**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ  
ФЕДЕРАЦИИ**

Федеральное государственное автономное образовательное учреждение  
высшего образования

**«Национальный исследовательский**

**Нижегородский государственный университет им. Н.И. Лобачевского»**

**Институт информационных технологий, математики и механики**

Направление подготовки: «Прикладная математика и информатика»  
Магистерская программа: «Вычислительные методы и суперкомпьютерные технологии»

**МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ**

**«Моделирование распространения света**

**с применением технологии Intel OneAPI»**

|  |  |
| --- | --- |
|  | Выполнил: студент группы 3821М1ПМвм |
|  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Николаев Д. Э. |
|  | (подпись) |

|  |  |
| --- | --- |
|  | Научный руководитель: доцент кафедры МОСТ, к. т. н. |
|  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Горшков А. В. |
|  | (подпись) |

Нижний Новгород

2023

# Оглавление

[Оглавление 2](#_Toc133776189)

[Введение 3](#_Toc133776190)

[1. Постановка задачи 4](#_Toc133776191)

[2. Метод решения (об алгоритме распространения фотонов) 5](#_Toc133776192)

[3. Обзор метода Монте-Карло Ошибка! Закладка не определена.](#_Toc133776193)

[4. Обзор технологий параллельных вычислений 16](#_Toc133776194)

[5. Обзор методов программирования на GPU 18](#_Toc133776195)

[3.1. Технология Nvidia CUDA 18](#_Toc133776196)

[Архитектура CUDA и язык CUDA C 18](#_Toc133776197)

[~~Взаимодействия центрального и графического процессоров~~ 18](#_Toc133776198)

[~~Характеристики устройства~~ 20](#_Toc133776199)

[5.2. Технология Intel OneAPI 22](#_Toc133776200)

[Стандарт OneAPI и язык DPC++ 22](#_Toc133776201)

[Архитектура 23](#_Toc133776202)

[Вычислительные единицы DPC++ 24](#_Toc133776203)

[Доступ к памяти между устройствами 25](#_Toc133776204)

[6. Подсистема памяти многопоточного приложения Ошибка! Закладка не определена.](#_Toc133776205)

[7. Программная оптимизация 27](#_Toc133776206)

[8. Генерация псевдослучайных чисел 28](#_Toc133776207)

[Линейный конгруэнтный метод 28](#_Toc133776208)

[9. Параллельная реализация метода Монте-Карло OneAPI 33](#_Toc133776209)

[Результаты моделирования 33](#_Toc133776210)

[Заключение 35](#_Toc133776211)

[Литература 36](#_Toc133776212)

# Введение

Одним из наиболее перспективных направлений в биомедицине являются методы оптической диагностики. Они основаны на использовании света в видимом и инфракрасном диапазонах для зондирования биологических объектов и анализа рассеянного света. По сравнению с другими методами диагностики, такими как рентгеновские лучи или магнитно-резонансная томография, оптические методы имеют несколько преимуществ, среди которых можно выделить безопасность, неинвазивность, высокое разрешение и низкую стоимость. В последние годы развитие искусственного интеллекта сильно влияет на методы оптической диагностики, что требует соответствующего программного обеспечения для подготовки большого количества данных для машинного обучения.

Данные оптической диагностики могут быть получены путем моделирования распространения света внутри тканей биологических объектов. Сложность разработки такого ПО заключается в применении адекватной математической модели, для которой получить аналитическое описание в общем случае невозможно. Это объясняется тем, что биологические ткани задают сложные граничные условия, для которых в общем виде интегро-дифференциальное уравнение переноса излучения не имеет аналитического решения.

Наиболее широко применяемый метод для решения задачи физического моделирования распространения света внутри биологических тканей является метод Монте-Карло. Основная идея этого метода заключается в многократном повторении случайных независимых испытаний. Однако вычисление всех траекторий фотонов из пучка света может занять достаточно много времени, так как такая задача требует высокой точности и большого количества итераций.

В данной дипломной работе будет рассмотрено моделирование процесса распространения света методом Монте-Карло на графическом ускорителе с применением технологии Intel OneAPI. Будет проведен анализ возможностей и ограничений вычислительных систем с использованием GPU для данного метода, а также исследованы способы оптимизации вычислений на GPU. Результаты исследования могут быть использованы для улучшения эффективности вычислительных систем при реализации метода Монте-Карло в различных приложениях.

# Постановка задачи

В данной диссертации требуется разработать программное обеспечение для решения задачи физического моделирования распространения света для плоскопараллельной геометрии многослойной среды, выполнить перенос программной реализации на графический ускоритель.

В работе необходимо провести анализ возможностей и ограничений вычислительных систем, исследовать способы оптимизации вычислений, а также провести сравнительный анализ результатов полученных реализаций для решения поставленной задачи с применением метода Монте-Карло.

Решение поставленной задачи разбивается на следующие этапы:

1. Реализация последовательного алгоритма моделирования.
2. Разработка ПО для визуального контроля вычислений.
3. Реализация параллельного алгоритма моделирования.
4. Перенос параллельной реализации для вычислений на графическом ускорителе.
5. Профилирование и оптимизация полученной реализации.

Необходимо также провести исследование производительности алгоритма и сравнительный анализ полученной реализации.

# Модель распространения света

Метод Монте Карло выполняет многократное повторение случайных независимых испытаний, обсчитывая сложную математическую модель. На основе накопленных статистических данных делается вывод о вероятностных характеристиках исследуемого объекта.

Фотоны пучка света распространяются в многослойной среде. Каждый слой характеризуется следующими оптическими параметрами: коэффициентом рассеяния , коэффициентом поглощения , параметром анизотропии или фазовой функцией рассеяния показателем преломления *n*, толщиной, и формой границ. Значения показателей преломления внешней среды также учитываются при расчете.

Распространение фотона в среде описывается в декартовых координатах. Положение фотона определяется координатами (x, y, z), а текущее направление движения – направляющими косинусами:

где – орт направления скорости, – орты координатных осей.

Отражение фотона на границе раздела сред, имеющих разные показатели преломления, рассчитывается в соответствии с законом Френеля для неполяризованного излучения:

Распространение фотона в среде описывается в декартовых координатах. Положение фотона определяется координатами а текущее направление движения вектором скорости. Отражение фотона на границе раздела сред, имеющих разные показатели преломления, рассчитывается в соответствии с законом Френеля для неполяризованного излучения. Углы преломления определяются в соответствии с законом Снеллиуса [6]:

Рассмотрим итерацию алгоритма. Пучок фотонов начинает движение от источника излучения. Точка входа фотона в среду и его начальное направление определяются в соответствии с заданными параметрами падающего пучка, определяющими угловое и пространственное распределение интенсивности. Далее, исходя из параметров верхнего слоя (единственного в случае однослойной среды), происходит расчет длины свободного пробега. Длина свободного пробега определяется функцией плотности вероятности

где средняя длина свободного пробега определяется как

Случайная длина свободного пробега определяется в соответствии со следующей формулой:

,

где - случайная величина, равномерно распределенная на интервале .

Изменение направления движения фотона рассчитывается при каждом акте рассеяния определяемое фазовой функцией рассеяния:

Рассеиватели обычно считаются сферически симметричными, в связи с чем, величина считается равномерно распределенной на отрезке , а угол рассчитывается в соответствии с задаваемой фазовой функцией единичного рассеивателя.

