**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 помощью стандарта 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

[Введение 3](#_Toc135000404)

[1. Постановка задачи 4](#_Toc135000405)

[2. Модель распространения света 5](#_Toc135000406)

[3. Обзор технологий параллельных вычислений 8](#_Toc135000407)

[3.1 Стандарт Data Parallel C++ 10](#_Toc135000408)

[3.2 План разработки ПО на DPC++ 11](#_Toc135000409)

[4. Последовательный алгоритм решения методом Монте Карло 12](#_Toc135000410)

[4.1 Реализация последовательного алгоритма на CPU 12](#_Toc135000411)

[4.2 Генерация псевдослучайных чисел 15](#_Toc135000412)

[Линейный конгруэнтный метод 16](#_Toc135000413)

[4.3 Тестирование и визуализация результатов моделирования 17](#_Toc135000414)

[Визуализатор выходных данных алгоритма моделирования 18](#_Toc135000415)

[4.4 Результаты экспериментов 20](#_Toc135000416)

[5. Параллельный алгоритм решения методом Монте Карло 21](#_Toc135000417)

[5.1 Архитектурное решение 21](#_Toc135000418)

[5.2 Генерация псевдослучайных чисел 24](#_Toc135000419)

[Мультипликативный конгруэнтный метод 24](#_Toc135000420)

[5.3 Программная оптимизация 25](#_Toc135000421)

[Подсистема памяти 25](#_Toc135000422)

[Доступ к памяти между устройствами 28](#_Toc135000423)

[Выравнивание адресов памяти 29](#_Toc135000424)

[Примитивы синхронизации 29](#_Toc135000425)

[Минимизация взаимных блокировок 30](#_Toc135000426)

[Применение типов пониженной точности 30](#_Toc135000427)

[5.4 Результаты экспериментов 32](#_Toc135000428)

[Заключение 34](#_Toc135000429)

[Литература 35](#_Toc135000430)

# 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 рамках настоящей диссертации необходимо разработать программное обеспечение для решения задачи физического моделирования распространения фотонов света для плоскопараллельной геометрии многослойной среды, а также выполнить перенос параллельной 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 полученных реализаций для центральных и графических процессоров.

# Модель распространения света

Метод Монте Карло выполняет многократное повторение случайных независимых испытаний на основе полученной математической модели. Статистические данные накапливаются, чтобы сделать вывод о вероятностных характеристиках исследуемого объекта.

Каждый слой среды, в которой распространяются фотоны, 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 – вектором , где каждый элемент тройка определяет направляющий косинус:

где – орт направления скорости, – орты координатных осей.

Отражение фотона на границе разных 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.

# Обзор технологий параллельных вычислений

Наиболее принципиальным методом сокращения времени вычислений 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 его характеристиками, а также поддерживаемыми 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. Data Parallel C++ – это открытый стандарт параллельного программирования, который разрабатывается в рамках технологии OneAPI, созданной компанией Intel. DPC++ создан на основе открытого стандарта SYCL, внося в него упрощения в применении и добавляя новые возможности. Как и SYCL, DPC++ предоставляет возможность разработки параллельных программ для 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 | 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.

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. OneAPI 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;

OneAPI предоставляет легкий в освоении и изучении собственный язык программирования Data Parallel C++ и отдельный компилятор для него. Его 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.

Стандартом поддерживается большинство вендоров, и это дает возможность разрабатывать решение, независимое от конкретного производителя. Программная модель гарантирует переносимость кода между устройствами, но не гарантирует одинаковую производительность на разных устройствах. Другим преимуществом является возможность не только разрабатывать приложения для разнородных устройств, но и проводить сеансы отладки на CPU, что позволяет останавливать выполнение программы и вникать в суть проблем значительно быстрее.

## Этапы разработки ПО на DPC++

Разработка параллельной реализации на специализированных устройствах разделяется на два главных этапа:

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 таким образом, чтобы вычисления проходили наиболее оптимально. Эта часть программы разрабатывается для управляющего устройства, которое выполняет оркестрацию задач. Зачастую таким устройством выступает СPU, т.к. именно оно управляет потоком задач для GPU. Код, выполняющийся на CPU, называют host domain code.
2. **Написание параллельного кода.**   
   Целью этой части является проработка алгоритмов, которые будут выполняться параллельно. Необходимо определить операции над предоставленными данными, синхронизировать доступ на чтение и запись между единицами параллельного исполнения. Часть программы, называемая kernel, выполняется на устройстве. Код, исполняемый на целевом устройстве (CPU, GPU, FPGA), называют device domain code.

