Санкт–Петербургский государственный университет

ГЛАЗЫРИН Антон Георгиевич

Выпускная квалификационная работа

Оптимизация реализации метода моделирования «дыма Шрёдингера» за счет применения технологий вычислений на сопроцессорах GPGPU

Уровень образования: бакалавриат
Направление 02.03.02 «Фундаментальная информатика и информационные технологии»
Основная образовательная программа СВ.5003.2016 «Программирование и информационные технологии»
Профиль «Автоматизация научных исследований»

Научный руководитель:
профессор, кафедра компьютерного моделирования и многопроцессорных систем, д.т.н. Дегтярев Александр Борисович

Рецензент:
эксперт, общество с ограниченной ответственностью «Т-Системс Рус», Типикин Юрий Александрович

Санкт-Петербург
2020 г.

Содержание

Введение

В современном мире люди все чаще и чаще прибегают к помощи компьютерной симуляции в совершенно различных задачах, так как она является более дешевой и удобной альтернативой реальным экспериментам. Однако для такого подхода необходима, во-первых, корректная математическая модель, описывающая воспроизводимое явление, и во-вторых, что не менее важно, качественная программная реализация этой модели.

Как одно из важнеших направлений компьютерной симуляции можно рассматривать имитирование потока частиц [1]. В общем смысле это достаточно универсальный подход к моделированию различных процессов, например, поведение заряженных частиц в электромагнитных полях [2], внутреннее состояние жидкостей [3], или формирование ударной волны при вызрыве [4], которые имеют в основе одну и ту же идею: большое количество очень маленьких объектов перемещаются под влиянием заданных внешних ограничений и условий.

Отдельно можно выделить моделирование потока несмижаемомой жидкости [5]. Под этим термином понимается такая сплошная среда, плотность которой сохраняется при изменение давления. Несмотря на то, что это математическая абстракция, ее особые физические свойства часто используются при моделировании реальных течений жидкостей в различных условиях, при которых можно сделать соответствующие приближения. Это обеспечивает актуальность исследования темы во многих сферах, например: предсказание погоды (моделирование океана и атмосферы), применения в химии и тд. [6].

Одним из новейших методов симуляции этого процесса является несжимаемый поток Шрёдингера [7]. Особенность подхода заключается в описании поведения частиц с помощью волновых функций комплексных переменных. В данной работе рассматривается практическая реализация этого метода моделирования.

Постановка задачи

Авторы оригинальной статьи для демонстрации корректности и возможностей своего подхода подготовили демо-видео, содержащие различные симуляции. В качестве инструмента они выбрали пакет Houdini[1]. Это программное обеспечение, используемое для 3D-анимации и моделирования (рис. 1)[2].

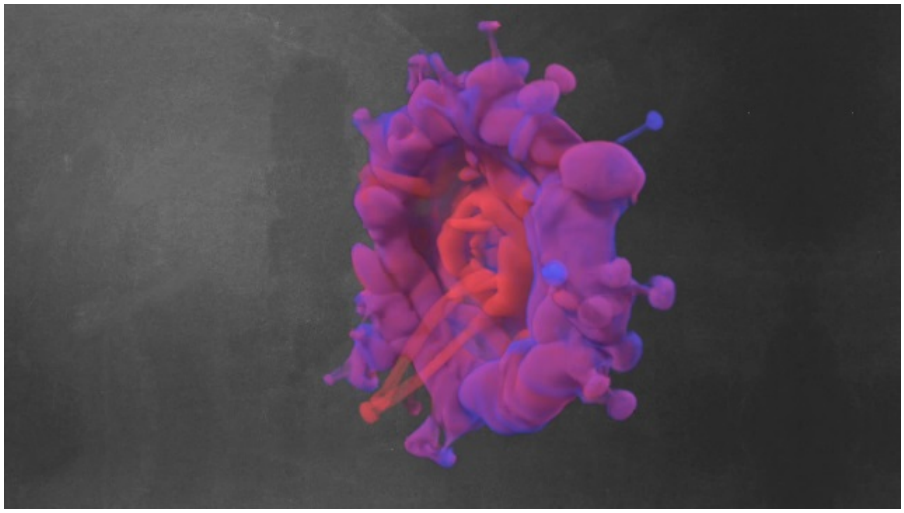


Рис. 1: Пример визуализации метода в Houdini.

И хотя этот проект отлично служит для демонстрации теоретических результатов, у него есть ряд существенных недостатков:

- пакет предназначет для рендера 3D-сцен и не поддерживает симуляцию в реальном времени;
- из-за специфики работы с пакетом код не получится использовать для других целей.

Существует также решение, написанное на matlab[1]. Это пакет для решения задач технических вычислений, который также имеет возможности строить различные графики (рис. 2). И хотя такая реализация уже больше подходит на роль готового продукта, она все еще имеет схожие минусы:

[1]http://page.math.tu-berlin.de/ chern/projects/SchrodingersSmoke/

[2]Источник: https://www.youtube.com/watch?v=5C9BLAXCe1I&feature=emb_logo

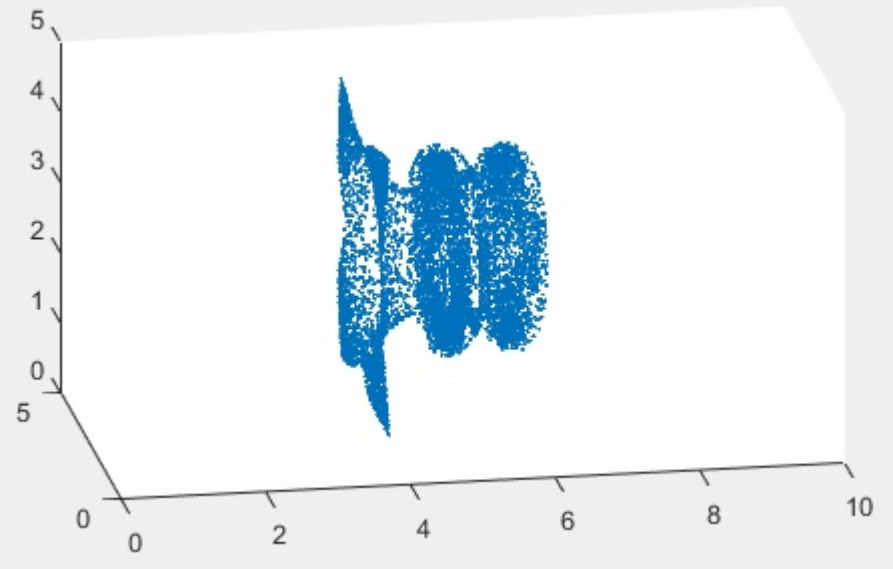


Рис. 2: Пример визуализации метода в Matlab.

- недостаточно высокая производительность для симуляции в реальном времени;
- язык matlab нельзя использовать вне пакета;
- скудные возможности для визуализации.

Таким образом, оригинальные реализации не подходят для практического применения при решении каких-либо реальных задач и являются по сути только примером того, как можно использовать теоретический метод. Эта проблема является ключевой, и ее устранение является основой решаемой в данной работе задачи.

Для прикладного использования рассматриваемого метода моделирования необходимо создать достаточно универсальный программный продукт, который бы удовлетворял следующим требованиям:

- для написания используется один из самых часто встречающихся языков программирования;
- имеет достаточно высокую производительность для проведения симуляиции в режиме реального времени;

- имеет возможность достаточно простой интеграции в какие-либо существующие системы.

Решение обозначенной проблемы и удовлетворение вышеперечисленных условий позволить применять метод несжимаемого потока Шрёдингера в более широком спектре задач.

