МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ  
Федеральное государственное автономное образовательное учреждение  
высшего образования

**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского»  
(ННГУ)  
  
Институт информационных технологий, математики и механики**

**Кафедра: Математического обеспечения и   
суперкомпьютерных технологий**  
Направление подготовки: «Прикладная математика и информатика»  
Магистерская программа: «Вычислительные методы и суперкомпьютерные технологии»

**МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ**

**Тема:**

**«Монте Карло моделирование распространения света   
на гетерогенных архитектурах»**

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

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

# Оглавление

[Введение 3](#_Toc137068813)

[1. Постановка задачи 5](#_Toc137068814)

[2. Модель распространения света 6](#_Toc137068815)

[3. Обзор технологий параллельных вычислений 10](#_Toc137068816)

[3.1 Стандарт Data Parallel C++ 12](#_Toc137068817)

[3.2 Этапы разработки ПО на DPC++ 14](#_Toc137068818)

[4. Последовательный алгоритм решения методом Монте Карло 16](#_Toc137068819)

[4.1 Реализация последовательного алгоритма на CPU 16](#_Toc137068820)

[4.2 Генерация псевдослучайных чисел 20](#_Toc137068821)

[Линейный конгруэнтный метод 20](#_Toc137068822)

[4.3 Тестирование и визуализация результатов моделирования 21](#_Toc137068823)

[Визуализатор выходных данных алгоритма моделирования 22](#_Toc137068824)

[4.4 Результаты экспериментов 26](#_Toc137068825)

[5. Параллельный алгоритм решения методом Монте Карло 27](#_Toc137068826)

[5.1 Архитектурное решение 27](#_Toc137068827)

[5.2 Генерация псевдослучайных чисел 30](#_Toc137068828)

[Мультипликативный конгруэнтный метод 30](#_Toc137068829)

[5.3 Программная оптимизация 31](#_Toc137068830)

[Подсистема памяти 31](#_Toc137068831)

[Доступ к памяти между устройствами 34](#_Toc137068832)

[Выравнивание адресов памяти 35](#_Toc137068833)

[Примитивы синхронизации 35](#_Toc137068834)

[Минимизация взаимных блокировок 36](#_Toc137068835)

[Применение типов пониженной точности 37](#_Toc137068836)

[5.4 Результаты экспериментов 39](#_Toc137068837)

[Заключение 42](#_Toc137068838)

[Список литературы 43](#_Toc137068839)

[Приложение 1. Программный код алгоритма моделирования 44](#_Toc137068840)

[Приложение 2. Программный код распределения параллельных вычислений 57](#_Toc137068841)

# Введение

Одними из наиболее перспективных направлений в биомедицине являются методы оптической диагностики. Они основаны на использовании света в видимом и инфракрасном диапазонах для зондирования биологических объектов, с помощью анализа рассеянного излучения. 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 развитие методов оптической диагностики, что требует соответствующего программного обеспечения для подготовки большого количества данных в сфере машинного обучения.

Данные оптической диагностики могут быть получены путем моделирования распространения лучей 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 применяемым методом для решения задачи физического моделирования распространения света внутри биологических тканей является метод Монте-Карло [6]. Данная реализация часто применяется для решения задач многослойной плоскопараллельной геометрии [1, 2, 4] при моделировании работы приборов оптической биомедицинской диагностики. Основная идея этого метода заключается в многократном повторении случайных независимых испытаний. Однако вычисление всех испытаний может занять достаточно много времени.

В данной магистерской диссертации будет рассмотрено моделирование процесса распространения света методом Монте-Карло на графическом ускорителе с применением технологии Intel OneAPI [3]. Будет проведен анализ возможностей и ограничений вычислительных систем с использованием графических ускорителей для данного метода, а также исследованы способы оптимизации вычислений на GPU [7]. 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 [1, 2] 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, и провести сравнительный анализ результатов полученных реализаций для решения поставленной задачи с применением метода Монте-Карло [6].

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 центрального и графического процессоров.

# Модель распространения света

Метод Монте Карло выполняет многократное повторение случайных независимых испытаний на основе полученной математической модели. Результаты каждого испытания сохраняются и обобщаются. Накопленные статистические данные используются, чтобы сделать вывод о вероятностных характеристиках исследуемого объекта.

Каждый слой сильно рассеивающей среды [1], 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 вектора определяет направляющий косинус:

где – орт направления скорости, – орты координатных осей.

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

где и – соответственно углы падения и преломления луча, и – показатели преломления соответствующих сред. Угол преломления определяется в соответствии с законом Снеллиуса [6]:

Рассмотрим шаг распространения фотона, за который принимается один акт рассеяния. Пучок фотонов начинает движение из начальной точки в заданном направлении от источника излучения. В соответствии с заданными параметрами определяются угловое и пространственное распределение интенсивности. Далее происходит расчет длины свободного пробега, которая рассчитывается исходя из параметров текущего слоя. 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 каждый раз при рассеянии и определяется фазовой функцией рассеяния:

Величина считается как равномерно распределенная на отрезке . Это связано с тем, что рассеиватели в общем случае считаются сферически симметричными. В соответствии с заданной фазовой функцией единичного рассеивателя рассчитывается угол .

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.

Каждый акт рассеяния сопровождается поглощением. Каждый фотон обладает начальным весом (задается для повышения статистики), который постепенно уменьшается в соответствии со следующей формулой:

где – текущий вес фотона.

Обработка каждого акта рассеяния считается завершенным только после вычисления новых координат фотона. Затем последовательность действий повторяется, вычисляются новые направляющие косинусы вектора скорости, новая длина свободного пробега.

# Обзор технологий параллельных вычислений

Наиболее принципиальным методом сокращения времени вычислений является параллелизация вычислений [5]. Этот способ используется для ускорения выполнения сложных задач, которые могут быть разбиты на более мелкие подзадачи, выполняемые параллельно. Такие вычисления могут выполняться не только на ядрах одного центрального процессора общего назначения (CPU, Central Processing Unit), но и параллельно в кластерах, связанных сетевым интерфейсом. Такие системы были достаточно доступными для параллельных расчетов на протяжении долгого времени, но оставались достаточно дорогими и сложными в использовании [7]. 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 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 [8], которая позволяет использовать графические процессоры Nvidia для параллельных вычислений.
3. AMD HIP API – проприетарная технология компании AMD [12], которая позволяет использовать графические процессоры AMD для параллельных вычислений.
4. OpenCL – открытая технология некоммерческого консорциума Khronos Group [11], 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 разработки ПО для программирования параллельных вычислений, который разрабатывается компанией Intel в рамках технологии OneAPI [3]. 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 ускорителей.