Возможность балансировать нагрузку между разными устройствами является важным преимуществом. Так, например, код может одновременно выполняться и на CPU, и на GPU, 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.

Существуют ограничения, которые накладываются на код, разработанный для kernel DPC++:

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

# Последовательный алгоритм решения методом Монте Карло

Реализация последовательного алгоритма стала первым этапом в данной работе. Он включает в себя перевод математической модели 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. Так, например, более высокоуровневые языки имеют несколько слоев абстракции, каждый из которых вносит дополнительные проверки и преобразования для общих случаев. Ситуация осложняется тем, что разные устройства могут иметь разные зависимости, в общем случае, учесть влияние которых достаточно сложно.

## 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 };

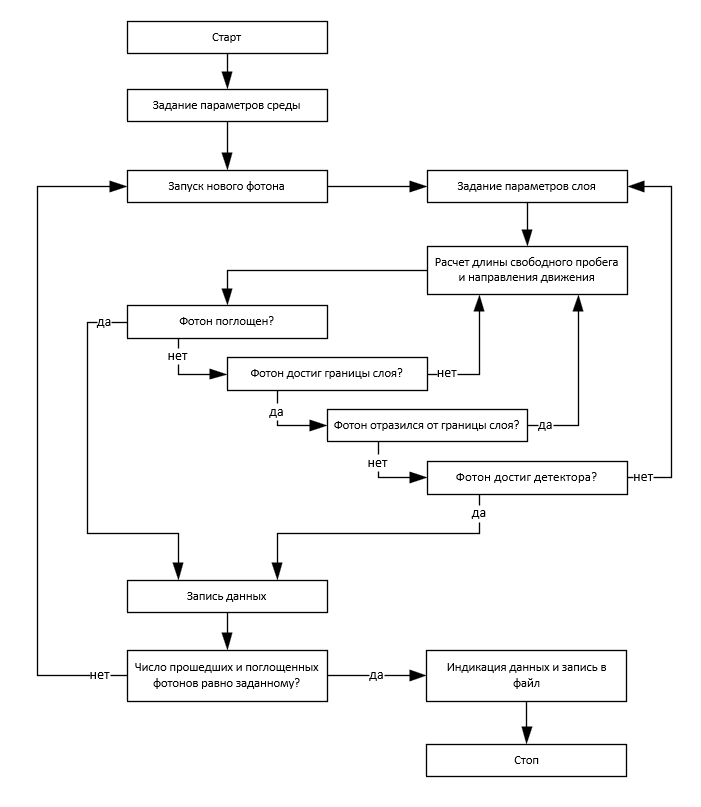
const InputStruct\* input;

}

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 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 приведет 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 значение вычисляется на основе результатов предыдущего значения по следующей формуле:

где:

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

Модуль определяет максимальную длину последовательности. Выбор наиболее подходящего значения модуля 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 благодаря менее трудоемки операциям. 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 последовательности 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. Оно необходимо для обеспечения должного качества и надежности программ. Ошибки могут возникать в разных частях программы, и если они не будут обнаружены и исправлены, то они отрицательно повлияют на корректность результатов.

Так же тестирование является неотъемлемым шагом при оптимизации программ. Выявленные трудоемкие операции могут быть оптимизированы с повышением производительности в частных случаях, но с допущением ошибок в общем случае. Как правило, ошибки, возникшие во время оптимизации, не воспроизводятся в тех сценариях, над улучшением которых проводятся оптимизации. Как следствие, такие ошибки остаются незамеченными без проведения комплексного тестирования.

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 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 реализации был разработан модуль для визуализации карты интенсивности фотонов, так как он позволяет представить результаты работы алгоритма в наглядном и понятном виде.

