple M1 (GPU)		Ryzen 3700X		NVIDIA RTX 3060	
	t, ms	t, unit	t, ms	t, unit	t, ms	t, unit	t,ms	t, unit
ρ	14.65	1	45.08	0.32	11.52	1.27	6.58	2.22
ψ	68.83	1	40.37	1.7	62.50	1.10	5.50	12.51
$\boldsymbol{F}_i^l$	47.11	1	0.01	4711	42.15	1.12	3.71	12.70
$\boldsymbol{r}_i$	133.63	1	12.37	10.8	242.97	0.55	1.78	75.07
Total	264.22	1	97.81	1.98	359.14	0.74	17.57	15.03

В табл. 8 и 9 приведена зависимость алгоритмов PM и P^3M от числа частиц N и от разрешения сетки n^3. Также для сравнения приведены результаты для выполнения похожей задачи (по разрешению сеток и числу частиц) из работы [3]. Запуск делался на видеокарте NVIDIA RTX 3060. Для алгоритма P^3M близкодействующие силы рассчитываются на сетке 64^3. В работе [3] расчет выполнялся на кластере из 8 узлов по 8 ядер каждый узел. Например, для сетки 64^3 и числа частиц 3×10^7 в [3] получилось 30 мс. В данной работе на видеокарте получилось 98.51 мс, что очень неплохо, с учетом того, что расчет делался на обычной домашней технике. Для алгоритма P^3M в [3] для 4×10^6 частиц и на сетке 64^3 получилось 231 мс, в данной работе — 339 мс.

Алгоритм PM работает очень производительно, можно было бы дальше увеличивать число частиц, но в этом случае столкнемся с ограничением видеопамяти.

2023, **24** (1), 37–54. doi 10.26089/NumMet.v24r104

Таблица 8. Производительность алгоритма *PM* для различного числа частиц и различных сеток. Время в миллисекундах. $*$ — для сравнения результаты из [3]

Table 8. Performance of the *PM* algorithm for various number of particles and various resolutions. Time in milliseconds. $*$ — for comparison results of [3]

N, $\times 10^6$	$n = 32$		$n = 64$		$n = 128$
2	5.68		9.44		36.44
4	10.02		15.75		44.91
8	18.37		29.47		60.21
30	62.43	17.3*	98.51	30.5*	143.11
40	80.02	22.5*	126.70	36.0*	180.65
100	193.58		310.70		410.53

Таблица 9. Производительность алгоритма P^3M для различного числа частиц и различных сеток. Время в миллисекундах. $*$ — для сравнения результаты из [3]

Table 9. Performance of the P^3M algorithm for various number of particles and various resolutions. Time in milliseconds. $*$ — for comparison results of [3]

N, $\times 10^6$	$n = 32$		$n = 64$		$n = 128$
2	90.99	58*	98.77	70.3*	136.20
4	331.23	220.0*	339.77	231.1*	375.96
8	1141.58		1172.94		1184.04

6. Выводы. Современные игровые видеокарты среднего уровня, а также встроенные видеокарты подходят для ускорения численных расчетов. Предложенная реализация алгоритмов эффективна. Время работы на видеокарте сравнимо с временем работы на кластере из нескольких узлов. Полученные результаты соответствуют результатам других авторов. Написанную программу можно использовать в качестве основы для расчета методом частиц других физических задач. Некоторые из подпрограмм, например подпрограммы для решения уравнения Пуассона, можно использовать для решения более сложных задач гидродинамики. Одно из преимуществ описанного подхода — возможность использовать персональный компьютер для быстрого проведения различных численных экспериментов с различными параметрами, а также возможность наблюдать изменения, происходящие в ходе эксперимента, без необходимости складывать результаты в промежуточные файлы и без использования отдельных программ визуализации.

Не все этапы описанной реализации алгоритма работают хорошо на GPU. Например, стоит подумать над тем, как улучшить распределение масс по сетке. Также следует уделить внимание оптимизации параметров GPU, таких как разбиение вычислений на группы и число потоков в группе, подбору размера используемой разделяемой памяти в группах потоков и т.д. Оптимизация параметров может дать хороший эффект в улучшении производительности.

Список литературы

1. *Hockney R.W., Eastwood J.W.* Computer simulation using particles. New York: McGraw-Hill, 1981.
2. *Harnois-Déraps J., Pen U.-L., Iliev I.T., et al.* High-performance P^{3}M N-body code: CUBEP^{3}M // Monthly Notices of the Royal Astronomical Society. 2013. **436**, N 1. 540–559. doi 10.1093/mnras/stt1591.
3. *Kyziropoulos P.E., Filelis-Papadopoulos C.K., Gravvanis G.A.* Parallel N-body simulation based on the PM and P3M methods using multigrid schemes in conjunction with generic approximate sparse inverses // Math. Probl. Eng. 2015. **2015**, Article ID 450980. doi 10.1155/2015/450980.
4. *Bagla J.S.* TreePM: a code for cosmological N-body simulations // J. Astrophys. Astr. 2002. **23**. 185–196. doi 10.1007/BF02702282.
5. *Xu K., Jing Y.* An accurate P^{3}M algorithm for gravitational lensing studies in simulations // Astrophys. J. 2021. **915**, N 2. doi 10.3847/1538-4357/ac0249.
6. *Bryan G.L., Norman M.L., O'Shea B.W., et al.* ENZO: an adaptive mesh refinement code for astrophysics // Astrophys. J. Suppl. 2014. **211**, N 2. Article ID 19. doi 10.1088/0067-0049/211/2/19.