Обзор литературы

В данной работе рассматривается симуляция потока частиц. Теоритические особенности такого моделирования можно изучить в книге [1], а примеры использования - в статьях [2–4].

Чтобы получить представление о предметной области - несжимаемой жидкости - можно обратиться к [5, 6], где содержатся необходимые формулы и способы применения этой физической модели в реальных задачах. Конкретно метод моделирования, используемый в данной работе, подробно описан в статье [7];

Особенности подхода к проведению всех необходимых вычислений можно найти в [8–11]. В этих статьях показывается, как можно эффективно использовать численные методы для решения сложных уравнений.

Основное задачей данной работы является написание приложения, реализуещего метод моделирования. Статьи [12, 13] рассказывают об использовании технологий GPGPU для оптимизации кода, а [14, 15] подробнее описывают работу с технологией CUDA. Общий подход к разработке основывается на [16].

Глава 1. Описание метода моделирования

1.1 Теоретическая часть

Особенностью метода несжимаемого потока Шрёдингера является то, что состояние жидкости представлено в виде трехмерной комплексной волновой функции двух переменных $\psi = (\psi_1, \psi_2)$. При этом классические плотность ρ и скорость $\nu = (\nu_1, \nu_2, \nu_3)$ можно вычислить как:

$$\rho = |\psi^2| = (\psi, \psi)_R \tag{1}$$

$$\rho\nu_\alpha = \hbar(\frac{\partial\psi}{\partial x_a}, i\psi)_R, \alpha = 1, 2, 3 \tag{2}$$

где:

$$(\phi, \psi)_R = Re(\overline{\phi_1}\psi_1 + \overline{\phi_2}\psi_2),$$

$\hbar$ - постоянная планка

Данная функция изменяется во времени под влиянием уравнения Шредингера:

$$i\hbar\psi = -\frac{\hbar^2}{2}\Delta\psi + p\psi \tag{3}$$

с учетом ограничений:

$$(\Delta\psi, i\psi) = 0 \tag{4a}$$

$$|\psi^2| = 1 \tag{4b}$$

где: p - потенциал, являющийся Лагранжевым множителем для ограничения расходимости [8].

Эти ограничения равносильны классическим уравнениям, характеризующим несжимаемую жидкость: $div(v) = 0$ и $\rho = 1$ [6]. Таким образом, рассматриваемый метод описывает тот же процесс, что и более стандартные методы, однако не использует непосредственное представление скоростей и завихрений. Благодаря этому изменению повышается точность и целостность структуры моделируемого потока.

Для проведения вычислений будем считать пространство и время дискретными, таким образом будет симулироватся дискретная система частиц [9]. Рассматриваемый метод моделирования является сеточным, то есть пространтсво представляет собой регулярную сетку, в узлах которой определяются значения функций [10]. Каждая точка содержит информацию о среде в компактной области вокруг себя. Такая дискретизация позволяет удобно производить необходимые вычисления.

Для численного решения имеющихся нелинейных дифференциальных уравнений в частных производных используется метод быстрого преобразования Фурье на дисперсионном шаге (split step Fourier method) [11]. Само преобразование производится в трехмерном пространстве. Формула в общем виде:

$$f(\omega) = \frac{1}{(2\pi)^{n/2}} \int_{R^n} f(x) e^{-ix*\omega} dx$$

где ω и x - векторы пространства R^n.

1.2 Ход алгоритма

Процесс симуляции процесса является итерационным. На каждом шаге выполняется ряд действий, преобразующих характеристики потока во всех точках, и затем под влиянием этих изменений обновляются положения частиц. Рассмотрим каждый этап более детально:

1. Поток Шрёдингера

   (a) Решение уравнения Шрёдингера (3) - эволюция во времени волновой функции ψ. Перед рассчетами необходимо провести прямое преобразование Фурье, после - обратное;

   (b) Нормализация - предотвращение роста абсолютных значений функции;

   (c) Проекция давления - накладывание основных ограничений (4a), (4b) равенства дивиргенции скорости нулю и неизменности плотности с помощью решения уравнения Пуассона, калибровочное преобразование волновой функции.

2. Обновление положений частиц

   (a) Рассчет скоростей в узлах решетки - используя уравнения (1), (2) выделения классического представления скорости из функции ψ;

   (b) Вычисление новых координат частиц - решение дифференциальных уравнений при помощи метода Рунге-Кутты 4 порядка.

Определив основные этапы алгоритма и производимые вычисления можно выбрать подходящие технологии для написания программы, которая бы эффективно реализовывала данный метод.

Глава 2. Обзор технологий и инструментов разработки

2.1 Выбор основного языка программирования

Первым шагом при проектирование системы является выбор подходящего языка программирования. Так как главная цель, к которой нужно стремиться, это максимальная универсальность итогового решения, выбор в основном стоит между самыми популярными современными языками, которые подходят для создания масштабной системы: C++, C# и Java.

C++ довольно низкоуровневый язык, что затрудняет разработку, и кроме того не является кросс-платформенным, так что можно исключить его из вариантов. Так как итоговое приложение должно будет некоторым образом интегрироваться в другие проекты, нужно рассматреть возможные варианты его использования. Например, часто используемым инструментом для визуализации, создания полноценных симуляций, а также игр, где часто применяются различные эффекты, основанные на частицах, является игровой движок Unity. Работа с ним подразумевает написание кода на языке C#, поэтому в конечном счете выбор пал именно на него.

2.2 Оптимизация при помощи технологий GPGPU

Моделирование потока частиц - крайне ресурсоемкий процесс. Необходимо рассчитывать поведение каждого отдельного объекта, каких могут быть тысячи или даже десятки тысяч. Кроме того, основные операции, связанные с процессом симуляции, - это применения численных методов на дискретной сетке и поэлементные перемножения трехмерных массивов, что тоже требует огромных вычислительных мощностей. А так как качество модели напрямую зависит от таких параметров, как общее число частиц, шаг интегрирования по времени и разрешение сетки, представляющей пространство, можно уверенно сказать, что каких-либо приемлимых результатов невозможно добиться полагаясь исключительно на современные процессоры.

Относительно недавно появилось решение этой проблемы - проводить такие вычисления на видеокарте [12]. При помощи технологий GPGPU

можно свободно перемещать данные между центральным и графическими процессорами, производя на видеокарте математические вычисления, не связанные с графикой. Так как графические ускорители буквально созданы для параллельного проведения огромного количества относительно простых операций, использование этой технологии играет незаменимую роль в оптимизации процесса моделирования.

Когда речь заходит о выборе конкретного инструмента для реализации, выбор по сути стоит между CUDA и OpenCL - двумя неоспоримыми лидерами в данной области [13]. Разница между ними со всеми вытакающими плюсами и минусами заключается в том, что первая технология является проприетарной, тогда как вторая - открытый стандарт. Кроме того, у CUDA есть еще один довольно значительный минус - отсутствие кроссплатформенности, так как эта технология принадлежит Nvidia и работает только на видеокартах от этого производителя. Несмотря на это было решено использовать в качестве инструмента взаимодействия с ГПУ именно CUDA - она обладает более высокой производительностью и стабильностью по сравнению с аналогами, а также имеет множетсво бонусов, таких как качественная поддержка и документация.

2.3 Выбор библиотеки для работы с GPGPU

Программировать взаимодействие с видеокартой с нуля не имеет смысла, так как существует множество готовых решений, предлагающих обширный функционал и высокую производительность. Так как такие библиотеки могут сильно отличаться как с точки зрения их использования, так и внутреннего устройства, необходимо тщательно проанализировать возможные варианты. Единственное жесткое требование - это поддержка языка C#, в остальном важно то, насколько библиотека подходит для данного проекта.