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Семейства вычислительных устройств | Стандарты и технологии | | | | |
| OpenMP | CUDA | AMD HIP | OpenCL | DPC++ |
| CPU | + | – | – | + | + |
| GPU AMD | – | – | + | + | + |
| GPU Nvidia | – | + | – | + | + |
| GPU Intel Arc | – | – | – | + | + |

Таблица 1. Поддержка семейств вычислительных устройств для разных технологий и стандартов.

## Стандарт Data Parallel C++

Открытый стандарт OneAPI базируется на ряде других открытых стандартов, где наибольший вклад сделан OpenCL, поддерживаемый некоммерческим консорциумом Khronos Group [11]. Унифицированная программная модель, которая основана на открытых стандартах, 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. Теперь они могут быть использованными на разных устройствах и сохранить высокую производительность. OneAPI включает в себя следующие наборы:

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

OneAPI предоставляет легкий в освоении язык программирования Data Parallel C++ [3] 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 относительно малую когнитивную сложность восприятия кода.

Стандарт поддерживает совместимость с большей частью 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 вычислений, 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 программирование математической модели и 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 [5].

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. Анализ реализации осложняется тем, что разные устройства могут иметь разные зависимости для более высокоуровневых языков. В общем случае, учесть их совокупное влияние 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

{

data\_type\_t z0, z1;  
data\_type\_t n;  
data\_type\_t mua;  
data\_type\_t mus;  
data\_type\_t anisotropy;  
data\_type\_t cos\_crit0;  
data\_type\_t 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

{

size\_t num\_photons;

data\_type\_t Wth;

data\_type\_t dz;

size\_t nz;

size\_t num\_layers;

size\_t Nx;

size\_t Ny;

size\_t Nz;

size\_t 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 №1.

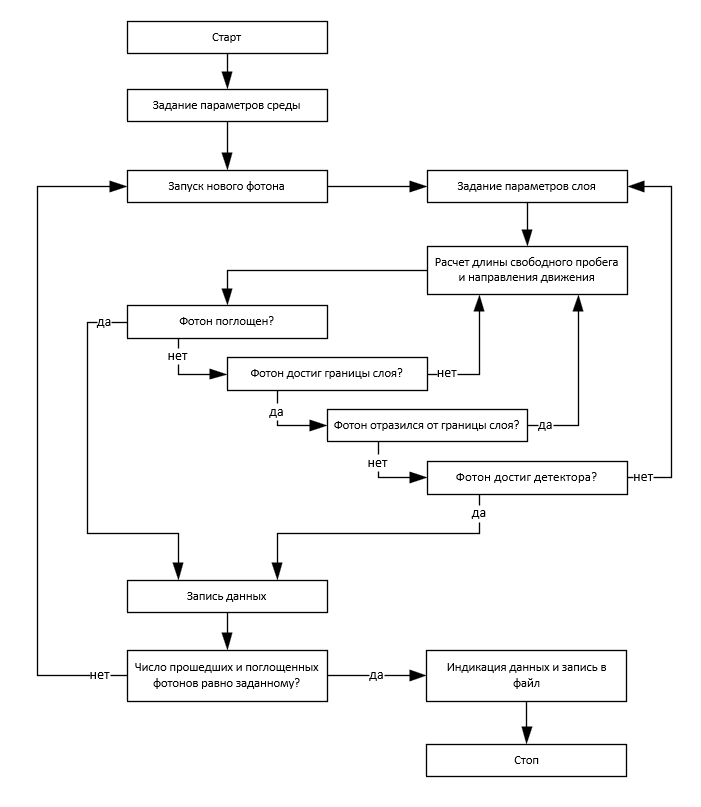


Рис. 1. Диаграмма выполнения процедур в алгоритме моделирования.

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 сопоставляются 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 накладывается математической моделью 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 псевдослучайное число 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 трудоемким 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 достижения должного качества и надежности программ, благодаря своевременному исправлению ошибок. 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 [14]. 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 наиболее 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 [13] для C++, которая является достаточно простой и удобной в использовании. Это одна из наиболее популярных библиотек для рисования графики в реальном времени, поэтому для нее можно найти избыточное количество примеров и документации.

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 графического интерфейса приложений.

Во время запуска визуализатора пользователю необходимо выбрать файл для визуализации (рисунок №2). После загрузки файла трехмерная матрица интенсивности визуализируется в трех разных координатных плоскостях Oxy, Oyz, Oxz. Для трехмерной матрицы фиксируется значение одной из координат (x, y, z), таким образом выполняется переход к двумерной матрице.

Каждая точка полученной двумерной матрицы однозначно соответствует пикселям двумерного изображения. Интенсивность фотонов определяется цветом и оттенком пикселей. Так, наиболее яркие участки соответствуют наиболее интенсивным участкам карты, т.е. наибольшему суммарному количеству потерянного веса в этой точке пространства.

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

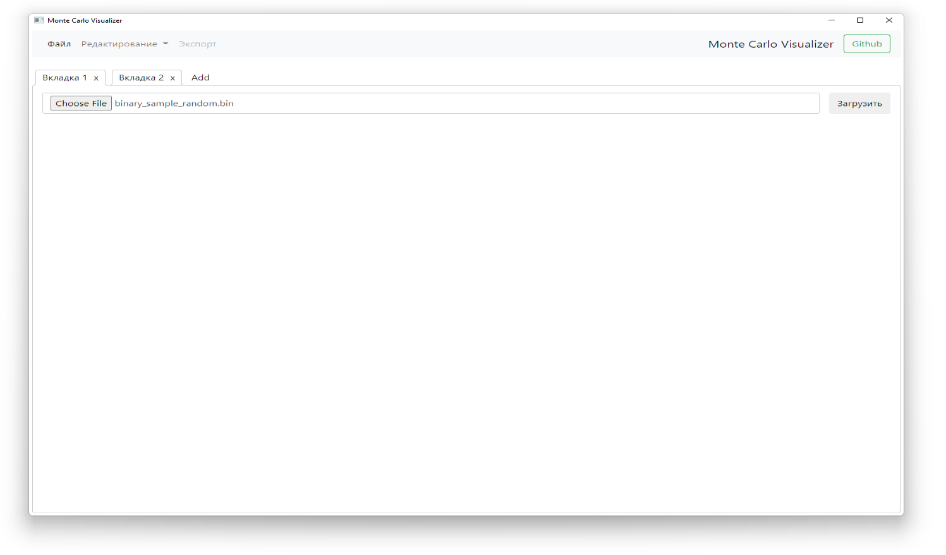


Рис. 2. Графический интерфейс пользователя.