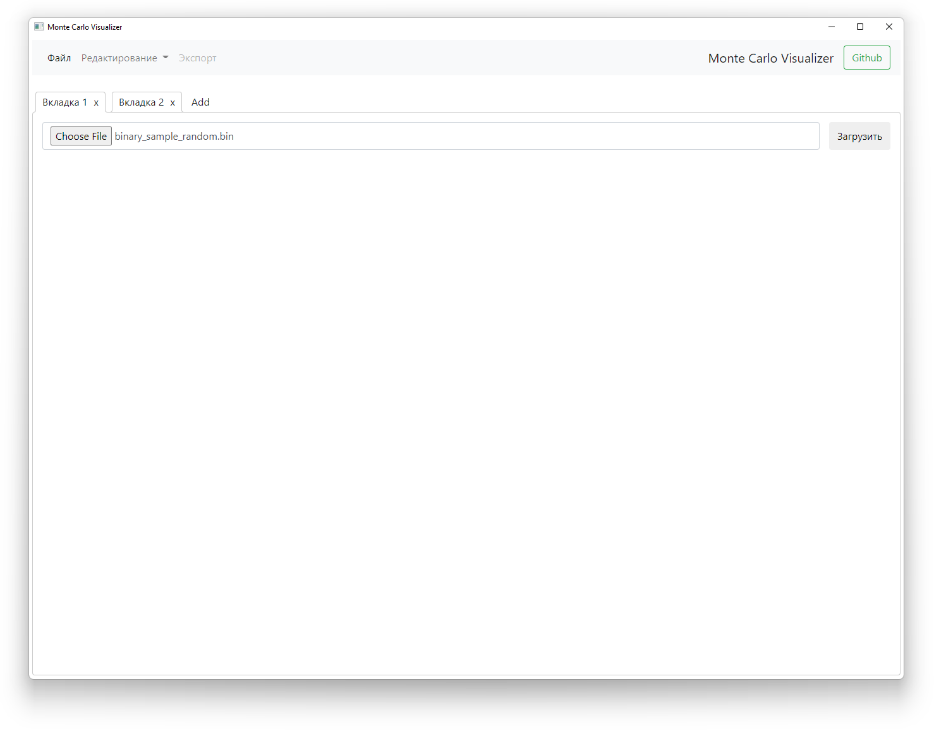
Визуализатор разработан на языке C++, т.к. он тесно связан с основной реализацией и использует те же самые заголовки и структуры данных для своей работы.

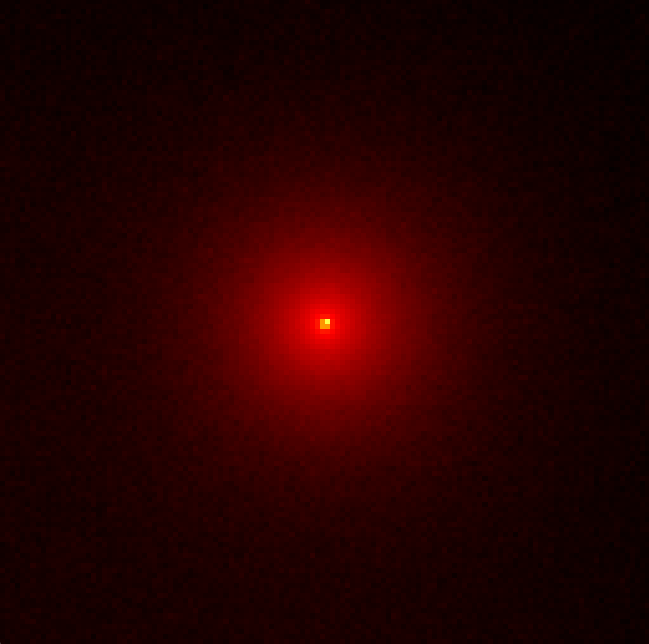
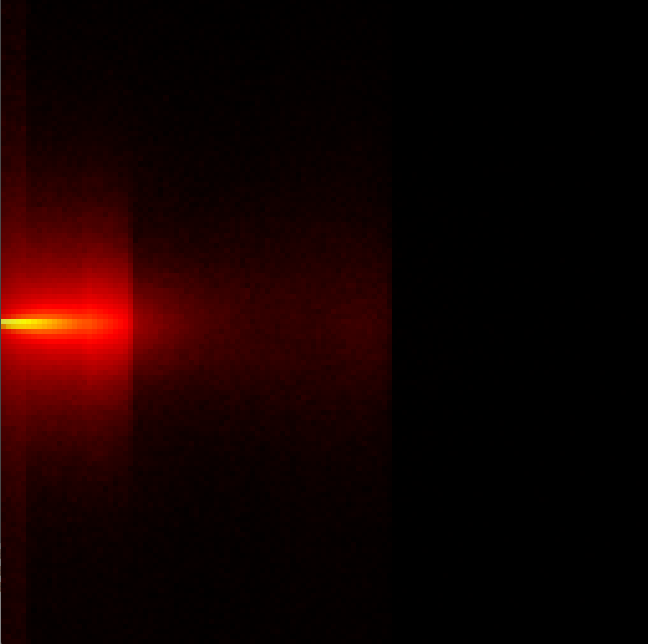
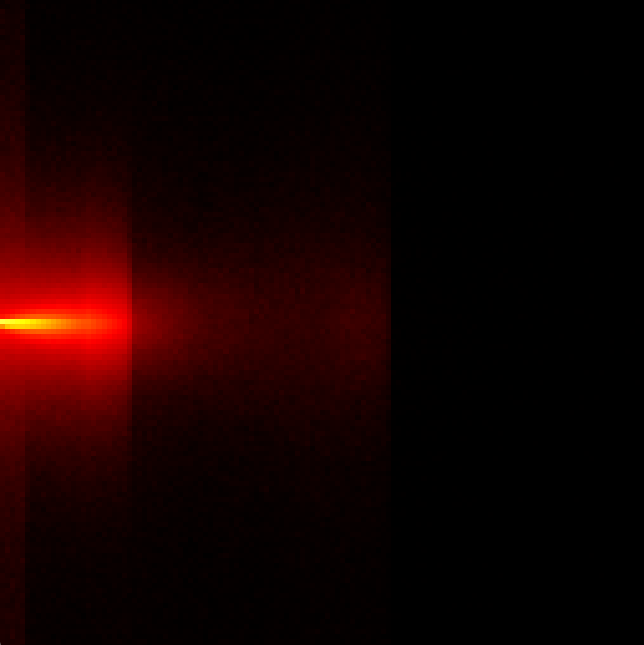
В качестве инструмента рисования используется библиотека OpenGL для C++, которая является достаточно простой в использовании и одной из наиболее популярных библиотек для рисования графики в реальном времени.