В качестве рассматриваемых альтернатив были выбраны CUDAfy[3],

[3]Источник: https://github.com/rapiddev/CUDAfy.NET

managedCuda[4], ArrayFire[5] и Cloo[6]. Для их анализа были выделены критерии, которые сильнее всего влияют на удобство и эффективность использования. Некоторые из них стоит выделить отдельно из-за их важности, или для приведения некоторых пояснений.

FFT3D - вычисление быстрого преобразования фурье для трехмерного массива используется часто, и так как это довольно сложный алгоритм, большим плюсом является встроенная реализация. OpenCL - хотя основной технологией была выбрана CUDA, если библиотека позволяет легко переключиться на OpenCL практически не меняя кода, это является преимуществом. CUDA kernels - возможность написания отдельных нативных ядер CUDA это плюс, так как их можно будет использовать вне приложения. Таблица сравнения приведена ниже.

Таблица 1: Сравнение библиотек для работы с GPGPU

Feature	CUDAfy	managedCuda	ArrayFire	Cloo
OpenCL	✓	×	✓	✓
CUDA	✓	✓	✓	×
CUDA kernels	✓	✓	×	×
Flexible data structures	✓	✓	×	×
FFT3D	×	✓	✓	✓
Unix support	×	✓	✓	✓
Last update	10.04.2020	26.04.2020	29.05.2020	13.03.2019

По результатам анализа было решено выбрать библиотеку managedCuda, так как она предоставляет наибольший спектр возможностей.

[4]Источник: https://github.com/kunzmi/managedCuda
[5]Источник: https://arrayfire.com/
[6]Источник: https://github.com/clSharp/Cloo

Глава 3. Разработка архитектуры

3.1 Внутреннее устройство

Разботка приложения велась на языке C#[7]. Для данной системы хорошо подходит модульный принцип построения приложения. Как было показано выше, процесс симуляции хорошо разбивается на две независимые части - преобразования состояния, связанные с ограничениями, и обновление положений частиц. Обособим эти этапы в отдельные модули, таким образом логика работы не будет лишний раз пересекаться. Также можно выделить инициализацию функций CUDA, так как они хранятся отдельно от основного проекта и должы загружаться уже после начала работы программы. Кроме главных классов, содержащих данные, также нужны вспомогательные. Они либо предоставляют функционал, требующий особой инициализации, либо позволяют компактно хранить параметры при помощи специальных структур, либо просто отделяют логику для упрощения работы с системой. Общий вид архитектуры показан на рис. 3.

Основные классы отвечают за хранение данных - в классе ISF находится функция ψ, а в классе Particles - координаты частиц, а также содержат методы для их обновления. Единственная промежуточная переменная, которая не инкапсулирована, это скорость. Такое решение было принято для того, чтобы можно было свободно доваблять какие-либо дополнительные внешние воздействия.

Для того, чтобы получить более развернутое представление о стуктуре приложения, можно обратиться к реализованному примеру симуляци. Visual Studio позволяет сгенерировать граф, содержащий такую информацию, как вызовы определенных функций, передачу данных и тп (рис. 4).

[7]Документация: https://docs.microsoft.com/ru-ru/dotnet/csharp/

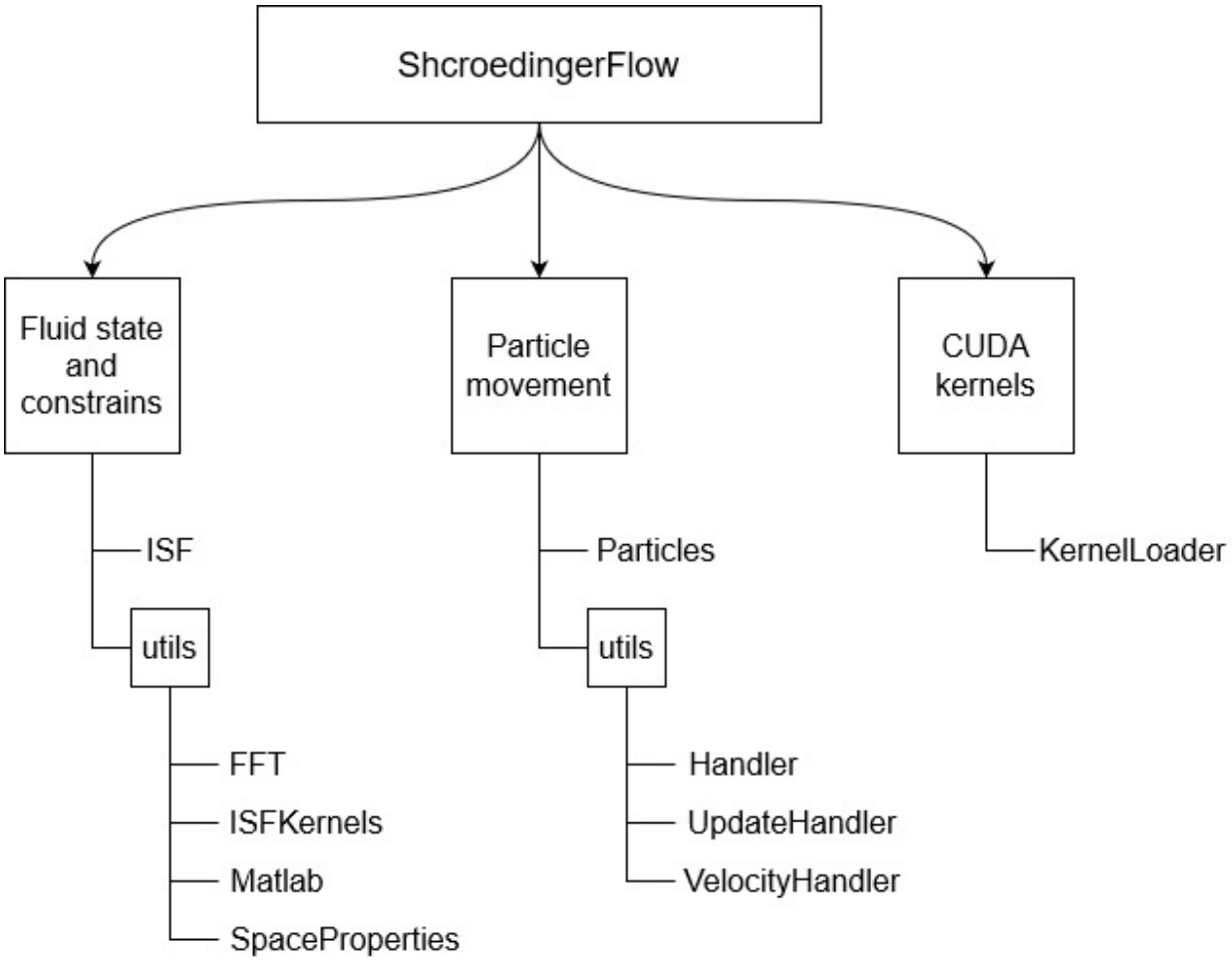


Рис. 3: Архитектура приложения.