Направляющие косинусы вектора скорости при рассеянии изменяются следующим образом:

Если угол движения фотона близок к нормальному:

то изменение направляющих косинусов вычисляется по следующим формулам:

После вычисления случайной длины свободного пробега рассчитываются новые координаты фотона по формулам:

,

где – начальные координаты фотона.

После вычисления новых координат фотона обработка одного акта рассеяния считается завершенной, и последовательность действий повторяется: вычисляется новая длина свободного пробега и новые направляющие косинусы вектора скорости.

Учет поглощения происходит следующим образом. Для повышения статистики проводимого расчета каждому фотону присваивается начальный вес, который уменьшается при каждом рассеянии на величину

где – текущий вес фотона.

# Последовательный алгоритм решения методом Монте Карло

Реализация последовательного алгоритма стала первым этапом в данной работе. Этот этап включает в себя перевод математического алгоритма в программный код и его тестирование на различных данных.

В качестве языка программирования выбран C++, так как одним из главных его преимуществ является высокая производительность и близость к аппаратному уровню, которая позволяет более точно распоряжаться ресурсами вычислительной системы.

Близость к аппаратному уровню языка C++ так же влияет важную роль для проведения сравнительных результатов производительности и позволит наиболее точно оценить эффективность.

## Реализация последовательного алгоритма на CPU

Программная реализация алгоритма моделирования распространения света опирается на математическую модель, которая описана в предыдущей главе. Она разделяется на несколько шагов:

1. Инициализация параметров эксперимента, параметров среды и фотонов
2. Вычисление траекторий движения фотонов в заданной среде
3. Сохранение результатов моделирования

Рассмотрим следующие типы данных, которые использует алгоритм.

Каждый слой ткани описывается классом **LayerStruct**.

struct LayerStruct

{

double z0, z1;

double n;

double mua;

double mus;

double anisotropy;

double cos\_crit0;  
double cos\_crit1;

}

Объекты класса хранят в себе параметры геометрии и оптические характеристики слоя:

* z0, z1 - координаты границы слоя для плоскопараллельной геометрии
* n - коэффициент преломления
* mua - коэффициент поглощения
* mus - коэффициент рассеяния
* anisotropy - коэффициент анизотропии
* cos\_crit0, cos\_crit1 - предельные значения косинусов угла падения фотона на границы слоя z0 и z1 соответственно для расчета отражения/преломления.

Параметры моделирования и структура многослойной ткани описываются классом **InputStruct**:

struct InputStruct

{

long num\_photons;

double Wth;

double dz;

short nz;

short num\_layers;

short Nx;

short Ny;

short Nz;

short num\_output\_layers;

LayerStruct\* layerspecs{ nullptr };   
}

Объекты класса хранят в себе информацию о каждом слое, пространстве моделирования и фотонах:

* num\_photons – количество фотонов моделирования
* Wth – предельный минимальный вес фотона.
* dz – размер ячейки слоя по координате z.
* nz – количество ячеек по координате z
* num\_layers – количество слоев в ткани для моделирования
* Nx, Ny, Nz – размер сетки для сохранения интенсивности фотонов в точке пространства
* num\_output\_layers – количество слоев ткан для сохранения результатов моделирования
* layerspecs – указатель на массив с описанием каждого слоя

Алгоритм моделирования хранит информацию о каждом фотоне в объектах класса **PhotonStruct**:

struct PhotonStruct

{

double x{ 0 }, y{ 0 }, z{ 0 };

double ux{ 0 }, uy{ 0 }, uz{ 0 };

weight\_tracker<float> w;

bool is\_dead{ false };

size\_t layer{ 0 };

double sleft{ 0 };

double step\_size{ 0 };

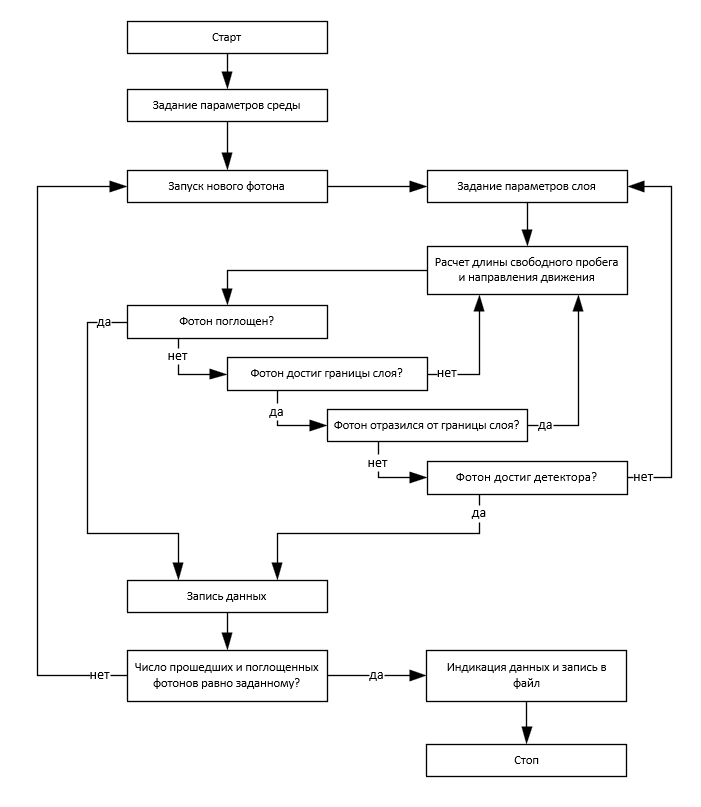
const InputStruct\* input;

}

Объекты класса хранят информацию о состоянии фотона:

* x, y, z – координаты фотона в пространстве
* ux, uy, uz – векторы направления фотона в пространстве
* w – вес, энергия фотона
* is\_dead – логическая величина, которая определяет состояние жизненного цикла фотона и необходимость продолжать моделирование
* layer – номер слоя, в котором находится фотон
* sleft – оставшееся смещение фотона, которое необходимо для расчета достижения границы слоя
* step\_size – размер шага внутри слоя, которое он выполнит в направлении своего движения
* input – указатель на структуру, определяющую параметры среды моделирования

Общая логика алгоритма представлена на рисунке ниже:



Фотон, выпущенный из излучателя, продвигается вглубь многослойной структуры ткани. Проходя каждый слой, он теряет какое-то количество энергии, по исчерпанию которой останавливается. В момент остановки выполняется регистрация позиции фотона и его состояния в пространстве .

Данные о позиции фотона регистрируются в трехмерной матрице. Для каждой ячейки матрицы определены интервалы, которые сопоставляется с каждой точкой трехмерной многослойной среды. Таким образом, при остановке фотона значения координат сохраняются в соответствующую ячейку.

В качестве выходных данных используется файл с сериализованными значениями этой трехмерной матрицы, которая содержит карту интенсивности фотонов для последующей обработки (визуализации).

## Генерация псевдослучайных чисел

Одно из главных требований, которое накладывает математическая модель на метод Монте-Карло, заключается в использовании случайных равномерных величин на интервале . Несоблюдение этого требования неизбежно придет к искажению результатов моделирования.