Визуализатор использует Chromium Embedded C++ Framework для создания графического интерфейса. В основе данного фреймворка используется движок Chromium для рендеринга веб страниц. Это позволяет воспользоваться HTML разметкой для создания приложений.

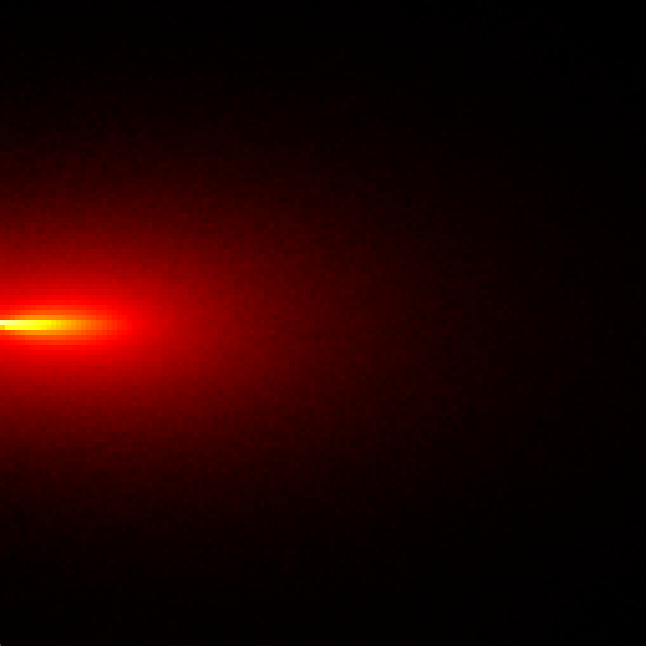
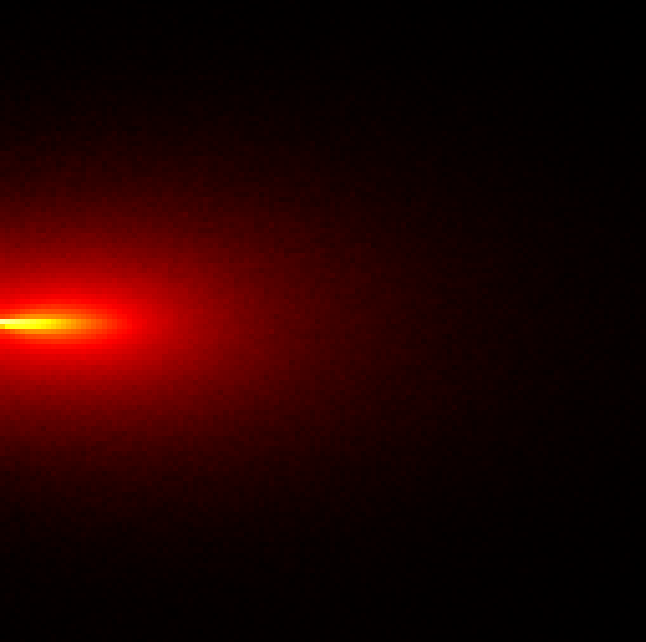
Во время запуска визуализатора пользователю необходимо выбрать файл для визуализации. После загрузки файла трехмерная матрица интенсивности визуализируется в трех разных координатных плоскостях Oxy, Oyz, Oxz.