3.2 Внешние интерфейсы

Для удобного взаимодействия с приложениям со стороны необходимо добавить интерфейсы. Каждый класс предоставляет необходимые методы для инициализации и работы, которые принимают только необходимые параметры, и использование которых интуитивно понятно. Таким образом при интеграции приложения в другие системы отсутствует необходимость знания его полной структуры, достаточно лишь общего понимания процесса симуляции. Всего существует пять методов, полностью ответственных за всю работу:

- Инициализация:

    1. состояния - ISF.init

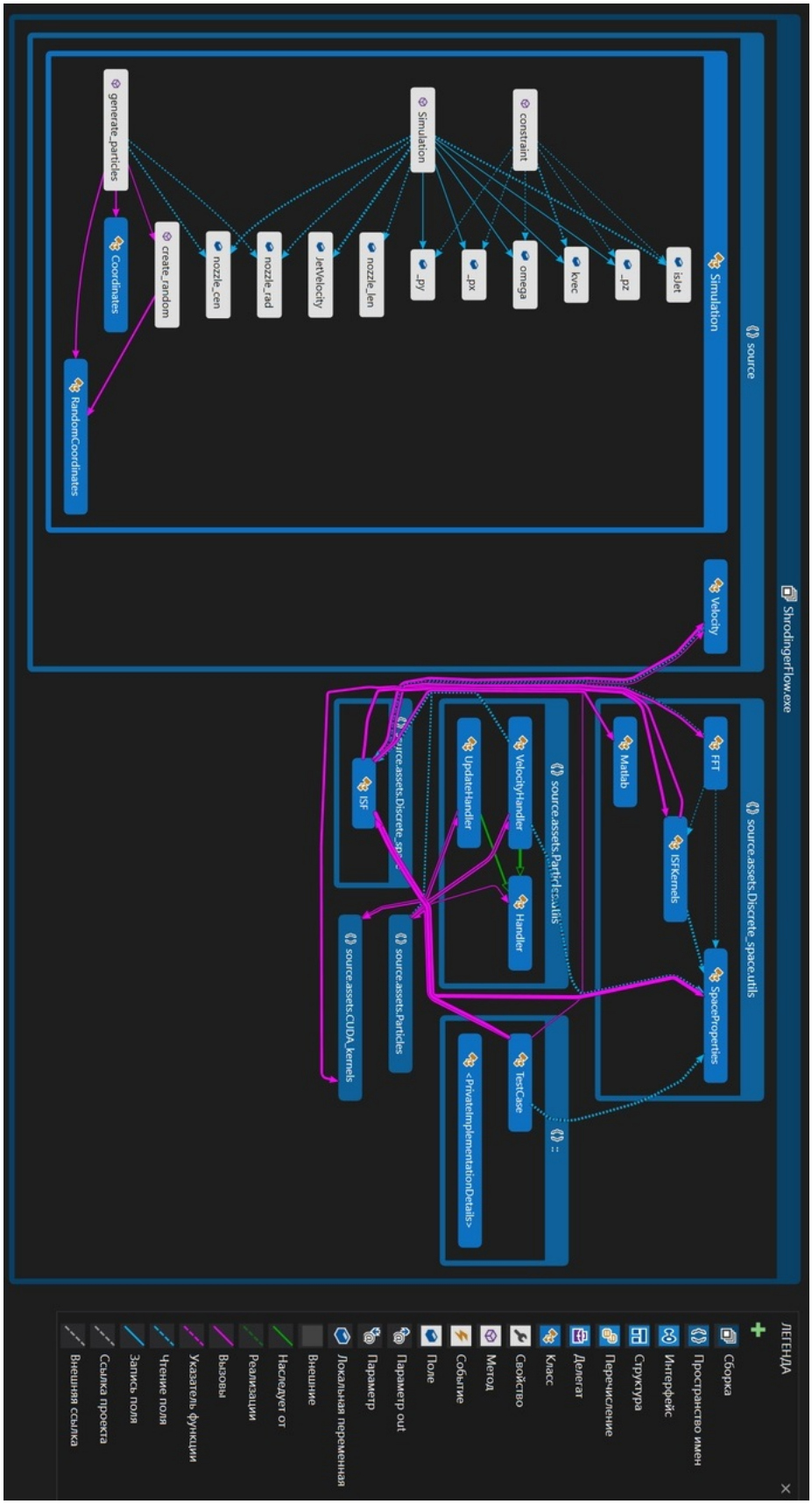


Рис. 4: Граф отношений в реализации.

2. частиц - Particles.init

- Шаг симуляции:

1. обновление состояния - ISF.update_space
2. расчет скоростей - ISF.update_velocities
3. обновление положения частиц - Particles.calculate_movement

3.3 Оптимизация работы с GPGPU

Основные вычисления написаны как функции CUDA[8]. Данная технология позволяет выполнять огромное количество действий параллельно. Это отлично подходит для работы с большими массивами, если нужно выполнить ряд одинаковых действий с каждым элементом массива. При вызове функций(ядер) CUDA можно указать нужное количество блоков и потоков, которое будет вызвано для проведения вычислений (рис. 5) [14]. Линейный индекс элемента исходного массива можно вычислить внутри ядра при помощи индеков потока(threadIdx) и блока(blockIdx), и размерности блока(blockDim) и решетки(gridDim).

Использование технологий GPGPU накладывает определенные ограничения на обычный поток работы приложения. Во-первых, для вычислений на видеокарте необходимо скопировать туда нужные данные. Этот процесс весьма трудозатратен, и чем реже он вызывается, тем лучше. Во-вторых, особенностью используемой библиотеки managedCuda является то, что она поддерживает только одномерные массивы, а для все рассчеты проводятся на трехмерных. И в-третьих, CUDA не поддерживает операции с комплексными числами, которые часто применяются при рассчетах.

Для решении первой проблемы было решено организовать поток данных таким образом, чтобы при промежуточных вычислениях все результаты всегда хранились на видеокарте. При этом методы получают и отдают лишь ссылки на эти массивы. Необходимость копировать данные обратно на ЦПУ возникает только в конце итерации, когда нужно получить новые координаты частиц для дальнейшей работы с ними.

[8]Документация: https://docs.nvidia.com/cuda/

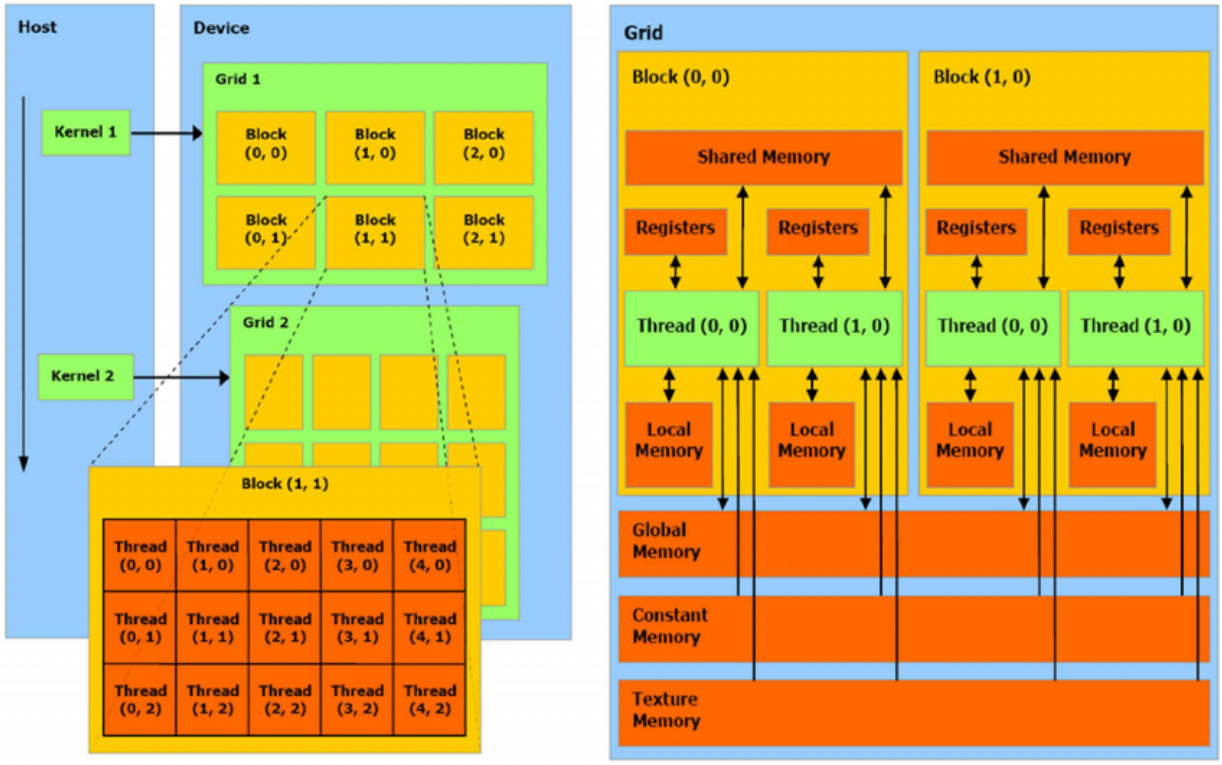


Рис. 5: Архитектура CUDA.