2023, **24** (1), 37–54. doi 10.26089/NumMet.v24r104

7. Исходный код *PM*, P^3M, версия для GPU. https://github.com/resetius/graphtoys (Дата обращения: 28 декабря 2022).
8. Исходный код *PM*, P^3M, версия для CPU. https://github.com/resetius/fdm (Дата обращения: 28 декабря 2022).
9. *Самарский А.А., Николаев Е.С.* Методы решения сеточных уравнений. М.: Наука, 1978.
10. *Verlet L.* Computer 'experiments' on classical fluids. I. Thermodynamical properties of Lennard–Jones molecules // Physical Review. 1967. **159**, N 1. 98–103.
11. *Ewald P.P.* Die Berechnung optischer und elektrostatischer Gitterpotentiale // Annalen der Physik. 1921. **369**, N 3. 253–287. doi 10.1002/andp.19213690304.
12. *Горбунов Д.С., Рубаков В.А.* Введение в теорию ранней Вселенной: Теория горячего Большого взрыва. М.: URSS, 2008.
13. *Byrd G.G., Chernin A.D., Valtonen M.J.* Cosmology: foundations and frontiers. M.: URSS, 2007.
14. *Тихонов А.Н., Самарский А.А.* Уравнения математической физики. М.: Наука, 2004.
15. *Вольф Д.* OpenGL 4. Язык шейдеров. Книга рецептов. М.: ДМК Пресс, 2015.
16. *Nyland L., Harris M., Prins J.* Fast N-body simulation with CUDA // GPU Gems 3. https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-31-fast-n-body-simulation-cuda (Дата обращения: 28 декабря 2022).
17. *Боресков А.В., Харламов А.А* Основы работы с технологией CUDA. М.: ДМК Пресс, 2019.
18. *Miki Y, Umemura M.* MAGI: many-component galaxy initializer // Monthly Notices of the Royal Astronomical Society. 2018. **475**, N 2. 2269–2281. doi 10.1093/mnras/stx3327.
19. *Petitet A., Whaley R.C., Dongarra J., Cleary A.* HPL — a portable implementation of the high-performance Linpack benchmark for distributed-memory computers. https://netlib.org/benchmark/hpl/ (Дата обращения: 21 июля 2022).
20. Apple Accelerate. https://developer.apple.com/documentation/accelerate (Дата обращения: 28 декабря 2022).
21. Atlas Library. http://math-atlas.sourceforge.net (Дата обращения: 28 декабря 2022).
22. Apple M1 Performance in Teraflops. https://www.ixbt.com/news/2021/02/18/soc-apple-m1x-geforce-gtx-1070.html (Дата обращения: 28 декабря 2022).
23. Radeon RX6600XT Specifications. https://www.techpowerup.com/gpu-specs/radeon-rx-6600-xt.c3774 (Дата обращения: 28 декабря 2022).
24. GeForce RTX3060 Specifications. https://www.techpowerup.com/gpu-specs/geforce-rtx-3060.c3682 (Дата обращения: 28 декабря 2022).
25. Radeon RX6800XT Specifications. https://www.techpowerup.com/gpu-specs/radeon-rx-6800-xt.c3694 (Дата обращения: 28 декабря 2022).
26. GeForce RTX3080 Specifications. https://www.techpowerup.com/gpu-specs/geforce-rtx-3080.c3621 (Дата обращения: 28 декабря 2022).

Поступила в редакцию
7 октября 2022 г.

Принята к публикации
21 декабря 2022 г.

Информация об авторе

Алексей Владимирович Озерицкий — к.ф.-м.н., ведущий разработчик программного обеспечения, учитель; 1) Яндекс, Москва, ул. Льва Толстого, 16, 119021, Москва, Российская Федерация; 2) Университетская гимназия (школа-интернат) МГУ имени М. В. Ломоносова, Ломоносовский пр-кт, 27, корп. 7, 119192, Москва, Российская Федерация.