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 с разными значениями параметров слоёв.

На продемонстрированных рисунках можно наблюдать рассеивание света непосредственно от источника излучения в трех плоскостях с одинаковыми значениями параметров слоёв.

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

Ниже приведены результаты экспериментов разработки последовательного алгоритма для CPU.

Операционная система: Windows 11, 64-bit.

Процессор: Intel Core i9 9900K 4.48 GHz.

Количество ядер в процессоре: 16.

Количество процессоров: 1.

Объем оперативной памяти: 24GB.

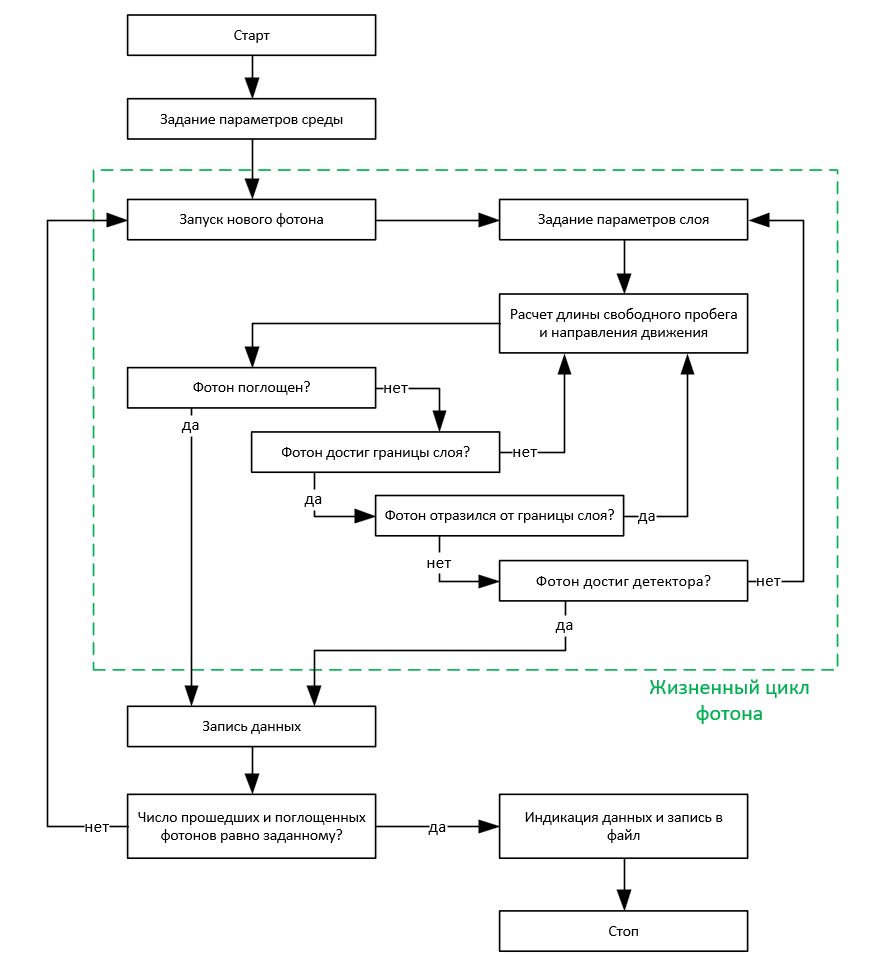
|  |  |  |  |
| --- | --- | --- | --- |
| Вычислительное устройство | Число фотонов | | |
| 100 000 | 1 000 000 | 10 000 000 |
| CPU Intel i9 9900K (1 ядро) | 5942 мс | 59426 мс | 594260 мс |