Трехмерные массивы не нужно вручную вытягивать в одномерные - метод копирования данных на ГПУ библиотеки managedCuda делает это сам. Однако возникает другая проблема - индексация. Необходимо определить правила сопоставления элемента трехмерного массива и его одномерного аналога. Рассмотрим массив a размерности X x Y x Z. Элемент с индексами i, j, k обозначим a[i, j, k], div - целочисленное деление, mod - операция взятия остатка. При копировании его на ГПУ получим массив b размерности X * Y * Z, обозначим это произведение как size. Тогда справедливы следующие соотношения:

1. Простая индексация: a[i, j, k] = b[i * Y * Z + j * Z + k]
2. Циклические сдвиги влево:
   - По оси x: a[i + 1, j, k] = b[(i + Z * Y) mod size]
   - По оси y: a[i, j + 1, k] = b[i - ((i div Z) mod Y) * Z + (((i + Z) div Z) mod Y) * Z]
   - По оси z: a[i, j, k + 1] = b[(i div Z) * Z + (i + 1) mod Z]

3. Циклические сдвиги вправо:

   - По оси x: a[i - 1, j, k] = b[(i div Z) * Z + ((i - 1 + size) mod size) mod Z]
   - По оси y: a[i, j - 1, k] = b[i - ((i div Z) mod Y) * Z + ((((i - Z + size) mod size) div Z) mod Y) * Z]
   - По оси z: a[i, j, k - 1] = b[(i - Y * Z + size) mod size]

Используя данные формулы можно проводить все необходимые вычисления на ГПУ.

Хотя CUDA и не поддерживает комплексные числа отдельный тип, можно заменить их структурами, содержащими два числа с плавающей точкой. Однако в таком случае придется вручную реализовать математические операции. Так как работа с комплексным числом по сути является работой с двумя действительными, дополнительных проблем не возникает, потому что встроенный тип cuFloatComplex библиотеки корректно копируется в массив типа float2, который понимает CUDA.

Стоит отметить, что общая производительность сильно зависит от того, какое количество нитей и блоков задействуется при вызове функций [15]. В данной ситуации больше - не значит лучше, ведь лишние потоки будут простаивать, а организация большого их числа - дополнительная нагрузка. Из-за этого было решено добавить динамическое определение оптимальной размерности решетки и блоков - чем больше данных приходит, тем больше создается потоков.

3.4 Визуализация

Одним из основных преимуществ приложения должна быть возможность интегрироваться в практически любую систему. Исходя их этого результат его работы - просто набор координат. Для визуализации необходимо воспользоваться сторонними средствами. Также таким образом можно оценить, насколько удобно работать с проектом.

В качестве инструмента был выбран движок Unity[9]. Он имеет встро-

[9]Документация: https://docs.unity3d.com/ru/current/Manual/UnityManual.html

еную систему отображения частиц, так что достаточно раз в кадр передавать в нее обновленные координаты. Схема процесса визуализации приведена ниже (рис. 6)

Рис. 6: Архитектура проекта Unity.

Глава 4. Ход работы

Основная разработка происходила при помощи технологий Virtual Network Computing(VNC). Удаленный сервер работал под управлением операционной системой Ubunty и имел в распоряжении видеокарту Nvidia. Кроме того, использовались две машины с Windows, в основном для тестирования, одна также с ГПУ от Nvidia, другая - с ГПУ от компании AMD. Такая организация процесса разработки играет важную роль в создании приложения, так как одной из поставленных задач является достижение как можно большей кросс-платформенности.

Отдельно стоит отметить решение проблем с конфигурацией проекта. Так как инструменты для разработки, сборки и компиляции на Windows и Ubuntu могут довольно сильно отличаться, необходимо было настроить взаимодействие со всеми соответствующими компонентами на обеих системах.

В качестве методологии разработки была выбрана RUP [16]. Эта модель хорошо подходит для решения поставленной задачи из-за специфики оптимизации при помощи GPGPU: для проведения некоторых вычислений на ГПУ необходимо реализовать отдельную функцию, которая будет отвечать только за этот шаг. После внесения изменений, нужно провести тестирование для проверки корректности работы программы, причем время от времени оно должно быть более масштабным и охватывать все используемые машины. Для общего доступа к коду использовалась система контроля версий git[10], в качестве платформы был выбран github. Схема процесса разработки приведена ниже (рис. 7).

[10]Документация: https://git-scm.com/doc

Рис. 7: Схема рабочего процесса.

Глава 5. Описание результатов

5.1 Тестирование производительности

В результате данной работы были создано приложение, позволяющее проводить симуляцию согласно рассматриваемому методу моделирования. Из особенностей реализации стоит отметить поведение алгоритма на границах моделируемого объема. Частицы, вышедшие за рамки, будут зависать на месте: они не исчезнут, но будут игнорироваться в дальнейших расчетах.

После завершения проекта необходимо протестировать корректность его работы. Так как в процессе моделирования нигде не испольуются генераторы случайных чисел, то результаты его работы при нескольких запусках на одних и тех же данных не будут отличаться. Тестирование было пройдено успешно - при одинаковых начальных значениях параметров и одинаковом наборе частиц, после того, как алгоритм отработал 100 итераций были получены те же (с точностью до погрешности рассчетов) значения координат, что и у оригинального решения на matlab.

Далее следует тестирование производительности. Единственными параметрами, которые влияют на время выполнения, являются разрешение вычислительной сетки пространства в моделируемом объеме(количество узлов для каждого измерения) и число частиц. Будем изменять эти параметры и засекать, за сколько программа выполнит определенное количество итераций. Сравнивать будем оригинальное решение на matlab, запускаться которое будет при помощи пакета Octave, и проект, созданный в рамках данной работы. В обоих случаях моделируется 500 итераций одного и того же процесса. Характеристики окружения, на котором проводилось тестирование:

- CPU: Intel® Core™ i7-3820
- GPU: Nvidia GEFORCE® GTX 1080
- GPGPU library: managedCuda, version 10
- OS: Windows 10 & Ubuntu 18.04

Результаты представлены в таблицах(формат времени - hh:mm:ss.ms):