### Линейный конгруэнтный метод

Одним из наиболее простейших и часто используемых методов генерации псевдослучайных чисел является линейный конгруэнтный метод. Процедура генерации использует операцию взятия остатка деления аргумента на аргумент : . Каждое последующее случайное число использует в своих расчетах результат предыдущих вычислений по следующей формуле:

где:

* – модуль ();
* множитель ()
* приращение ()

Модуль определяет максимальную длину последовательности. Поиск подходящего числа может быть нетривиальной задачей. С одной стороны, модуль должен быть достаточно большим, т.к. период не может иметь больше элементов. С другой стороны, если модуль выбран как степень двойки , то вычисление остатка будет выполняться быстрее. Это обусловлено тем, что в общем случае операция взятия остатка использует целочисленное деление, но в частном случае компилятор заменяет её простой логической операцией AND. Также широко распространен выбор наибольшего простого числа , меньшего, чем . Можно доказать, что в этом случае младшие разряды получаемого случайного числа ведут себя так же случайно, как и старшие, что положительно сказывается на всей последовательности случайных чисел в целом. В качестве примера можно привести одно из чисел Мерсенна, равное , и таким образом, константа определяется как .

В качестве выбирается число из промежутка для генерации последовательно псевдослучайных чисел. Полученная последовательность называется линейной конгруэнтной последовательностью.

Существует теорема, которая позволяет определить, возможно ли достижение периода максимальной длины для конкретных значений , и :

**Теорема.** Линейная конгруэнтная последовательность, определенная числами M, k, b и r0, имеет период длиной M тогда и только тогда, когда:

* числа и взаимно простые;
* кратно для каждого простого , являющегося делителем ;
* кратно 4, если кратно 4.

## Тестирование и визуализация результатов моделирования

Тестирование программных реализаций является важным этапом разработки программного обеспечения и необходимо для обеспечения должного качества и надежности программ. Ошибки могут возникать в разных частях программы, например, в программной логике и проблемами в работе с памятью. Если эти ошибки не будут обнаружены и исправлены, то они отрицательно повлияют на корректность результатов.

Тестирование является неотъемлемым шагом при оптимизации программ. После выявления узких мест разработчик может провести оптимизацию, чтобы улучшить производительность, но случайно допустить некоторые ошибки. Как правило, они не воспроизводятся в тех сценариях, над улучшением которых проводятся оптимизации, следовательно, могут остаться необнаруженными на этом этапе.

В настоящее время наибольшую популярность имеют инструменты автоматического тестирования такие как, например, Google Test или Boost Test. Они позволяют очень быстро проверять большое количество сценариев и тем самым исключить человеческий фактор во время проверки. Как правило, такие тесты могут быть применены к тем участкам кода, которые имеют строгий алгоритм выполнения, дают однозначный ожидаемый ответ для заданных входных данных.

К сожалению, автоматическое тестирование не всегда применимо. Программный код может не иметь строгого алгоритма исполнения, например, при использовании генераторов случайных чисел или параллельном исполнении. В данном случае прибегают к ручному тестированию программ.

При проведении ручного тестирования важно учитывать человеческий фактор, который может сильно повлиять на результаты. Так, например, при большом количестве выходных данных человек может упустить из внимания какие-то ошибки. Важно сокращать количество объектов внимания и представлять их в наиболее понятном виде, например, визуально или в сводных таблицах.

### Визуализатор выходных данных алгоритма моделирования

С целью улучшения качества тестируемого кода был разработан модуль для визуализации карты интенсивности фотонов, так как он позволяет представить результаты работы алгоритма в наглядном и понятном виде.

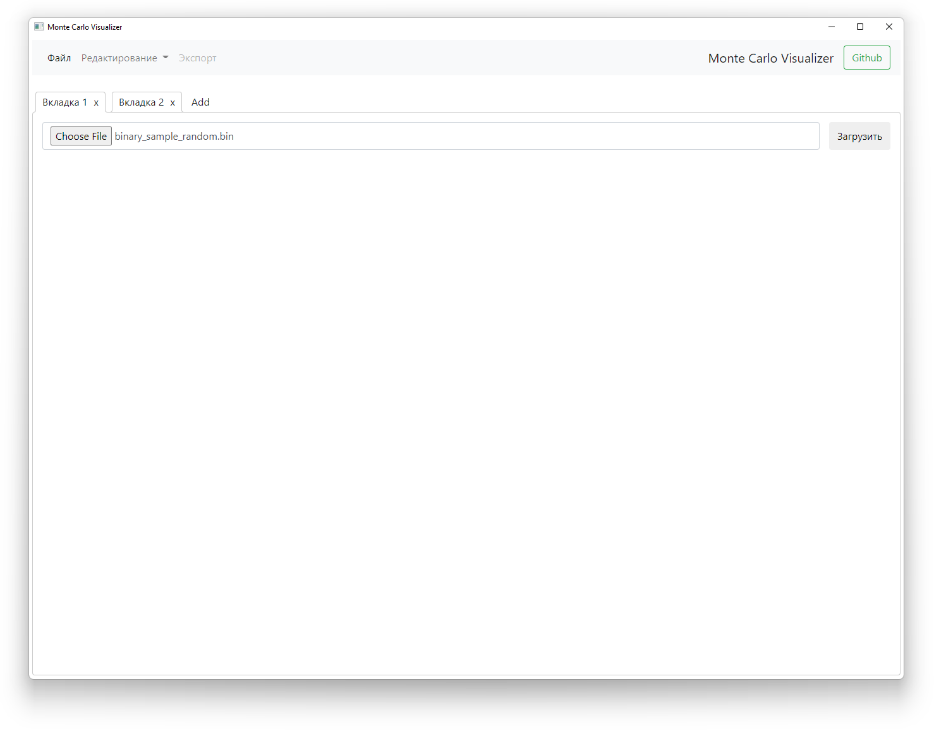
В реализации данного модуля использован язык C++, т.к. он тесно связан с основной реализацией и использует те же самые заголовки и структуры данных для своей работы.