# 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:

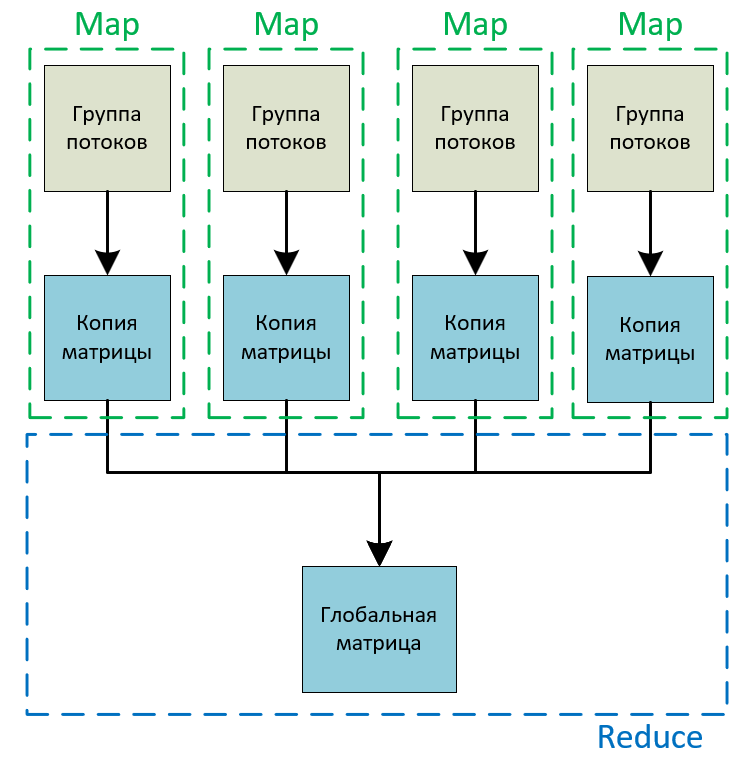


Каждое испытание проводится независимо друг от друга, таким образом они могут быть выполнены параллельно, с учетом, что все данные будут синхронизировано сохранены в памяти устройства. В качестве сохраняемых данных записывается потерянный вес фотона. Учитывая особенность генерации случайных данных, невозможно 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.

## 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

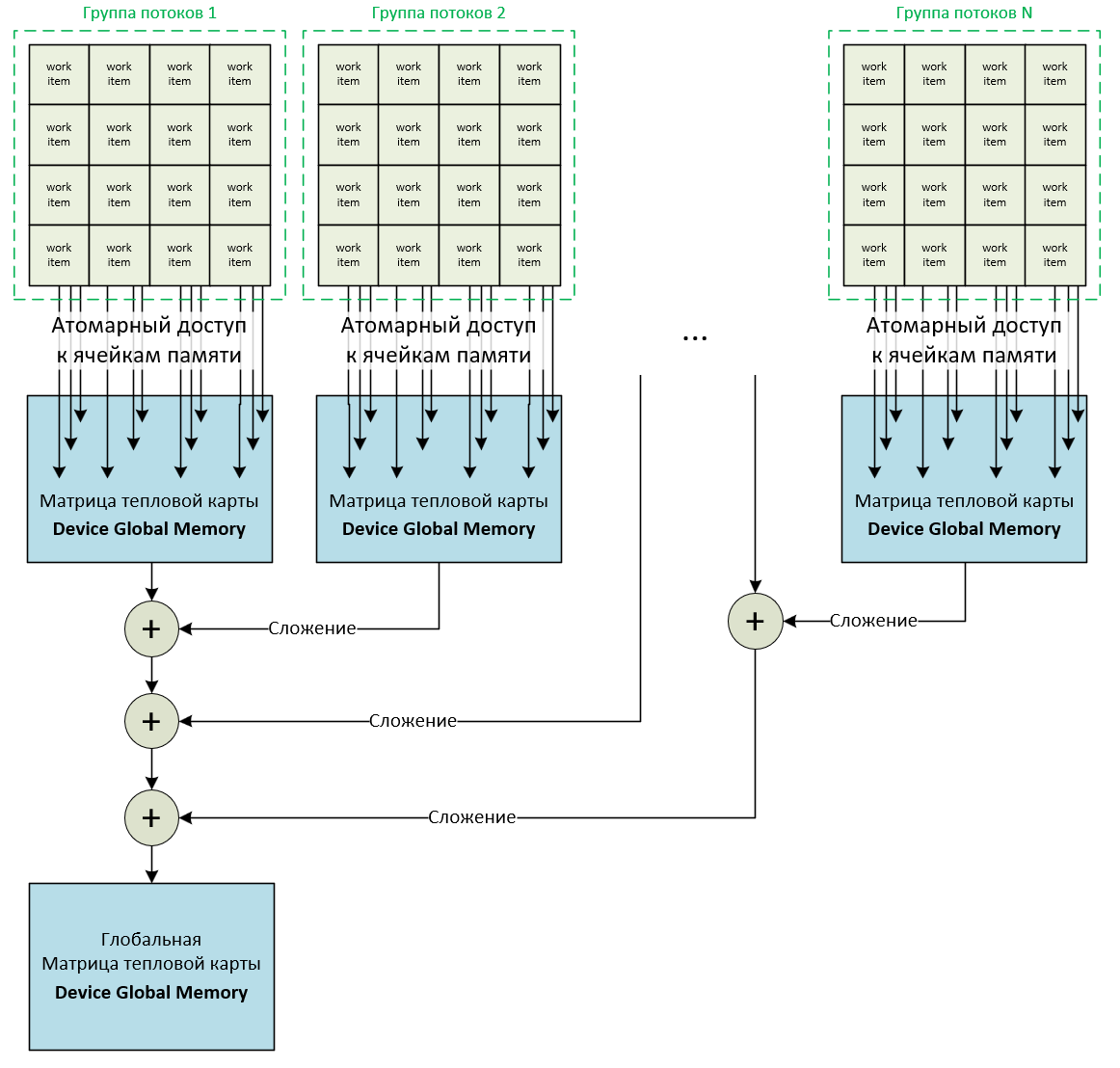
Проблема многих многопоточных реализаций заключается в синхронизированном доступе к общим ресурсам, например, таким как память. В частности, одной из таких проблем является одновременный доступ. Data race т.н. состояние гонки – это ошибка проектирования многопоточной системы, при которой нарушен синхронизированный 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:

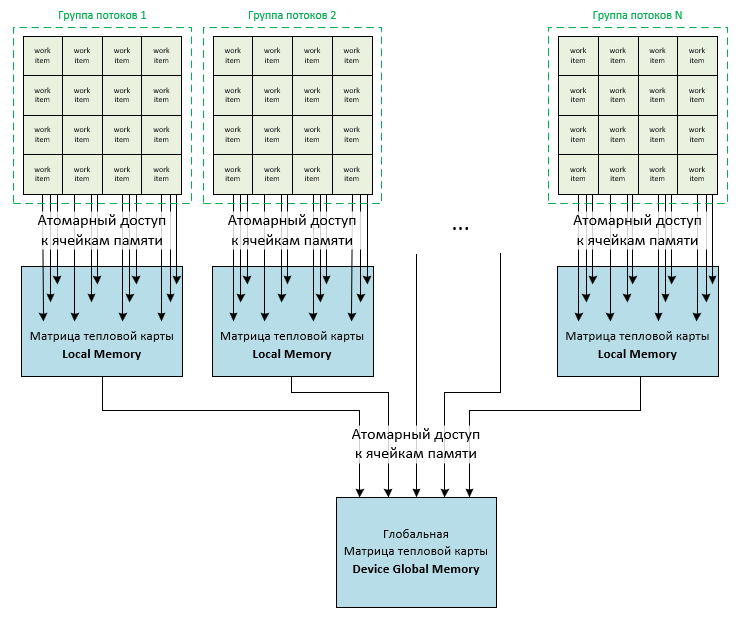
1. Среднее время ожидания доступа для записи/чтения определяется размером группы.
2. Количество групп вычислителей не ограничено, хорошая масштабируемость.

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.

### 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.

Существует три абстракции для управления памятью: унифицированная общая память (USM), буферы и изображения. USM — это подход на основе указателей, который должен быть знаком программистам на C/C++. Одним из преимуществ USM является более простая интеграция с существующим кодом C++, работающим с указателями.

Буферы, представленные классом шаблона буфера, описывают одно-, двух- или трехмерные массивы:

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

sycl::accessor<T, dimensions, sycl::access::mode, sycl::access::target::>

Они обеспечивают абстрактное представление памяти, к которому можно получить доступ как на хосте, так и на устройстве. Программа не имеет прямого доступа к буферам, а вместо этого используется через объекты доступа. Элементы объекта буфера могут иметь скалярный тип данных (например, int, float или double), векторный тип данных или определенный пользователем класс или структуру.

Другим способом управления данными является USM. USM — это подход на основе указателей, аналогично языкам на C и C++, использующим malloc или new для размещения данных. USM упрощает разработку при переносе существующего кода C/C++, в котором интенсивно используются указатели. Устройства, поддерживающие USM, поддерживают единое виртуальное адресное пространство. Наличие единого виртуального адресного пространства означает, что любое значение указателя, возвращаемое процедурой распределения USM на хосте, будет действительным значением указателя на устройстве. Разработчику не нужно вручную переводить указатель хоста, чтобы получить «версию устройства» — оно имеет одно и то же значение указателя и на хосте, и на устройстве.