Таблица 2: Тест производительности оригинальной matlab реализации

Volume Resolution	Particle Number	Execution time	
		Windows	Ubuntu
32 x 16 x 16	10000	00:00:15.01	00:00:12.21
	50000	00:00:48.34	00:00:41.00
	250000	00:05:11.34	00:03:10.51
	500000	00:10:50.17	00:04:27.34
	750000	00:15:50.72	00:06:36.93
	1000000	00:21:40.69	00:09:02.84
64 x 32 x 32	10000	00:00:55.19	00:00:32.55
	50000	00:01:31.83	00:00:50.00
	250000	00:06:32.84	00:02:33.06
	500000	00:11:55.35	00:04:55.91
	750000	00:17:04.48	00:07:18.76
	1000000	00:20:10.20	00:10:03.07
128 x 64 x 64	10000	00:6:50.20	00:03:43.47
	50000	00:7:05.49	00:03:58.76
	250000	00:11:27.75	00:06:34.88
	500000	00:16:40.00	00:09:00.7
	750000	00:22:14.69	00:11:01.23
	1000000	00:27:05.50	00:13:45.50
256 x 128 x 128	10000	00:54:09.98	01:16:40.02
	50000	00:55:04.00	01:16:45.03
	250000	00:59:48.57	01:20:01.02
	500000	01:04:43.66	01:27:10.63
	750000	01:10:53.06	01:29:50.85
	1000000	01:20:03.07	01:32:05.55

Таблица 3: Тест производительности оптимизированной C# + CUDA реализации

Volume Resolution	Particle Number	Execution time	
		Windows	Ubuntu
32 x 16 x 16	10000	00:00:01.81	00:00:01.22
	50000	00:00:01.81	00:00:01.27
	250000	00:00:02.32	00:00:02.94
	500000	00:00:02.94	00:00:01.52
	750000	00:00:03.09	00:00:01.62
	1000000	00:00:03.60	00:00:01.80
64 x 32 x 32	10000	00:00:02.72	00:00:04.11
	50000	00:00:03.10	00:00:04.07
	250000	00:00:03.28	00:00:02.58
	500000	00:00:03.51	00:00:04.43
	750000	00:00:04.16	00:00:04.48
	1000000	00:00:04.57	0000::03.05
128 x 64 x 64	10000	00:00:12.74	00:00:18.63
	50000	00:00:12.73	00:00:18.27
	250000	00:00:13.02	00:00:18.95
	500000	00:00:13.49	00:00:19.24
	750000	00:00:13.91	00:00:19.64
	1000000	00:00:14.31	00:00:19.81
256 x 128 x 128	10000	00:01:21.63	00:01:23.81
	50000	00:01:22.18	00:01:23.95
	250000	00:01:21.91	00:01:24.19
	500000	00:01:23.53	00:01:24.54
	750000	00:01:25.31	00:01:24.65
	1000000	00:01:22.84	00:01:25.01

Для большей наглядности построим графики зависимости времени, потраченного на одну итерацию, от параметров. Сравнивать также будем оригинальный и оптимизированный варианты программ. В качестве време-

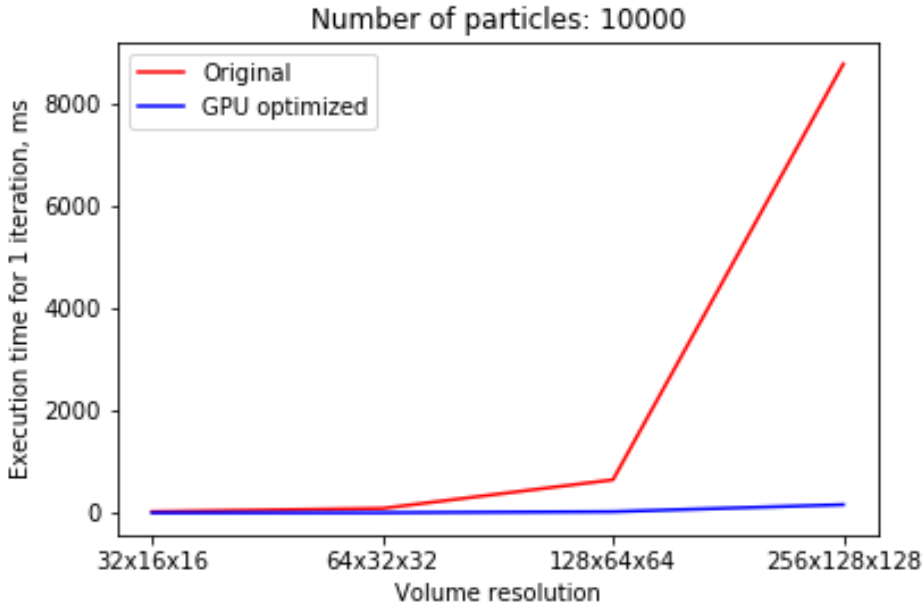


Рис. 8: Сравнение результатов 1.

ни выполнения при определенных параметрах будем брать среднее между Windows и Ubuntu и выражать в миллисекундах(рис. 8–10).

Вычислим ускорение. Для каждого набора параметров $p \in params$ определим пару чисел: среднее время выполнения одной итерации для обоих приложений, найдем их отошение и усредним по всем наборам параметров:

$$n = \frac{\sum_{p \in params} \bar{t}_{orig}(p) / \bar{t}_{opt}(p)}{|params|}$$

Ускорение получилось примерно в 80 раз. Стоит отметить, что в большинстве случаев оптимизированное приложение лучше отработало на Windows, а оригинальное наоборот, на Ubuntu.

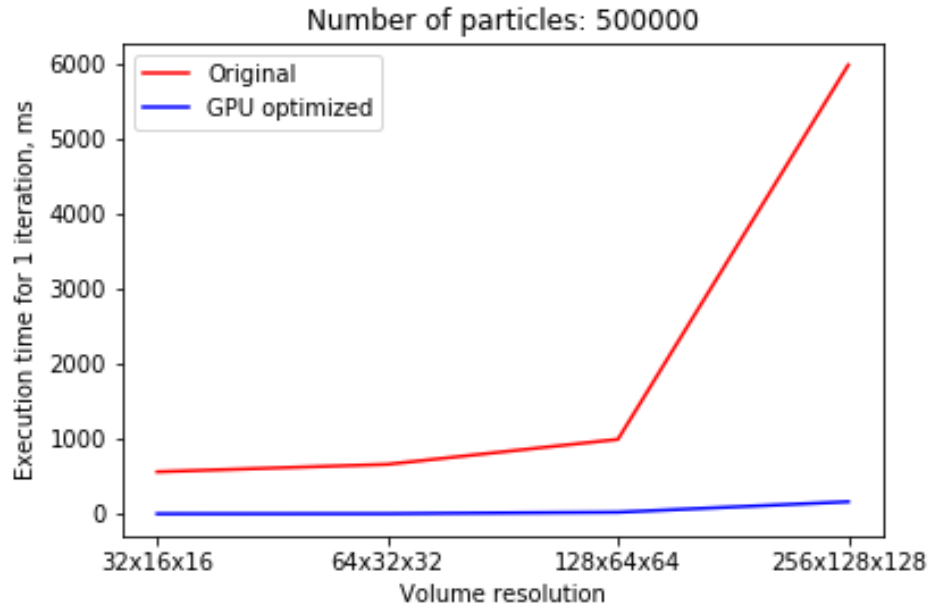


Рис. 9: Сравнение результатов 2.