Ниже представлены визуализации рассеянного излучения для многослойных сред с плоскопараллельной геометрией в трех плоскостях. На рисунке №3 можно наблюдать результат моделирования распространения фотонов внутри многослойной среды с разными характеристиками слоёв.

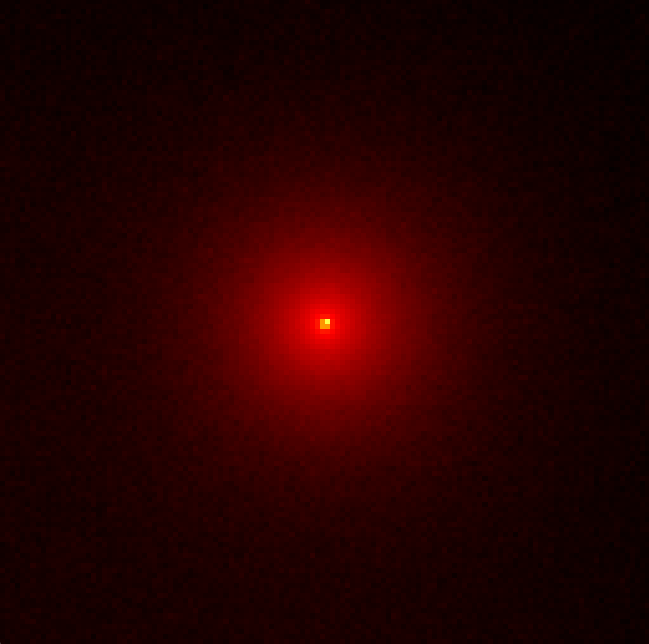
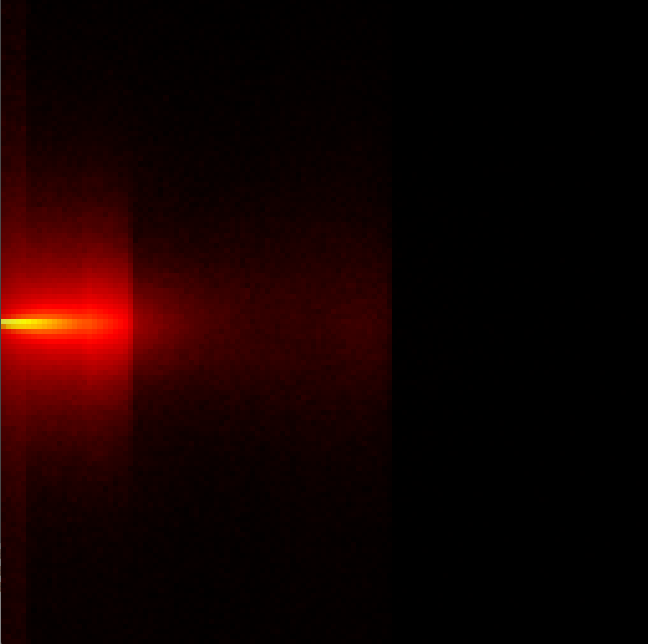
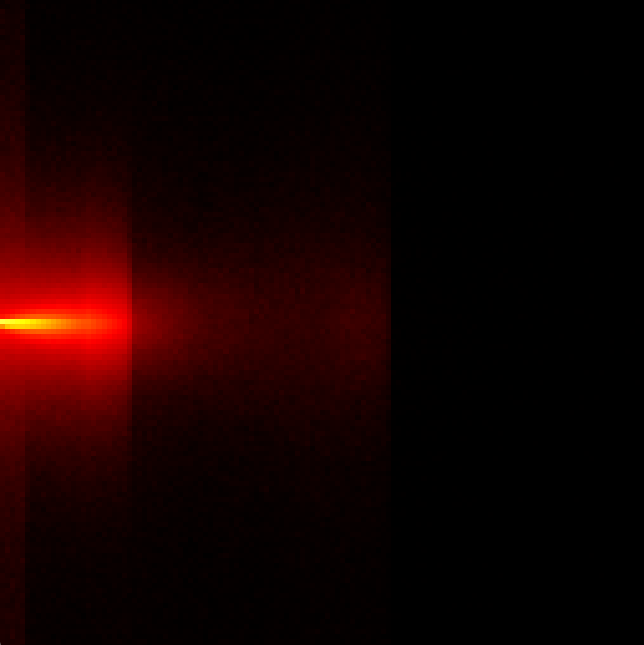
  

Рис 3. Визуализация результатов моделирования для многослойной среды с разными характеристиками слоя.

На рисунке №4 можно наблюдать результат моделирования распространения фотонов внутри многослойной среды с одинаковыми характеристиками слоёв.

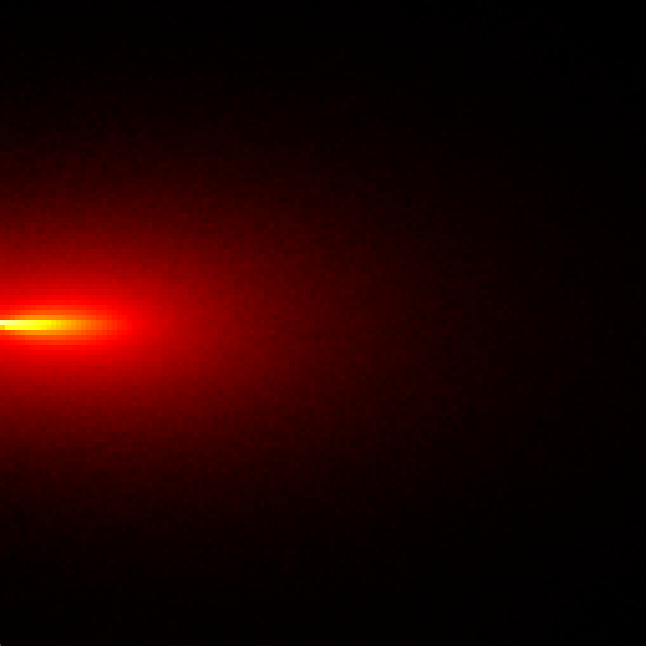
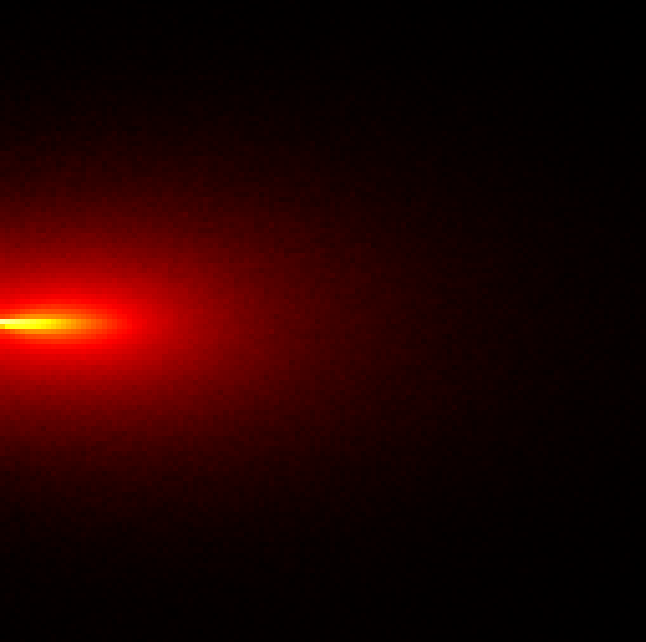
  

Рис. 4. Визуализация результатов моделирования для многослойной среды с одинаковыми характеристиками слоя.