Неявное перемещение данных, как правило, проще для начала, потому что все перемещение данных обрабатывается средствами языка DPC++, что позволяет сосредоточиться на выражении вычислений. Однако, когда в приоритете находится производительность, это требует повышенного контроля за ресурсами, а значит применение явных операций выделения, копирования и освобождения памяти. Так было принято решение отказаться от буфферов в пользу USM.

### 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

Современные центральные процессоры умеют достаточно быстро работать с 64 битными вещественными типами данных, переход на тип одинарной точности не существенно влияет на прирост производительности. Однако, архитектуры центрального и графического процессоров во многом различны, т.к. они оптимизированы под разные задачи.

Большинство задач GPU связаны с отрисовкой графики и эта категория задач не требует такой высокой точности. Выполнение операций с 64 битными типами и поддерживаются, но скорость их выполнения значительно снижена. В рамках оптимизации вычислений под графический процессор это дает значительное обоснование для совершения перехода на тип одинарной точности. С другой стороны, снижение точности может отрицательно сказаться на результатах моделирования. Однако погрешность, связанная с точностью вычислений разных типов с плавающей запятой, не выходит за рамки допустимой погрешности, которая определяется размером сетки матрицы. Предположение экспериментально подтверждается с помощью визуализатора.

Так же не малыми преимуществом при переходе на тип пониженной точности является кратное снижение используемого объема оперативной памяти устройства, что позволяет в разы больше групп потоков.

Графические процессоры имеют поддержку разных типов пониженной точности для чисел с плавающей запятой. Например, тип половинной или одинарной точности. Наибольшего эффекта позволяет добиться 16 битный тип половинной точности, но в таком случае эксклюзивный доступ к ячейкам памяти (атомарный доступ) становится невозможным в силу ограничений стандарта. На основе этого ограничения было принято решение установить 32 битный тип с одинарной точностью.

Также вместе с этим появляется возможность использовать менее точные математические операции, которые можно задействовать с помощью опций компилятора на этапе сборки приложения.

## Результаты экспериментов

Ниже приведены результаты экспериментов разработки последовательного алгоритма для CPU и GPU до и после оптимизации.

Операционная система: Windows 11, 64-bit.

Процессор: Intel Core i9 9900K 4.48 GHz.

Количество ядер в процессоре: 16.

Количество процессоров: 1.

Объем оперативной памяти: 24GB.

Графический ускоритель: Intel UHD Graphics 630.

Графический ускоритель: Nvidia RTX 3060 12GB.

|  |  |  |  |
| --- | --- | --- | --- |
| Вычислительное устройство | Число фотонов | | |
| 100 000 | 1 000 000 | 10 000 000 |
| CPU Intel i9 9900K (1 ядро) | 5 942 мс | 5 9426 мс | 594 260 мс |
| CPU Intel i9 9900K  (8 ядер) | 775 мс | 7 758 мс | 77 588 мс |
| CPU Intel i9 9900K (16 ядер) | 524 мс | 5 242 мс | 52 423 мс |
| GPU Intel UHD Graphics 630 | 27 809 мс | 278 094 мс | 2 780 944 мс |
| GPU NVIDIA RTX 3060 | 13 067 мс | 130 674 мс | 1 306 743 мс |

После выполнения вышеописанных оптимизаций становится разумным увеличить число моделируемых траекторий фотонов.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Вычислительное устройство | Число фотонов | | | | |
| 100 000 | 1 000 000 | 10 000 000 | 100 000 000 | 1 000 000 000 |
| CPU Intel i9 9900K (1 ядро) | 5 679 мс | 5 6790 мс | 567 900 мс | 5 667 642 мс | 56 563 067 мс |
| CPU Intel i9 9900K  (8 ядер) | 3 058 мс | 30 580 мс | 305 806 мс | 2 963 260 мс | 28 713 989 мс |
| CPU Intel i9 9900K (16 ядер) | 492 мс | 4 923 мс | 49 231 мс | 475 165 мс | 5 232 400 мс |
| GPU Intel UHD Graphics 630 | 405 мс | 4 054 мс | 40 546 мс | 400 999 мс | 3 965 880 мс |
| GPU NVIDIA RTX 3060 | 91 мс | 919 мс | 9 192 мс | 46 053 мс | 448 356 мс |

# Заключение

# 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~~](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/~~](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~~
9. ~~Ampere Tuning Guide~~
10. ~~CUDA BEST PRACTICE~~