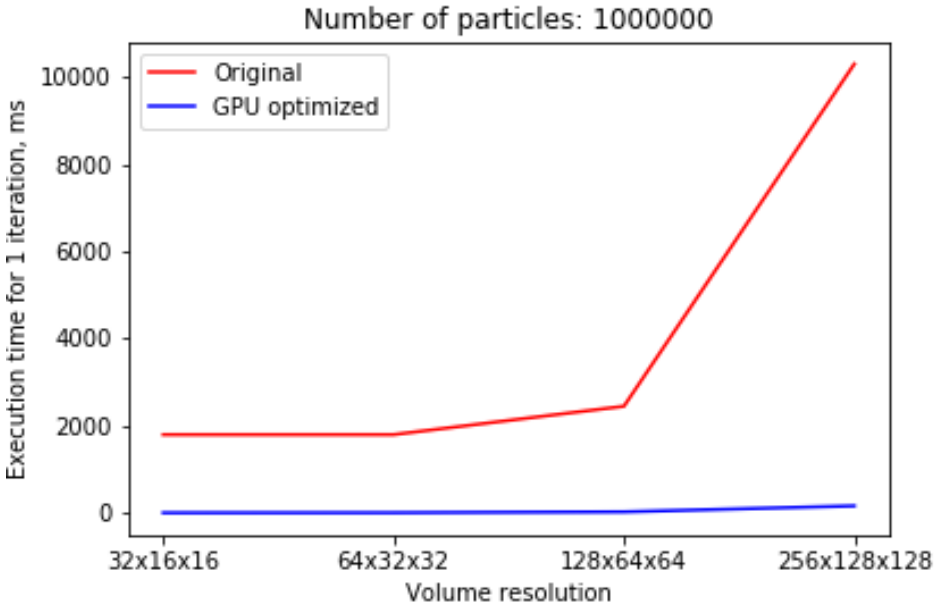
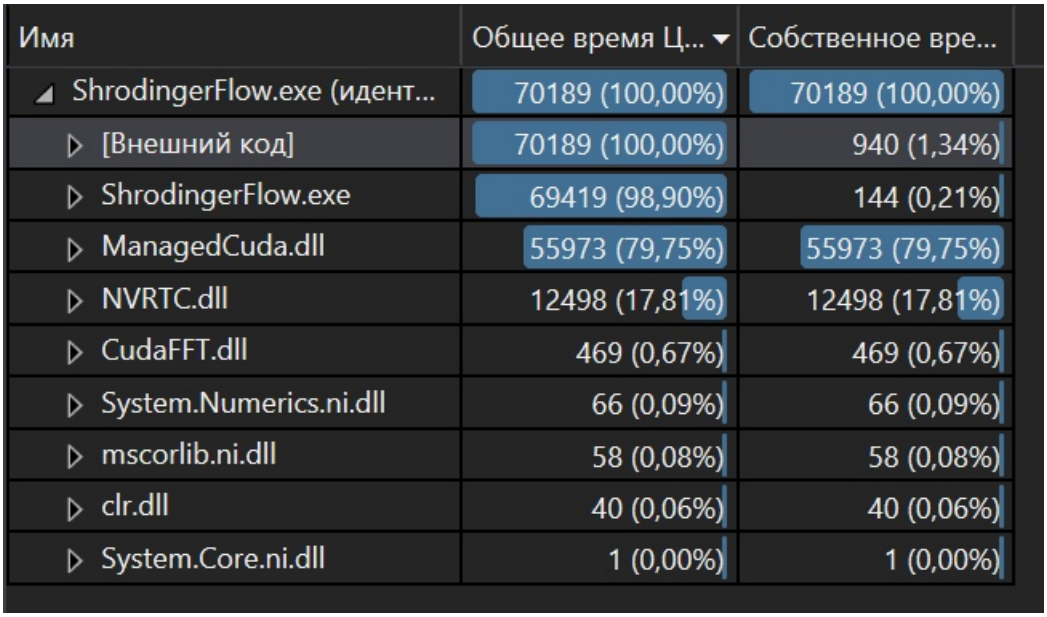


Рис. 10: Сравнение результатов 3.

5.2 Профилирование кода

Для оценки производительности различных частей приложения удобно воспользоваться профилировщиком, который предоставляет Visual Studio. Благодаря нему можно получить полезную информацию, которую потом можно использовать для дальнейшей оптимизации.

Сводка о компонентах(рис. 11) позволяет узнать, какая часть работы приходится на каждую часть проекта. Как можно видеть, практически все использование ресурсов приложения происходит в библиотеке ManagedCuda - это и есть работа с ГПУ(NVRTC - это компилятор ядер CUDA, который используеься только при инициализации и не играет никакой роли в самом процессе симуляции).



Имя	Общее время Ц...	Собственное вре...
ShrodingerFlow.exe (идент...	70189 (100,00%)	70189 (100,00%)
[Внешний код]	70189 (100,00%)	940 (1,34%)
ShrodingerFlow.exe	69419 (98,90%)	144 (0,21%)
ManagedCuda.dll	55973 (79,75%)	55973 (79,75%)
NVRTC.dll	12498 (17,81%)	12498 (17,81%)
CudaFFT.dll	469 (0,67%)	469 (0,67%)
System.Numerics.ni.dll	66 (0,09%)	66 (0,09%)
mscorlib.ni.dll	58 (0,08%)	58 (0,08%)
clr.dll	40 (0,06%)	40 (0,06%)
System.Core.ni.dll	1 (0,00%)	1 (0,00%)

Рис. 11: Сводка затрат ресурсов по компонентам.

Сводка по функциям(рис. 12) дает представление о том, какая часть работы приходится на каждый этап моделирования. Самый затратный процесс в ходе симуляции - обновление позиций частиц, а за ним идет преобразование состояния(инициализацию не учитываем, так как она не влияет на основную часть работы). Вклад остальных частей кода по отдельности довольно незначительный.

Круговая диаграмма(рис. 13) показывает, к каким категориям относится большинство проводимых операций.

ShrodingerFlow.exe (идентификатор процесса: 18680)	70189 (100,00%)
[Внешний код]	70045 (99,79%)
TestCase::Main	69419 (98,90%)
TestCase::iterate	69404 (98,88%)
source.assets.Particles.Particles::calculate_movement	44438 (63,31%)
source.assets.Particles.utils.VelocityHandler::update_particles	44433 (63,30%)
[Внешний вызов] ManagedCuda.CudaKernel::Run	44404 (63,26%)
[Внешний код]	18 (0,03%)
[Внешний код]	3 (0,00%)
source.assets.Discrete_space.ISF::Init	10014 (14,27%)
source.assets.Discrete_space.ISF::update_space	9404 (13,40%)
source.assets.Particles.Particles::init	3137 (4,47%)
source.assets.Discrete_space.ISF::update_velocities	2188 (3,12%)
source.assets.Discrete_space.ISF::add_circle	51 (0,07%)
[Внешний вызов] System.Random.Sample()	32 (0,05%)

Рис. 12: Сводка затрат ресурсов по функциям.

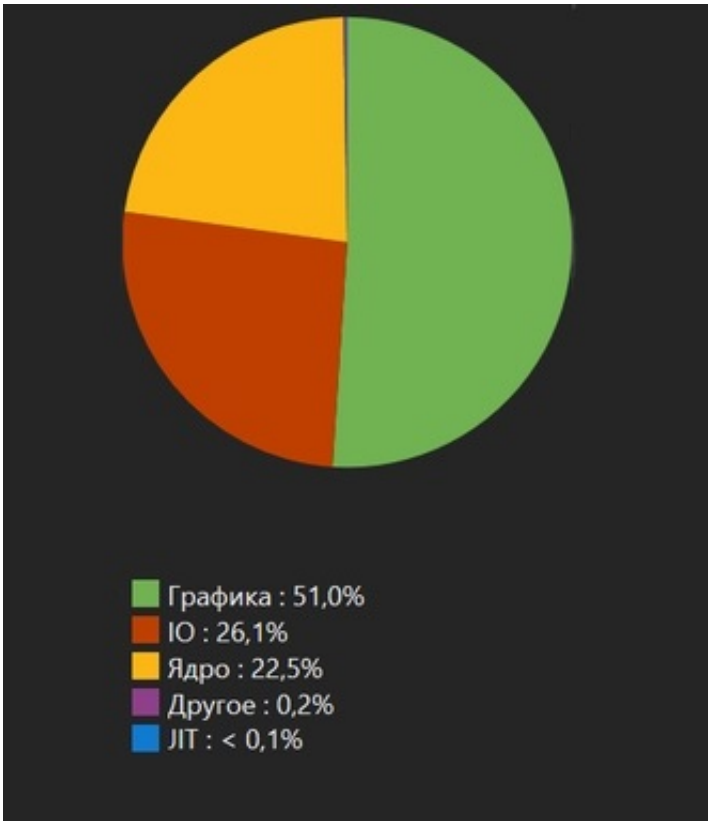


Рис. 13: Пять ведущих категорий операций.