## Результаты экспериментов

Ниже приведены результаты экспериментов разработки последовательного алгоритма для CPU.

Характеристики испытательного стенда:

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 | 15 090 мс | 127 003 мс | 1 415 620 мс |

Таблица 2. Время выполнения последовательной реализации для разного количества фотонов.

Рис. 5. График зависимости времени выполнения последовательной реализации от количества фотонов.

Так, на рисунке №5, можно заметить, что время моделирования испытания напрямую зависит от количества фотонов.

# 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 [6] предполагает, что каждое испытание проводится независимо. Эту особенность можно легко увидеть на рисунке №6, где схематично показан жизненный цикл фотона.

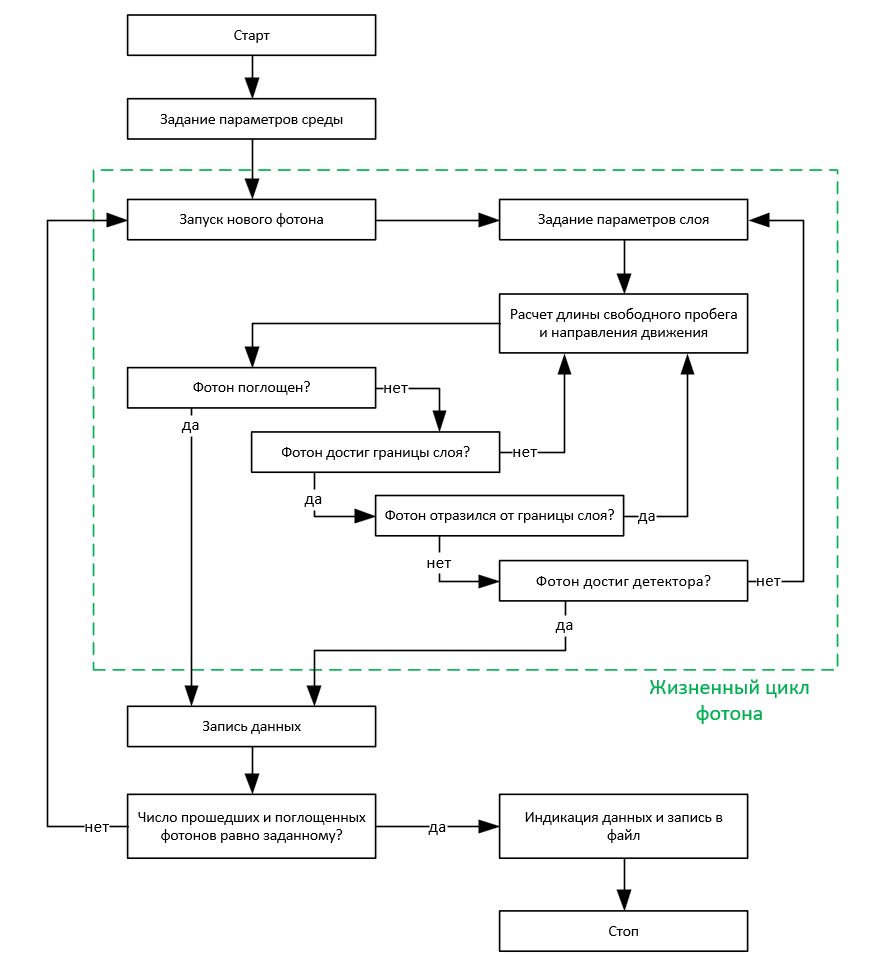


Рис. 6. Жизненный цикл фотона на диаграмме вызовов процедур алгоритма моделирования.

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 применения примитива многопоточной синхронизации для эксклюзивного доступа на чтение и запись.

Использование примитива синхронизации предполагает упорядоченный доступ к одним и тем же ячейкам памяти, что приводит к простаиванию потоков, ожидающих запись. С увеличением количества параллельных единиц вычисления неизбежно растет время ожидания доступа к общему ресурсу, поэтому общий прирост производительности оказывается меньше. 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 [10]. На рисунке №7 схематично описана эта концепция. Обработка большого объема данных разбивается на два этапа: Map и Reduce.

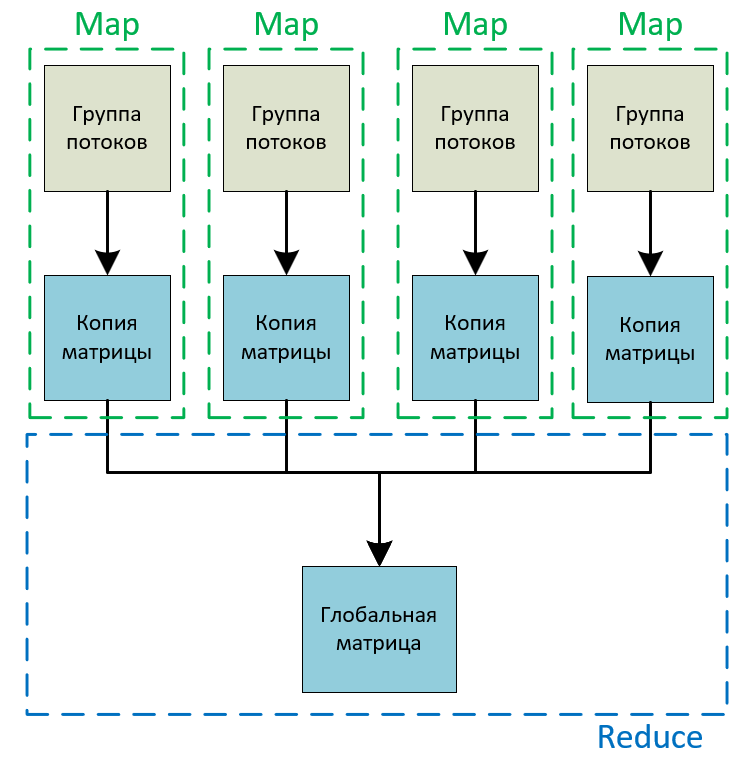


Рис. 7. Графическая интерпретация концепции MapReduce.