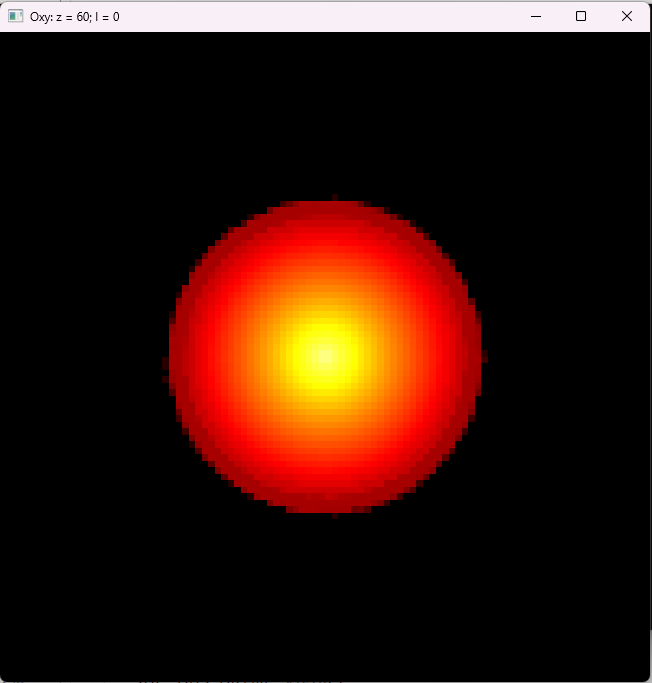
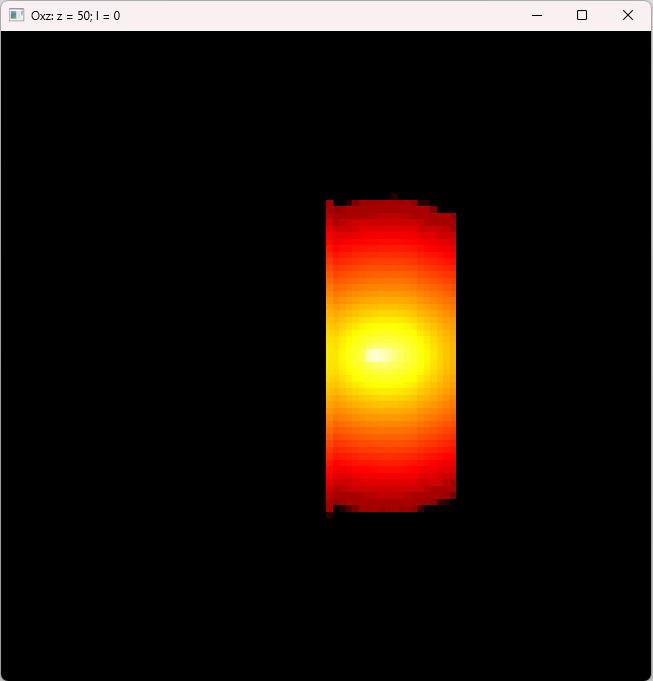
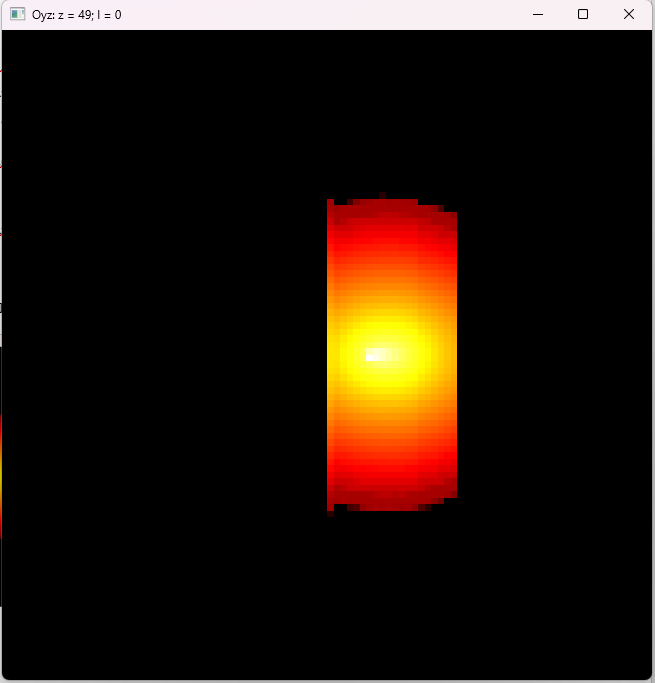
В качестве инструмента визуализации используется библиотека OpenGL для C++, которая является достаточно простой в использовании и одной из наиболее популярных библиотек для рисования графики в реальном времени.

В качестве инструмента для создания графического интерфейса используется Chromium Embedded Framework для C++, который использует движок Chromium для рендеринга веб страниц. Это позволяет воспользоваться HTML разметкой для создания приложений.

Во время запуска визуализатора пользователю необходимо выбрать файл для визуализации. После загрузки файла трехмерная матрица интенсивности визуализируется в трех разных координатных плоскостях Oxy, Oyz, Oxz.

Для обеспечения интерактивности визуализатора использована обработка событий клавиатуры и мыши, которые позволяют пользователю управлять отображаемыми объектами.



На продемонстрированных рисунках можно наблюдать рассеивание света непосредственно от источника излучения в трех плоскостях.

Интенсивность фотонов определяется цветом и оттенком. Так, наиболее яркие участки соответствуют наиболее интенсивным участкам карты, т.е. наибольшему суммарному количеству потерянного веса в этой точке пространства.

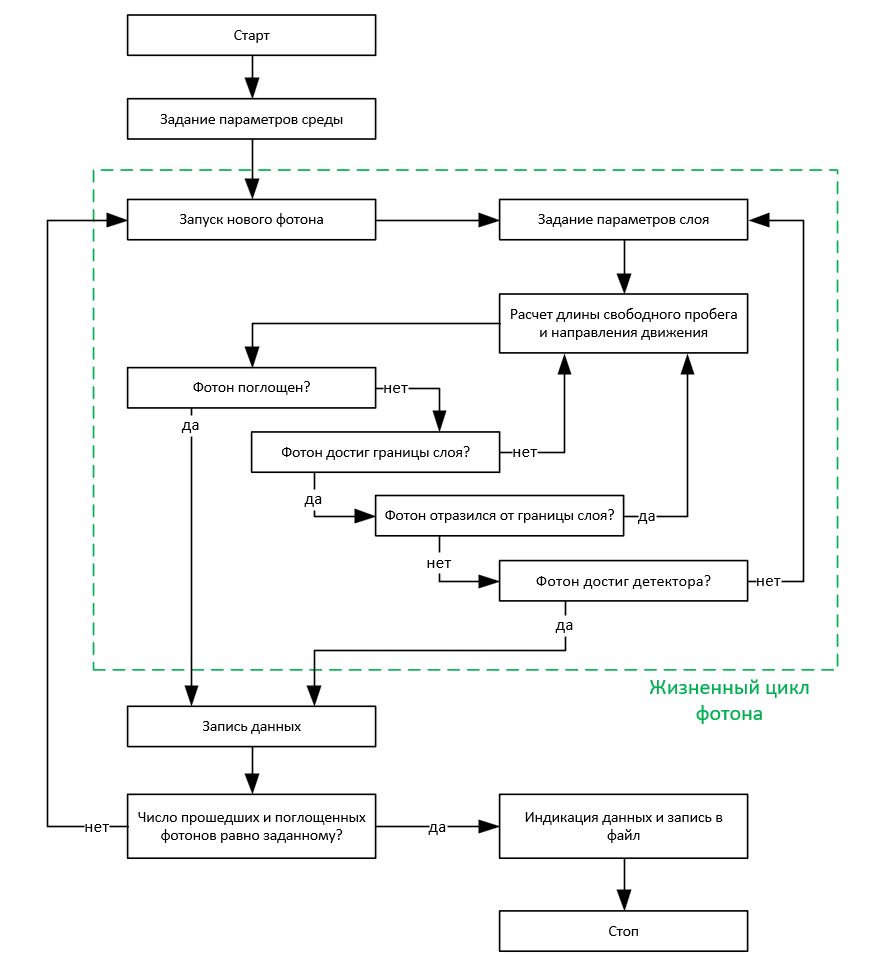
## Результаты экспериментов

# Параллельный алгоритм решения методом Монте Карло

Требование к высокой производительности при решении поставленной задачи привело к необходимости разработки параллельного алгоритма и соответствующего архитектурного решения, которое позволяет распределять задачи между несколькими вычислительными потоками, ускоряя выполнение вычислений.