References

1. R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles* (McGraw-Hill, New York, 1981).
2. J. Harnois-Déraps, U.-L. Pen, I. T. Iliev, et al., "High-Performance P^{3}M N-Body Code: CUBEP^{3}M," Mon. Not. R. Astron. Soc. **436** (1), 540–559 (2013). doi 10.1093/mnras/stt1591.
3. P. E. Kyziropoulos, C. K. Filelis-Papadopoulos, and G. A. Gravvanis, "Parallel N-Body Simulation Based on the PM and P3M Methods Using Multigrid Schemes in Conjunction with Generic Approximate Sparse Inverses," Math. Probl. Eng. **2015**, Article ID 450980 (2015). doi 10.1155/2015/450980.

2023, **24** (1), 37–54. doi 10.26089/NumMet.v24r104

4. J. S. Bagla, "TreePM: A Code for Cosmological N-Body Simulations," J. Astrophys. Astr. **23**, 185–196 (2002). doi 10.1007/BF02702282.
5. K. Xu and Y. Jing, "An Accurate P^{3}M Algorithm for Gravitational Lensing Studies in Simulations," Astrophys. J. **915** (2) (2021). doi 10.3847/1538-4357/ac0249.
6. G. L. Bryan, M. L. Norman, B. W. O'Shea, et al., "ENZO: An Adaptive Mesh Refinement Code for Astrophysics," Astrophys. J. Suppl. **211** (2), Article ID 19 (2014). doi 10.1088/0067-0049/211/2/19.
7. The Source Code of *PM*, P^3M, GPU Version. https://github.com/resetius/graphtoys. Cited December 26, 2022.
8. The Source Code of *PM*, P^3M, CPU Version. https://github.com/resetius/fdm. Cited December 26, 2022.
9. A. A. Samarskii and E. S. Nikolaev, *Numerical Methods for Grid Equations* (Nauka, Moscow, 1978; Birkhäuser, Basel, 1989).
10. L. Verlet, "Computer 'Experiments' on Classical Fluids. I. Thermodynamical Properties of Lennard–Jones Molecules," Phys. Rev. **159** (1), 98–103 (1967).
11. P. P. Ewald, "Die Berechnung Optischer und Elektrostatischer Gitterpotentiale," Annalen der Physik **369** (3), 253–287 (1921). doi 10.1002/andp.19213690304.
12. D. S. Gorbunov and V. A. Rubakov, *Introduction to the Theory of the Early Universe: Hot Big Bang Theory* (URSS, Moscow, 2008; World Scientific, Singapore, 2011).
13. G. G. Byrd, A. D. Chernin, and M. J. Valtonen, *Cosmology: Foundations and Frontiers* (URSS, Moscow, 2007).
14. A. N. Tikhonov and A. A. Samarskii, *Equations of Mathematical Physics* (Nauka, Moscow, 2004; Dover Publications, New York, 2013).
15. D. Wolf, *OpenGL 4. Shading Language Cookbook* (Packt Publishing, Birmingham, 2018).
16. L. Nyland, M. Harris, and J. Prins, "Fast N-Body Simulation with CUDA," in *GPU Gems 3* https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-31-fast-n-body-simulation-cuda. Cited December 28, 2022.
17. A. V. Boreskov and A. A. Harlamov, *Basics with CUDA Technology* (DMK Press, Moscow, 2019) [in Russian].
18. Y. Miki and M. Umemura, "MAGI: Many-Component Galaxy Initializer," Mon. Not. R. Astron. Soc. **475** (2), 2269–2281 (2018). doi 10.1093/mnras/stx3327.
19. A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary, "HPL — A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers," https://netlib.org/benchmark/hpl/. Cited July 21, 2022.
20. Apple Accelerate. https://developer.apple.com/documentation/accelerate. Cited December 27, 2022.
21. Atlas Library. http://math-atlas.sourceforge.net. Cited December 27, 2022.
22. Apple M1 Performance in Teraflops. https://www.ixbt.com/news/2021/02/18/soc-apple-m1x-geforce-gtx-1070.html [in Russian]. Cited December 27, 2022.
23. Radeon RX6600XT Specifications. https://www.techpowerup.com/gpu-specs/radeon-rx-6600-xt.c3774. Cited December 27, 2022.
24. GeForce RTX3060 Specifications. https://www.techpowerup.com/gpu-specs/geforce-rtx-3060.c3682. Cited December 27, 2022.
25. Radeon RX6800XT Specifications. https://www.techpowerup.com/gpu-specs/radeon-rx-6800-xt.c3694. Cited December 27, 2022.
26. GeForce RTX3080 Specifications. https://www.techpowerup.com/gpu-specs/geforce-rtx-3080.c3621. Cited December 27, 2022.

Received
October 7, 2022

Accepted for publication
December 21, 2022

Information about the author

Aleksei V. Ozeritskii' — Ph.D., principal software engineer, teacher; 1) Yandex, Moscow, ulitsa Lva Tolstogo, 16, 119021, Moscow, Russia; 2) Gymnasium of Moscow State University, Lomonosovskiy prospekt, 27, building 7, 119192, Moscow, Russia.