5.3 Визуализация

Процесс был визуализирован на Unity (рис. 14, 15) с разрешением сетки 64 x 32 x 32 и 30000 частицами. Получившаяся в результате анимация была реалистичной и отрисовывалась в реальном времени достаточно быстро, чтобы укладываться в 60 кадров в секунду.

Рис. 14: Визуализация на Unity. Пример 1.

После того, как координаты переданы от симуляции в Unity, с внешним видом частиц можно делать что угодно, например, добавлять текстуры, определять размер и т.п. Также можно добавлять различные дополнительные эффекты или внешнее влияние, так как у проекта есть доступ не только к положению частиц, но и к их скоростям.

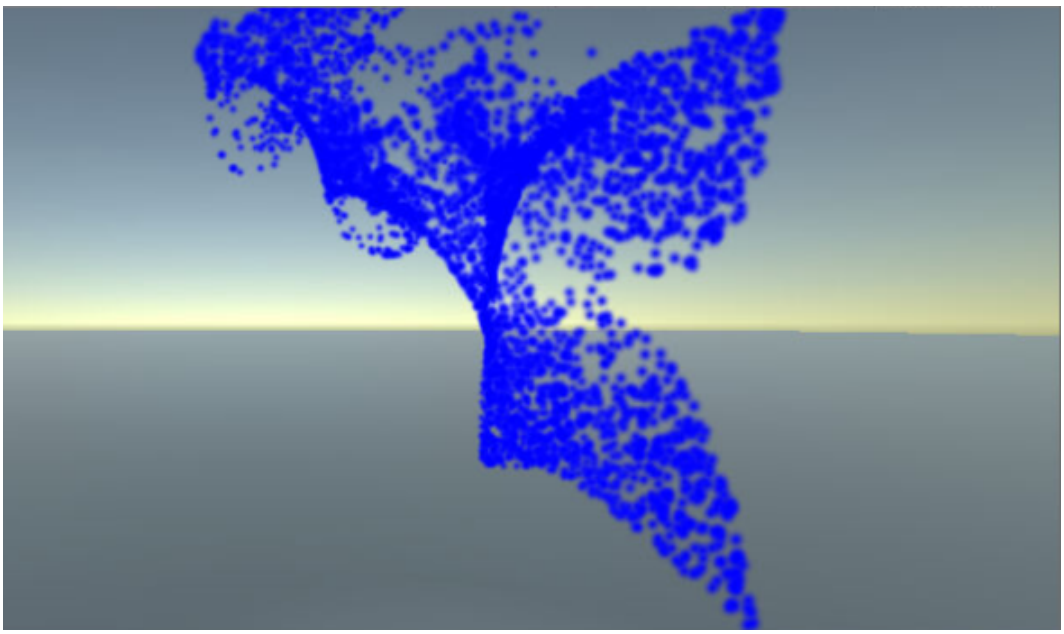


Рис. 15: Визуализация на Unity. Пример 2.

Выводы

В результате работы были достигнуты следующие цели:

1. Изучен метод моделирования несжимаемого потока Шредингера;
2. Разработана архитектура проекта;
3. Реализовано приложение на языке C#;
4. Проведено тестирование производительности;
5. Создан проект в Unity для наглядной демонстрации работы приложения.

Заключение

Созданное в ходе работы приложение отвечает поставленным в начале требованиям. Благодаря оптимизации при помощи технологий GPGPU оно обладает достаточно высокой производительностью, чтобы его можно было использовать для расчетов в реальном времени - увеличение производительности по сравнению с оригиналом примерно в 80 раз. Единственным ограничением доступности является необходимость в видеокарте Nvidia, в остальном приложение кросс-платформенное. Достаточно удобный интерфейс позволяет легко использовать его в других проетах, что было продемонстрировано на примере Unity.

В качестве направлений для дальнейшего развития стоит отметить:

1. улучшение общей архитектуры проекта для более простого внесения изменений и поддержки;
2. добавление отказоустойчивости;
3. создание полноценной библиотеки на основе проекта.

Исходный код проекта представлен в репозитории в GitHub [17].

Список литературы

[1] Hockney R. W., Eastwood J. W. Computer simulation using particles. New York: Taylor & Francis Group, 1988. 509 c.

[2] Ermak D. L. A computer simulation of charged particles in solution. I. Technique and equilibrium properties // The Journal of Chemical Physics. 2008. vol. 62, issue 10.

[3] Schofield P. Computer simulation studies of the liquid state // Computer Physics Communications. 1973. vol. 5, issue 1. P. 17-23.

[4] Liu M. B., Liu G. R., Zong Z., Lam K. Y. Computer simulation of high explosive explosion using smoothed particle hydrodynamics methodology // Computers & Fluids. 2003. vol. 32, issue 3. P. 305-322.

[5] Koshizuka S., Oka Y. Moving-Particle Semi-Implicit Method for Fragmentation of Incompressible Fluid // Nuclear Science and Engineering. 1996. vol. 123, issue 3. P. 421-434.

[6] Simon J. A. Introductory incompressible fluid mechanics // lecture notes. 2015.

[7] Chern A., Knöppel F., Pinkall U., Schröder P., Weißmann S. Schrödinger's smoke // ACM Transactions on Graphics. 2016. vol.35, No. 4, Article 77.

[8] Rockafellar R. T. Lagrange Multipliers and Optimality // SIAM Review. 1993. vol. 35, issue 2. P. 183–238.

[9] Zhu H. P., Zhou Z. Y., Yang R. Y., Yu A. B. Discrete particle simulation of particulate systems: A review of major applications and findings // Chemical Engineering Science. 2008. vol. 63, issue 23. P. 5728-5770.

[10] Almgren A. S., Bell J. B., Colella P., Marthaler T. A Cartesian Grid Projection Method for the Incompressible Euler Equations in Complex Geometries // SIAM Journal on Scientific Computing. 1997. vol. 18, issue 5. P. 1289 - 1309.

[11] Bayındır C. Compressive Split-Step Fourier Method // TWMS Journal of Applied and Engi-neering Mathematics. 2015. vol. 5, No. 2, P298-306.

[12] Ryoo S., Rodrigues C. I., Baghsorkhi S. S., Stone S. S., Kirk D. B., Hwu W. W. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA // PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. New York: Association for Computing Machinery, 2008. P. 73–82.

[13] Du P., Weber R., Luszczek P., Tomov S., Peterson G., Dongarra J. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming // Parallel Computing. 2012. vol. 38, issue 8. P. 391-407.

[14] Nobile M., Cazzaniga P., Besozzi D., Pescini D., Mauri G. cuTauLeaping: A GPU-Powered Tau-Leaping Stochastic Simulator for Massive Parallel Analyses of Biological Systems // PloS one. 2014. vol. 9. e91963.

[15] Nickolls J., Buck I., Garland M., Skadron K. Scalable Parallel Programming with CUDA // ACM Queue. 2008. vol. 6, No. 2. P. 40–53

[16] Kruchten, Philippe What is the rational unified process? // The Rational Edge. 1999.

[17] Репозиторий проекта на платформе GitHub [Электронный ресурс]: URL:https://github.com/IceWind2/ShrodingerFlow(дата обращения: 02.06.2020).