## Архитектурное решение

Возможность параллельной реализации обусловлена наличием последовательности действий, которые не зависят от результатов друг друга. Главная особенность текущего решения методом Монте Карло предполагает, что каждое испытание проводится независимо. Эту особенность можно легко увидеть, если схематично описать жизненный цикл фотона:

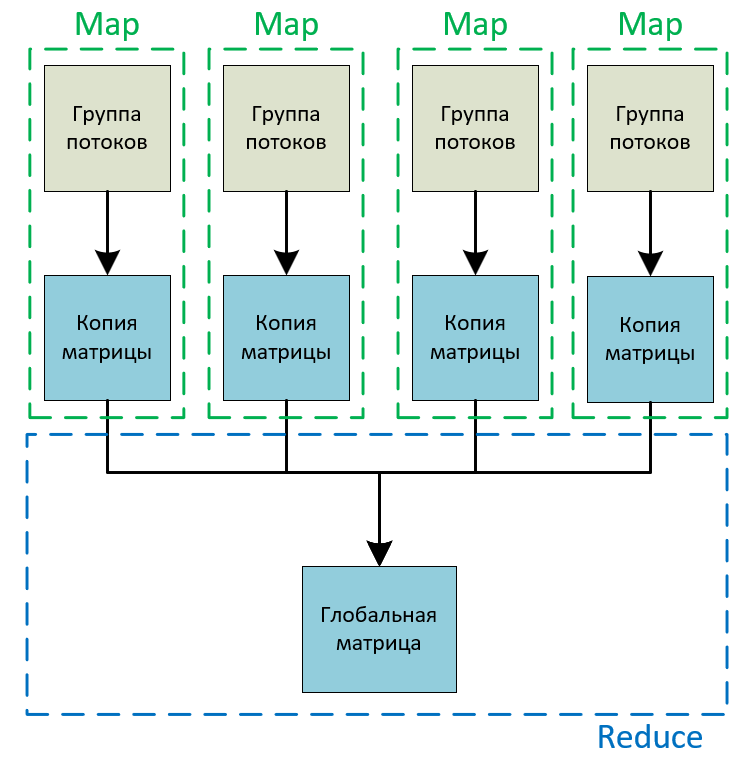


Каждое испытание проводится независимо друг от друга, таким образом они могут быть выполнены параллельно, с учетом, что все данные будут синхронизировано сохранены в памяти устройства. В качестве сохраняемых данных записывается потерянный вес фотона. Учитывая особенность генерации случайных данных, невозможно предсказать какой следующий адрес будет использован для сохранения результатов. Следовательно, операции сохранения требуют применение какого-то примитива многопоточной синхронизации, который предоставит эксклюзивный доступ для чтения и записи.

Использование примитива синхронизации предполагает упорядоченный доступ к одним и тем же ячейкам памяти для разных ядер, что приводит к простаиванию ядер, ожидающих запись. С увеличением количества вычислителей неизбежно растет время ожидания такого доступа к общему ресурсу, поэтому общий прирост производительности снижается. И наоборот, снижение количества вычислителей приведет к снижению времени ожидания на запись. Таким образом, рост времени ожидания доступа может быть снижен путем разделения областей памяти и балансировки вычислений на них.

С учетом всего выше сказанного, было рассмотрено архитектурное решение, в основе которого лежит концепция MapReduce. Обработка большого объема данных разбивается на два этапа: Map и Reduce.

На этапе Map данные разбиваются на небольшие части и обрабатываются параллельно разными группами потоков. Затем результаты Map-операции собираются и передаются на этап Reduce, где они объединяются и обрабатываются для получения окончательного результата.



Основные положения такой реализации:

1. Потоки-вычислители (work-items) объединяются в несколько групп потоков (work-group) фиксированного размера.
2. Каждая группа имеет собственную область памяти, к которой не имеет доступ другие группы.
3. Внутри группы каждый вычислитель получает эксклюзивный доступ к каждой ячейке матрицы весов.
4. Работа вычислителей происходит независимо от результатов других вычислителей.
5. После завершения работы всех вычислителей выполняется объединение всех копий матриц в итоговую результирующую матрицу.

Данное архитектурное решение допускает использование как разных типов памяти, так и разных примитивов синхронизации.

## Генерация псевдослучайных чисел

В случае параллельной обработки данных это требования так же должно выполняться относительно каждой сгенерированной пары последовательностей псевдослучайных чисел в потоках, вычисляющих траектории полета фотонов. Так, например, два потока могут инициализировать генератор псевдослучайных чисел с одинаковыми начальными значениями, и тогда траектории фотонов совпадут. Многократное повторение этой ошибки приведет к искажению результатов моделирования.

Во избежание пересекающихся последовательностей псевдослучайных чисел в данной работе будет применяться мультипликативный конгруэнтный генератор псевдослучайных чисел. Его главной особенностью является возможность получить доступ к последовательности псевдослучайных чисел из разных потоков за один шаг. Это позволяет организовать параллельную работу с последовательностью так, что каждый поток использует элементы этой последовательности.

### Мультипликативный конгруэнтный метод

В данной работе реализован алгоритм мультипликативного конгруэнтного генератора MCG59, который основан на линейном конгруэнтном методе:

В качестве выбирается число . В качестве и выбираются такие числа, чтобы каждый поток мог генерировать свою последовательность псевдослучайных чисел, не пересекающуюся с другими потоками. Таким образом требование к генератору псевдослучайных чисел будет выполнено, а скорость генерации увеличена.

## Обзор технологий параллельных вычислений

Наиболее принципиальным методом сокращения времени моделирования является параллелизация вычислений. До недавнего времени наиболее доступными системами для параллельных расчетов были слабосвязные кластеры, в которых вычисления производились на центральных процессорах общего назначения (CPU, Central Processing Unit). Однако такие кластеры достаточно дороги и сложны в эксплуатации. В последние годы с развитием графических процессоров (GPU, Graphics Processing Unit) возможности вычислительных систем значительно увеличились. В тоже время возросла сложность разработки и оптимизации программ для эффективного применения GPU в вычислительных задачах.

Выбор вычислительного устройства определяется вычислительными характеристиками этого устройства, а также технологиями и стандартами разработки. В рамках поставленной задачи, наибольший приоритет имеет, в первую очередь, производительность, а затем масштабируемость и возможности разработки. С другой стороны, некоторые технологии дают позволяют разрабатывать кросс-архитектурное ПО, что снижает сложность и стоимость разработки.

Рассмотрим существующие технологии и стандарты для параллельных вычислений:

1. OpenMP – это стандартная технология для параллельных вычислений на CPU. Она позволяет разрабатывать многопоточные программы на C, C++ и Fortran, используя директивы препроцессора и библиотеки OpenMP.
2. Nvidia CUDA – проприетарная технология компании Nvidia, которая позволяет использовать графические процессоры Nvidia для параллельных вычислений.
3. AMD HIP API – проприетарная технология компании AMD, которая позволяет использовать графические процессоры AMD для параллельных вычислений.
4. OpenCL – технология, которая позволяет использовать как GPU, так и центральные процессоры (CPU) для параллельных вычислений. Она используется для разработки кроссплатформенных параллельных приложений.
5. SYCL – это открытый стандарт параллельного программирования для гетерогенных систем, разработанный Khronos Group. Стандарт объединяет в себе возможности языка C++ и OpenCL, за счет этого предоставляет единый API для параллельного выполнения на различных устройствах, таких как CPU, GPU, FPGA.
6. Data Parallel C++ – это открытый стандарт параллельного программирования, который разрабатывается в рамках технологии OneAPI, созданной компанией Intel. Он основывается на SYCL, упрощая его применение и добавляя новые возможности для параллельного выполнения кода на различных устройствах, таких как CPU, GPU, FPGA и других ускорителях.