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 только своей 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 вычислений является высокая степень параллелизма, позволяющая значительно повысить производительность по сравнению с процессором общего назначения [5]. Поэтому для получения результата необходимо разработать параллельную реализацию алгоритма, учитывающую архитектурные особенности GPU [7].

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 так же называемое состоянием гонки – это ошибка проектирования многопоточной системы, при которой нарушен синхронизированный доступ к общим ресурсам, от которых зависят результаты вычислений [5]. 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, доступные для управления DPC++ [3]:

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 общей глобальной памяти устройства представлена на рисунке №8.

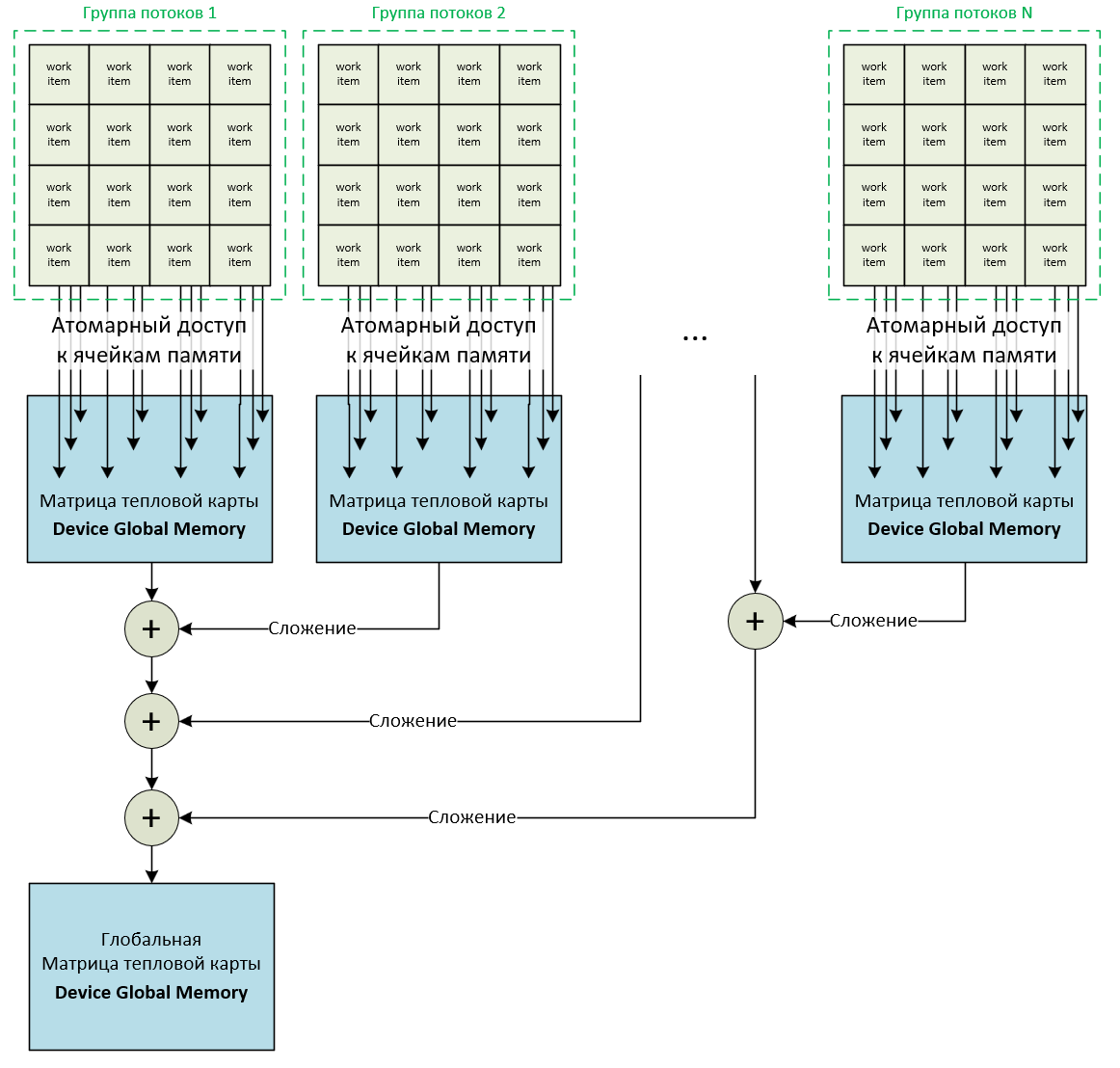


Рис. 8. Иллюстрация операций атомарного доступа к памяти внутри групп потоков.

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 можно отнести:

1. Повышенное 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:

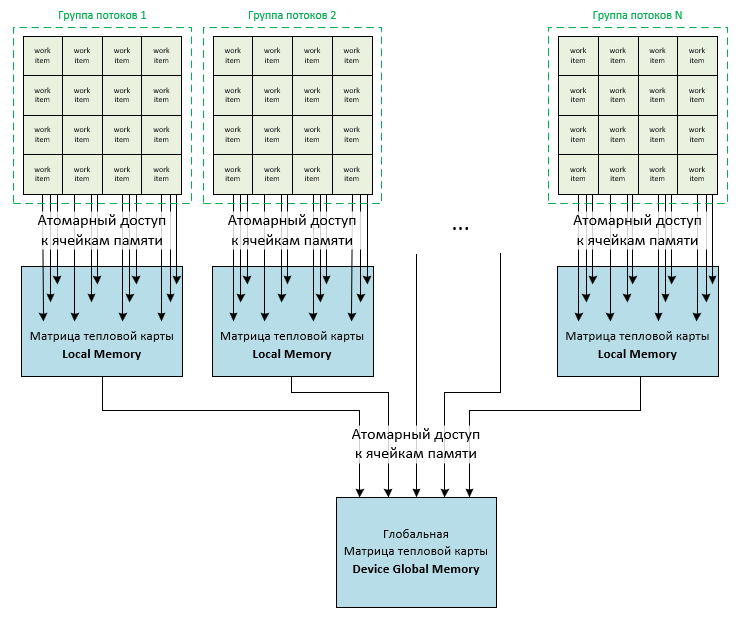


Рис. 9. Иллюстрация операций атомарного доступа к общей памяти между группами.

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. В DPC++ существует три абстракции для управления памятью: унифицированная общая память (USM), буферы и изображения [3].

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.