Каждая из этих технологий имеет свои преимущества и ограничения, и её выбор зависит от конкретной задачи и характеристик используемого оборудования.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | OpenMP | CUDA | AMD HIP | OpenCL | SYCL | DPC++ |
| CPU | + |  |  | + | + | + |
| Поддержка GPU AMD |  |  | + | + | + | + |
| Поддержка GPU Nvidia |  | + |  | + | + | + |
| Поддержка GPU Intel Arc |  |  |  | + | + | + |

## Стандарт Data Parallel C++

Открытый стандарт OneAPI базируется на ряде других открытых стандартов, где наибольший вклад сделан OpenCL поддерживаемый компанией Khronos [7]. Унифицированная программная модель, которая основана на открытых стандартах и подходит для кроссплатформенного программирования широкого спектра устройств CPU, GPU, FPGA и т.д. В эту технологию включает в себя наборы:

* компиляторов для разных языков, основным из которых является DPC++;
* библиотек алгоритмов и структурных компонентов;
* анализаторов и утилит для отладки;
* платформ-зависимых решений;

Поддерживается много вендоров, и это дает возможность быть независимым от какого-то одного. Программная модель гарантирует переносимость кода между устройствами, но не гарантирует одинаковую производительность на разных устройствах.

OneAPI имеет свой собственный язык DPC++ – Data Parallel C++ и отдельный компилятор для него. Синтаксис основан на уже существующих стандартах C++, что весьма отлично от низкоуровневого OpenCL. Это позволяет сочетать в себе высокую производительность и относительную малую когнитивную сложность восприятия кода.

OneAPI предоставляет легкий в освоении и изучении API, который не требует таких больших знаний, как в OpenCL. Одним из главных преимуществ является возможность отладки на CPU, что позволяет останавливать выполнение программы и изучать что происходит внутри.

Компания Intel имеет большое количество востребованных высокопроизводительных библиотек, которые зарекомендовали себя уже очень давно. С появлением OneAPI все библиотеки получили еще одно большое преимущество быть использованными на других устройствах и сохранить высокую производительность.

Появление такого инструмента не только увеличивает возможности разработчиков, но и расширяет рынок совместимого оборудования.

### Архитектура

Выполнение параллельных вычислений на специализированных устройствах подразумевает написание двух главных составных частей:

1. **Написание управляющего кода.**  
   Целью этой части разработки является подготовка данных и доступа к ним, а также управление очередью задач для других устройств (или множества разнородных устройств). Строится граф зависимостей и балансируется нагрузка. Эта часть программы компилируется и выполняется на СPU, как обычная программа, но также имеет возможность создать задачу и отправить ее на исполнение устройствами (выполнить submit). Код, выполняющийся на CPU, называют host domain code.
2. **Написание параллельного кода.**  
   Целью этой части является проработка параллельных алгоритмов, оперирование предоставленными свыше данными, синхронизациями между параллельными операциями доступа к данным. Эта часть программы, называемая kernel, выполняется на устройстве. Код, исполняемый на целевом устройстве (CPU, GPU, FPGA), называют device domain code.

Еще одним преимуществом технологии является возможность балансировать нагрузку между разными устройствами. Так, например, код может одновременно выполняться и на CPU, и на GPU (нескольких), и на FPGA. Вместе с этим может выполняться синхронизация и доступ к данным.

Идея последовательного исполнения kernel на устройствах достигается с помощью очередей sycl::queue, при инициализации которых устанавливается тип устройства для исполнения. Сразу после инициализации данной структуры очередь будет готова принимать kernel к исполнению.

Стоит всегда учитывать, что каждое отдельно взятое устройство обладает некоторым количеством ресурсов, которые могут выступать в качестве ограничений, например, по памяти.

### Вычислительные единицы DPC++

Так как целевые устройства имеют похожую, но все же различную архитектуру, OneAPI предоставляет унифицированную абстракцию для параллельных вычислений. Исполнение *kernel* происходит специальными юнитами, которые называются ***work-item***. Можно провести аналогию с потоками операционной системы, которые выполняют последовательный код, написанный в *kernel*.

Множество из целого фиксированного числа *work-item* исполняющих один и тот же kernel объединяются в ***work-group***. Каждая такая группа является вычислительной единицей, которая реализует какую-то микропроцессорную архитектуру на целевом устройстве. В её распоряжении которой есть вычислительные ресурсы, такие как набор конвейер инструкций, регистры, кэш, память и т.д. инкапсулированные интерфейсом OneAPI. Из этого следует, что *work-group* между собой не имеют тривиального доступа к ресурсам друг друга.

Набор из *work-group* объединяется в ***Nd-range***, и в рамках него имеют один и тот же размер. *Work-group* размещаются линейно внутри *Nd-range* линейно и могут быть индексированы в 0, 1, 2, 3 – мерном пространстве, в зависимости от задач программиста. В один момент времени на целевом устройстве может быть загружен только один *Nd-range*.

Соответственно в рамках введенной иерархии, существует удобный способ идентификации каждого *work-item*:

1. ***global range*** – определяет общее число *work-item* в каждом измерении
2. ***local range*** – определяет число *work-item* в рамках *work-group*, в которой он исполняется.

auto local\_range = sycl::range<2>(work\_item\_size::x, work\_item\_size::y);

auto global\_range = sycl::range<2>(work\_item\_size::x \* work\_group\_size::x,

work\_item\_size::y \* work\_group\_size::y);

auto nd\_range = sycl::nd\_range<2>(global\_range, local\_range)

Внутри work-group может быть выполнена синхронизация между принадлежащими ей work-item:

nd\_item.barrier(sycl::access::fence\_space::local\_space);

Существуют ограничения, которые накладываются на код, написанный внутри kernel:

* Нет динамического выделения памяти
* Нет динамического полиморфизма
* Нет функциональных указателей
* Нет рекурсий

### Доступ к памяти между устройствами

Интеграция с языком С++ требует разделения областей не только выполнения инструкций между CPU и GPU, но и разделения пространства хранения данных.

Данные, размещенные на куче, подконтрольны только CPU, поэтому чтобы иметь доступ к данным требуется выполнять транзакции на чтение и запись между *host* и *device* памятью. Одним из основных, но единственным способом обращения к памяти является использование структуры:

sycl::buffer <T, dimensions>(T\* pointer, sycl::range<dimensions> sizes)

Классической проблемой многопоточных и многопроцессорных программ является одновременный доступ к данным на чтение и запись. После передачи указателя на какой-то участок памяти, любые манипуляции над этим участком теперь будут выполняться только через конкретный объект sycl::buffer.