Другим способом управления данными являются буферы и изображения. 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 в host domain code, так и в device domain code. Программа не имеет прямого доступа к буферам, а вместо этого использует объекты доступа. Элементы объекта буфера могут иметь скалярный тип данных (например, int, float или double), векторный тип данных или определенный пользователем класс или структуру.

Неявное перемещение данных обрабатывается средствами языка DPC++. Как правило, его проще использовать в начале разработки, потому что это позволяет сосредоточиться на выражении вычислений. Однако, когда в приоритете находится производительность, это повышает требования к контролю за ресурсами, т.е. к применению 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. Более того, векторные инструкции не могут быть выполнены с данными, которые расположены без выравнивания адреса. 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 [9]:

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 по конкретному адресу памяти.

### Минимизация взаимных блокировок

Использование примитива синхронизации предполагает последовательный доступ к одним и тем же ячейкам памяти для вычислительных единиц. Так, вычислительные ядра выстраиваются в очередь на чтение и запись, поэтому они какое-то время могут простаивать в ожидании своей очереди. С увеличением количества вычислителей неизбежно растет время доступа к общему ресурсу. И наоборот, снижение количества вычислителей приведет к снижению времени ожидания на запись. Несмотря на увеличение количества вычислителей, прирост производительности постепенно приходит в насыщение. Чтобы этого избежать, необходимо снизить взаимные блокировки 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. Однако, в случае с другими архитектурами, это может быть не так. Например, архитектуры центрального и графического процессоров во многом различны, т.к. они оптимизированы под разные задачи.

Большинство задач GPU связаны с отрисовкой графики, и они не требуют такой высокой точности. В связи с этим, операции с 64 битными типами данных выполняется медленнее, особенно в видеокартах пользовательского сегмента [8]. 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 запустить в разы больше групп потоков.

Стоит добавить, что 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 параллельного алгоритма для CPU и GPU до и после оптимизации, разработанную на языке DPC++ [3].

Характеристики испытательного стенда:

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.

Ниже приведена таблица №3 с результатами экспериментов до выполнения оптимизаций.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Вычислительное устройство | Число фотонов | | | |
| 100 000 | 1 000 000 | 10 000 000 | 100 000 000 |
| CPU Intel i9 9900K | 1 242 мс | 10 453 мс | 116 512 мс | 2 012 297 мс |
| GPU Intel UHD Graphics 630 | 5 809 мс | 40 154 мс | 485 944 мс | 8 087 531 мс |
| GPU NVIDIA RTX 3060 | 2 581 мс | 17 654 мс | 209 143 мс | 3 547 160 мс |

Таблица 3. Время выполнения параллельной неоптимизированной реализации на разных устройствах для разного количества фотонов.

Рис. 10. График зависимости времени выполнения от количества фотонов для неоптимизированной реализации для разных устройств.

После выполнения вышеописанных оптимизаций было выполнено повторное сравнение производительности. Результаты представлены в таблице №4.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Вычислительное устройство | Число фотонов | | | |
| 100 000 | 1 000 000 | 10 000 000 | 100 000 000 |
| CPU Intel i9  9900K | 916 мс | 8 739 мс | 101 055 мс | 1 689 416 мс |
| GPU Intel UHD  Graphics 630 | 611 мс | 6 263 мс | 62 556 мс | 621 138 мс |
| GPU NVIDIA  RTX 3060 | 394 мс | 1 044 мс | 6 166 мс | 60 830 мс |

Таблица 4. Время выполнения параллельной оптимизированной реализации на разных устройствах для разного количества фотонов.

Рис. 11. График зависимости времени выполнения от количества фотонов для оптимизированной реализации для разных устройств.

Как можно заметить, проведенные оптимизации незначительно сказываются на скорости выполнения многопоточной реализации для CPU, однако дает значительное ускорения для GPU.

# Заключение

В рамках данной работы был реализован алгоритм моделирования распространения света в многослойных средах методом Монте-Карло в случае плоскопараллельной геометрии слоя. Также была разработана программная система, позволяющая производить вычисления одним из следующих способов:

* последовательно на CPU,
* параллельно на многоядерном CPU,
* параллельно на GPU,
* на гетерогенном кластере с использованием одновременно CPU и GPU.

Дополнительно было разработано ПО для визуализации результатов моделирования фотонов света в многослойной среде с плоскопараллельной геометрией. Выполнено исследование производительности полученной реализации алгоритма при переносе вычислений на GPU.

Результаты проведенных экспериментов показали, что перенос вычислений на GPU позволяет существенно уменьшить время работы алгоритма по сравнению с его параллельной версией, работающей на многоядерном процессоре.

Полученная и протестированная в результате настоящего исследования схема декомпозиции задачи может быть использована при разработке алгоритма моделирования распространения света в случае произвольной геометрии среды.

# Список литературы

1. Lihong Wang, Steven L. Jacques. Monte Carlo Modeling of Light Transport in Multi-layered Tissuesin Standard C. University of Texas M. D. Anderson Cancer Center. – 1992 г. – 185 с.
2. Lihong Wang, Steven L. Jacques, Liqiong Zheng. MCML – Monte Carlo modeling of light transport in multi-layered tissues. University of Texas M. D. Anderson Cancer Center. – 1995 г. – 186 с.
3. Reinders J., Ashbaugh B., Brodman J. Data Parallel C++. Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL. – 2021 г. – 548 с.
4. 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.
5. 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.
6. Соболь И.М. Численные методы Монте-Карло. Издательство "Наука". – 1973 г. – 313 с.
7. Некрасов К. А., Поташников С. И., Боярченков А. С., Купряжкин А. Я., Параллельные вычисления общего назначения на графических процессорах: учебное пособие. – 2016 г. – 108 с.
8. NVIDIA. CUDA C++ Best Practices Guide. – 2022 г. – 118 с.
9. Кофлер М., Сивченко О. Linux. Установка, настройка, администрирование. – 2014 г. – 768 с.
10. Dean J., Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. – 2004 г. – 13 с.
11. Khronos Group. OpenCL – официальный сайт: <http://www.khronos.org/opencl/>
12. AMD Corporation. HIP Programming Guide v5.3. – 2022 г.
13. Khronos Group. OpenGL® 4.5 Reference. – 2022 г.
14. Langr J. Modern C++ Programming with Test-Driven Development: Code Better, Sleep Better. – 2013 г.

# Приложение 1. Программный код алгоритма моделирования

struct LayerStruct

{

    data\_type\_t z0, z1;

    data\_type\_t n;

    data\_type\_t mua;

    data\_type\_t mus;

    data\_type\_t anisotropy;

    data\_type\_t cos\_crit0, cos\_crit1;

    inline bool is\_glass() const

    {

        return mua == 0.0 && mus == 0.0;

    }

    LayerStruct() = default;

    LayerStruct(const LayerStruct& o) :

        z0(o.z0),

        z1(o.z1),

        n(o.n),

        mua(o.mua),

        mus(o.mus),

        anisotropy(o.anisotropy),

        cos\_crit0(o.cos\_crit0),

        cos\_crit1(o.cos\_crit1)

    {;}

};

struct InputStruct

{

    char out\_fname[256];

    char out\_fformat;

    long num\_photons;

    data\_type\_t Wth;

    data\_type\_t dz;

    data\_type\_t dr;

    data\_type\_t da;

    short nz;

    short nr;

    short na;

    short   num\_layers;

    short   num\_output\_layers;

    LayerStruct\* layerspecs{ nullptr };

    InputStruct() = default;

    InputStruct(const InputStruct&) = default;

    void configure(short number\_of\_layers, short number\_of\_output\_layers, LayerStruct\* layerspecs\_ptr = nullptr);

};

data\_type\_t RFresnel(data\_type\_t incidentRefractiveIndex, data\_type\_t transmitRefractiveIndex, data\_type\_t incidentCos, data\_type\_t& transmitCos)

{

    data\_type\_t reflectance;

    if (incidentRefractiveIndex == transmitRefractiveIndex)

    {

        transmitCos = incidentCos;

        reflectance = 0.0;

    }

    else if (incidentCos > COSZERO)

    {

        transmitCos = incidentCos;

        reflectance = (transmitRefractiveIndex - incidentRefractiveIndex) / (transmitRefractiveIndex + incidentRefractiveIndex);

        reflectance \*= reflectance;

    }

    else if (incidentCos < COS90D)

    {

        transmitCos = 0.0;

        reflectance = 1.0;

    }

    else

    {

        data\_type\_t incidentSin = libset::sqrt(1.0F - incidentCos \* incidentCos);

        data\_type\_t transmitSin = incidentRefractiveIndex \* incidentSin / transmitRefractiveIndex;

        if (transmitSin >= 1.0)

        {

            transmitCos = 0.0;

            reflectance = 1.0;

        }

        else

        {

            transmitCos = libset::sqrt(1.0 - transmitSin \* transmitSin);

            data\_type\_t cap = incidentCos \* transmitCos - incidentSin \* transmitSin; /\* c+ = cc - ss. \*/

            data\_type\_t cam = incidentCos \* transmitCos + incidentSin \* transmitSin; /\* c- = cc + ss. \*/

            data\_type\_t sap = incidentSin \* transmitCos + incidentCos \* transmitSin; /\* s+ = sc + cs. \*/

            data\_type\_t sam = incidentSin \* transmitCos - incidentCos \* transmitSin; /\* s- = sc - cs. \*/

        }

    }

    return reflectance;

}

struct PhotonStruct

{

    template<class T>

    class weight\_tracker

    {

        T                      \_\_weight;

        PhotonStruct&          \_\_ps;

        matrix\_view\_adaptor<T> \_\_view;

        inline void \_\_track(T value)

        {

            if constexpr (CONFIGURATION\_WORK\_GROUP\_THREADS\_COUNT == 1U)

            {

                \_\_view.at(\_\_ps.x, \_\_ps.y, \_\_ps.z, 0) += value;

            }

            else

            {

#ifdef FEATURE\_USE\_LOCAL\_MEMORY

                sycl::atomic\_ref<T, sycl::memory\_order::relaxed, sycl::memory\_scope::work\_group, sycl::access::address\_space::local\_space>

                    atomic(\_\_view.at(\_\_ps.x, \_\_ps.y, \_\_ps.z, 0));

#else

                sycl::atomic\_ref<T, sycl::memory\_order::relaxed, sycl::memory\_scope::work\_group, sycl::access::address\_space::ext\_intel\_global\_device\_space>

                    atomic(\_\_view.at(\_\_ps.x, \_\_ps.y, \_\_ps.z, 0));

#endif

                atomic.fetch\_add(value);

            }

        }

    public:

        weight\_tracker(data\_type\_t weight, PhotonStruct& ps, matrix\_view\_adaptor<T>& view) :

            \_\_weight{ weight }, \_\_ps{ ps }, \_\_view{ view }

        {;}

        weight\_tracker<T>& operator=(T value)

        {

            \_\_weight = value;

            return \*this;

        }

        weight\_tracker<T>& operator+=(T value)

        {

            \_\_track(value);

            return \*this;

        }

        weight\_tracker<T>& operator-=(T value)

        {

            \_\_track(value);

            \_\_weight -= value;

            return \*this;

        }

        weight\_tracker<T>& operator/=(T value)

        {

            T tmp = \_\_weight;

            \_\_weight /= value;

            \_\_track(tmp - \_\_weight);

            return \*this;

        }

        weight\_tracker<T>& operator\*=(T value)

        {

            T tmp = \_\_weight;

            \_\_weight \*= value;

            \_\_track(tmp - \_\_weight);

            return \*this;

        }

        inline T operator\*(T value) const

        {

            return \_\_weight \* value;

        }

        inline bool operator==(T value) const

        {

            return \_\_weight == value;

        }

        inline bool operator!=(T value) const

        {

            return \_\_weight != value;

        }

        inline bool operator<(T value) const

        {

            return \_\_weight < value;

        }

    };

    data\_type\_t x{ 0 }, y{ 0 }, z{ 0 };    // vector of position

    data\_type\_t ux{ 0 }, uy{ 0 }, uz{ 0 }; // vector of direction

    weight\_tracker<data\_type\_t> w;

    bool dead{ false };

    size\_t layer{ 0 };

    data\_type\_t sleft{ 0 };

    data\_type\_t step\_size{ 0 };

    mcg59\_t &random;

    const InputStruct& input;

    const LayerStruct\* layerspecs;

    PhotonStruct(mcg59\_t& random, matrix\_view\_adaptor<data\_type\_t> view, const InputStruct& input) :

        w{ 0.0, \*this, view }, random{ random }, input{ input }, layerspecs{ input.layerspecs }

    {;}

    ~PhotonStruct() = default;

    void init(data\_type\_t Rspecular)

    {

        w = 1.0 - Rspecular;

        dead = 0;

        layer = 1; // LAYER CHANGE

        step\_size = 0;

        sleft = 0;

        x = 0.0; // COORD CHANGE

        y = 0.0;

        z = 0.0;

        ux = 0.0;

        uy = 0.0;

        uz = 1.0;

        if ((layerspecs[1].mua == 0.0) && (layerspecs[1].mus == 0.0))

        {

            layer = 2; // LAYER CHANGE

            z = layerspecs[2].z0;

        }

    }

    void spin(data\_type\_t anisotropy)