Внутри непосредственно самого *kernel* требуется вызвать метод: get\_access<access::mode>(cgh) c указанием типа доступа к данным. Таким образом будет гарантироваться атомарность операций взаимодействия с памятью.

int main()

{

constexpr size\_t N = 1024;

sycl::queue gpu\_queue{ sycl::gpu\_selector{} };

// выделение памяти происходит на куче

std::vector<float> a(N, 1.0f), b(N, 2.0f), c(N, 0.0f);

// local scope

{

sycl::buffer<float, 1U> buf\_a(a.data(), a.size());

sycl::buffer<float, 1U> buf\_b(b.data(), b.size());

sycl::buffer<float, 1U> buf\_c(c.data(), c.size());

gpu\_queue.submit(

[&](sycl::handler& cgh)

{

auto in\_a = buf\_a.get\_access<sycl::access::mode::read >(cgh);

auto in\_b = buf\_b.get\_access<sycl::access::mode::read >(cgh);

auto out\_c = buf\_c.get\_access<sycl::access::mode::write>(cgh);

cgh.parallel\_for<class Add>(sycl::range<1>(N),

[=](sycl::id<1> index)

{

// Kernel code

out\_c[index] = in\_a[index] + in\_b[index];

}

);

}

);

}

// local scope закрывается, sycl::buffer<float, 1U> ,

// выделенные на стеке, разрущаются => передают доступ

// к данным std::vector<float> a, b, c обратно на CPU

for (const auto& v : c)

{

assert(v == 3.0);

}

}

Так же обязательно следует упомянуть о локальной памяти, которая может быть использована *work-group*:

sycl::accessor<float, 1U, sycl::access::mode, sycl::access::target::local> lmem\_a(size\_of\_a, cgh);

Таким образом данные могут храниться внутри *work-group* с более быстрым и защищенным (от других *work-group*) доступом к ним.

## Программная оптимизация

Одним из основных преимуществ использования графических процессоров для решения задач общего назначения является высокая степень параллелизма, позволяющая значительно повысить производительность по сравнению с ЦП. Поэтому для получения результата необходимо разработать параллельную реализацию алгоритма, учитывающую архитектурные особенности GPU.

Рассмотрим проблемы, выявленные в процессе адаптации параллельного алгоритма.

### Подсистема памяти

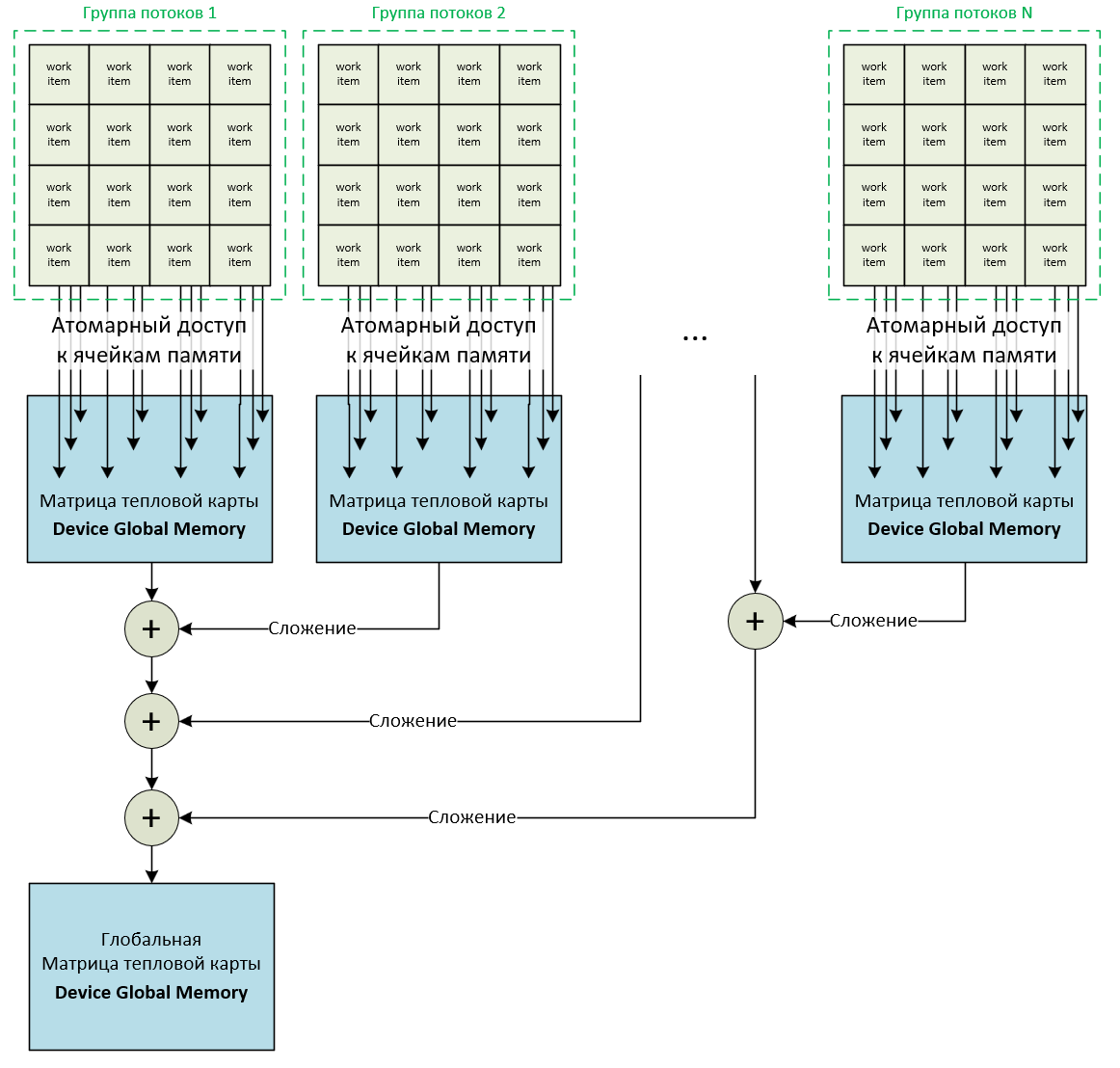
Проблема многих многопоточных реализаций заключается в синхронизированном доступе к общим ресурсам, например, таким как память. В частности, одной из таких проблем является одновременный доступ. Data race т.н. состояние гонки – это ошибка проектирования многопоточной системы, при которой нарушен синхронизированный доступ к общим ресурсам, от которых зависят результаты вычислений. Проблема устраняется введением блокировок при доступе, что, несомненно, сказывается на общей скорости вычислений. Существуют способы уменьшения количества совместных блокировок, некоторые из которых решаются на уровне архитектуры ПО.

Вычислительные устройства, такие как CPU и GPU, обладают двумя видами памяти:

1. Глобальная память
2. Локальная память

Глобальная память представляет из себя общий объем операционной памяти устройства, которым оно обладает и разделяет между всеми ядрами процессора.

Архитектурная альтернатива для параллельного алгоритма с использованием глобальной памяти устройства выглядит так:



Выделим преимущества и недостатки такой реализации. К плюсам можно отнести:

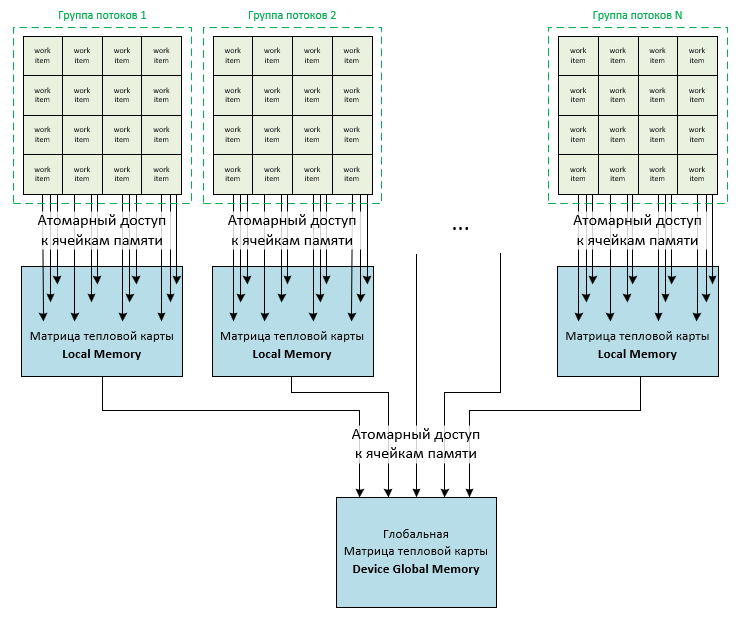
1. Среднее время ожидания доступа для записи/чтения определяется размером группы.
2. Количество групп вычислителей не ограничено, хорошая масштабируемость.

К минусам:

1. Большое потребление памяти.

Локальная память представляет из себя малый объем памяти с высокой пропускной способностью, который расположен непосредственно рядом с ядрами исполнения. Так же, она не имеет прямой адресации с глобальной областью памяти, поэтому при необходимости обмена данных приходится выполнять копирование. Локальная память обычно используется для хранения данных, которые не могут быть эффективно переданы между глобальной памятью и регистрами ядер исполнения. Поэтому локальная память обычно используется для хранения промежуточных результатов вычислений или для кэширования данных, которые часто используются внутри ядер исполнения.

Архитектурная альтернатива для параллельного алгоритма с использованием локальной памяти устройства выглядит так:



В рамках данной задачи применение локальной памяти в общем случае становится невозможным ввиду большого размера матриц, которые используются для хранения результатов моделирования.

Не использовать buffer accessors

### Выравнивание адресов памяти

Данные могут быть записаны и прочитаны по произвольному адресу в памяти. Однако скорость доступа к ним может быть разной в силу особенностей чтения машинных слов процессорами. Выравнивание данных – один из способов избежать нежелательных задержек при обращении к памяти. Общая рекомендация заключается в выделении памяти по адресу, кратному величине машинного слова. Данная оптимизация была применена в ходе выполнения работы.

### Примитивы синхронизации

Примитивы синхронизации — это инструменты программирования, которые используются для синхронизации выполнения нескольких потоков или процессов в многопоточной среде. Они позволяют контролировать доступ к разделяемым ресурсам, таким как память, а также синхронизировать выполнение операций между потоками.

Существует несколько примитивов синхронизации:

1. Мьютекс (mutex) — это примитивы синхронизации, которые используются для защиты разделяемых ресурсов от одновременного доступа нескольких потоков. Является одним из самых медленных примитивов, так как нуждаются в обработке системного вызова в kernel space. Еще одним минусом является отсутствие поддержки на GPU.
2. Фьютекс (futex) — это примитивы синхронизации, которые используются для защиты разделяемых ресурсов от одновременного доступа нескольких потоков. В отличии от mutex, их обработка происходит в user space. Они так же не имеют поддержки на GPU.
3. Атомарные операции (atomic operations) — это операции, которые гарантируют атомарность выполнения в многопоточной среде. Атомарные операции позволяют выполнять операции чтения, записи и обновления разделяемых ресурсов без блокировки других потоков. Операции поддерживаются как на CPU, так и на GPU.
4. Барьеры (barrier) — это примитивы синхронизации, которые используются для синхронизации выполнения потоков. Барьеры гарантируют, что все потоки завершили выполнение определенной операции, прежде чем продолжить выполнение следующей операции. Операции поддерживаются как на CPU, так и на GPU.

Наиболее подходящим примитивом синхронизации из рассмотренных являются атомарные операции, так как имеют поддержку на CPU/GPU и позволяют вносить изменения к по конкретному адресу памяти.

### Минимизация взаимных блокировок

Использование примитива синхронизации предполагает упорядоченный доступ к одним и тем же ячейкам памяти для разных ядер, что приводит к простаиванию ядер, ожидающих запись. С увеличением количества вычислителей неизбежно растет время ожидания такого доступа к общему ресурсу, поэтому общий прирост производительности снижается. И наоборот, снижение количества вычислителей приведет к снижению времени ожидания на запись. Таким образом, рост времени ожидания доступа может быть снижен путем разделения областей памяти и балансировки вычислений на них.

### Применение типов пониженной точности

Во время процесса моделирования требуется высокая точность для просчета распространения света внутри тканей. Однако, при сохранении данных, не требуется такая высокая точность с учетом относительной большой погрешности разбиения сетки результирующей матрицы. С учетом этого утверждения было принято решение установить для выходных значений результирующей матрицы с плавающей запятой тип данных пониженной точности. Например, тип одинарной или половинной точности. Данная оптимизация снижает требуемый на объем оперативной памяти устройства, что позволяет в разы больше групп потоков. Изменение типа данных так же положительно влияет на скорость записи данных.

Наибольшего эффекта позволяет добиться тип половинной точности, но такой тип данных не поддерживается атомиками. На основе этого ограничения было принято решение установить тип с одинарной точностью.

### Проблема дивергенции

Проблема дивергенции заключается в выполнении разных веток условного оператора внутри одного блока потоков графического процессора. Некоторые процедуры алгоритма моделирования были переписаны так, чтобы избежать ветвления.

### Использование sycl:: функций

## Результаты экспериментов

# Заключение

В данной работе изучено строение и особенности массивно-параллельных вычислений. Был проведен подробный обзор технологий Nvidia CUDA и Intel OneAPI. Рассмотрены основные принципы разработки ПО для решения задач в области распараллеливания программ, а также выполнена реализация практических задач на эту тему. Реализован параллельный алгоритм метода Монте Карло на языке DPC++ в задаче распространения света на GPU. Проведен ряд экспериментов с последующим анализом полученных результатов. Наибольший прирост производительности зафиксирован в параллельной версии метода Монте Карло, и составляет более 16 раз относительно последовательной версии, что говорит о значительном улучшении.

В дальнейшем планируется продолжение разработки с целью повышения скорости выполнения программы.

# Литература

1. Сандерс Д., Кэндрот Э.: Технология CUDA в примерах. Введение в программирование графических процессов – 2018 г. – 232 с.
2. Боресков А. В., Харламов А.: Основы работы с технологией CUDA – 2010 г. – 231с.
3. Intel OneAPI – официальный сайт: <https://www.intel.com/oneapi/overview.html>
4. Корняков К.В., Мееров И.Б., Сиднев А.А., Сысоев А.В., Шишков А.В. Инструменты параллельного программирования в системах с общей памятью: учебное пособие — изд-во ННГУ – 2010 г. — 201 с.
5. Wang L.V., Jacques S.L., Zheng L.Q. MCML – Monte Carlo modeling of light transport in multi-layered – 1995 г. – 186 с.
6. OpenCL – официальный сайт: <http://www.khronos.org/opencl/>
7. А.В. Горшков, М.Ю. Кириллин, В.П. Гергель: Улучшенный метод Монте-Карло для моделирования распространения зондирующего излучения в задачах оптической диффузионной спектроскопии – 2014 г. – 9 с.
8. J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner: Data Parallel C++ – Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL – 2021 г. – 548