    {

        const auto ux = this->ux;

        const auto uy = this->uy;

        const auto uz = this->uz;

        const auto cost = SpinTheta(anisotropy);

        const auto sint = libset::sqrt(1.0F - cost \* cost);

        const auto psi = 2.0F \* (data\_type\_t)PI \* get\_random();

        const auto cosp = libset::cos(psi);

        data\_type\_t sinp; // = std::sin(psi);

        if (psi < PI)

        {

            sinp = libset::sqrt(1.0F - cosp \* cosp);

        }

        else

        {

            sinp = -libset::sqrt(1.0F - cosp \* cosp);

        }

        if (libset::abs<data\_type\_t>(uz) > COSZERO)

        {

            const auto temp = (uz >= 0.0) ? 1.0F : -1.0F;

            this->ux = sint \* cosp;

            this->uy = sint \* sinp;

            this->uz = temp \* cost;

        }

        else

        {

            const auto temp = sycl::sqrt<data\_type\_t>(1.0F - uz \* uz);

            this->ux = temp \* (ux \* uz \* -sint - uy \* cost) / temp + ux \* olayer;

            this->uy = sint \* (uz \* temp \* cost + ux \* sinp) / olayer + uy \* cost;

            this->uz = -sint \* cosp \* temp + uz \* cost;

        }

    }

    void hop()

    {

        // COORD CHANGE

        x += step\_size \* ux;

        y += step\_size \* uy;

        z += step\_size \* uz;

    }

    void step\_size\_in\_glass()

    {

        const auto& olayer = get\_current\_layer();

        if (uz > 0.0)

        {

            step\_size = (olayer.z1 - z) / uz;

        }

        else if (uz < 0.0)

        {

            step\_size = (olayer.z0 - z) / uz;

        }

        else

        {

            step\_size = 0.0;

        }

    }

    bool hit\_boundary()

    {

        if (libset::abs(uz) > 1e-10)

        {

            const auto& olayer = get\_current\_layer();

            data\_type\_t dl\_b;

            if (uz > 0.0)

            {

                dl\_b = olayer.z1;

            }

            else

            {

                dl\_b = olayer.z0;

            }

            dl\_b = (dl\_b - z) / uz;

            if (step\_size > dl\_b)

            {

                const auto mut = olayer.mua + olayer.mus;

                sleft = (step\_size - dl\_b) \* mut;

                step\_size = dl\_b;

                return true;

            }

        }

        return false;

    }

    void roulette()

    {

        if (w != 0.0F && get\_random() < CHANCE)

        {

            w /= CHANCE;

        }

        else

        {

            dead = true;

        }

    }

    void record\_r(data\_type\_t reflectance)

    {

        w \*= reflectance;

    }

    void record\_t(data\_type\_t reflectance)

    {

        w \*= reflectance;

    }

    void drop()

    {

        const auto& olayer = get\_current\_layer();

        auto mua = olayer.mua;

        auto mus = olayer.mus;

        auto dwa = w \* mua / (mua + mus);

        w -= dwa;

    }

    void cross\_up\_or\_not()

    {

        data\_type\_t uz1 = 0.0;

        data\_type\_t r = 0.0;

        data\_type\_t ni = layerspecs[layer].n;

        data\_type\_t nt = layerspecs[layer - 1].n;

        if (-uz <= layerspecs[layer].cos\_crit0)

        {

            r = 1.0;

        }

        else

        {

            r = Fresnel\_data\_ext(nfi, ntf, up, up1);

        }

        if (get\_random() > r)

        {

            if (layer == 1)

            {

                uz = -uz1;

                dead = true;

            }

            else

            {

                layer--; // LAYER CHANGE

                ux \*= ni / nt;

                uy \*= ni / nt;

                uz = -uz1;

            }

        }

        else

        {

            uz = -uz;

        }

    }

    void cross\_down\_or\_not()

    {

        data\_type\_t uz1 = 0.0;

        data\_type\_t r = 0.0;

        data\_type\_t ni = layerspecs[layer].n;

        data\_type\_t nt = layerspecs[layer + 1].n;

        if (uz <= layerspecs[layer].cos\_crit1)

        {

            r = 1.0;

        }

        else

        {

            r = Fresnel\_data\_ext(nfi, ntf, up, up1);

        }

        if (get\_random() > r)

        {

            if (layer == input.num\_layers)

            {

                uz = uz1;

                record\_t(0.0);

                dead = true;

            }

            else

            {

                layer++; // LAYER CHANGE

                ux \*= ni / nt;

                uy \*= ni / nt;

                uz = uz1;

            }

        }

        else

        {

            uz = -uz;

        }

    }

    void cross\_or\_not()

    {

        if (uz < 0.0)

        {

            cross\_up\_or\_not();

        }

        else

        {

            cross\_down\_or\_not();

        }

    }

    data\_type\_t SpinTheta(data\_type\_t anisotropy)

    {

        data\_type\_t cost = 2.0 \* get\_random() - 1.0;

        if (libset::abs(anisotropy) > 1e-10)

        {

            const data\_type\_t anisotropy\_sqr = anisotropy \* anisotropy;

            const data\_type\_t temp = (1.0 - anisotropy\_sqr) / (1.0 + anisotropy \* cost);

            cost = 0.5 \* (1.0 + anisotropy\_sqr - temp \* temp) / anisotropy;

            cost = libset::clamp<data\_type\_t>(cost, -1.0f, +1.0f);

        }

        return cost;

    }

    void hop\_in\_glass()

    {

        if (uz == 0.0)

        {

            dead = true;

        }

        else

        {

            step\_size\_in\_glass();

            hop();

            cross\_or\_not();

        }

    }

    void hop\_drop\_spin()

    {

        const auto& olayer = get\_current\_layer();

        if (olayer.is\_glass())

        {

            hop\_in\_glass();

        }

        else

        {

            const auto mua = olayer.mua;

            const auto mus = olayer.mus;

            data\_type\_t rnd;

#ifdef OPT\_RAND

            do

            {

                rnd = get\_random();

            } while (rnd <= 0.0);

            sleft = libset::abs<data\_type\_t>(sleft) < 1e-10 ? -sycl::log(rnd) : sleft;

            step\_size = sleft / (mua + mus);

#else

            if (sycl::abs(sleft) < 1e-10)

            {

                data\_type\_t rnd;

                do

                {

                    rnd = get\_random();

                }

                while (rnd <= 0.0);

                step\_size = -sycl::log(rnd) / (mua + mus);

            }

            else

            {

                step\_size = sleft / (mua + mus);

            }

#endif

            sleft = 0.0;

            if (hit\_boundary())

            {

                hop();

                cross\_or\_not();

            }

            else

            {

                hop();

                drop();

                spin(olayer.anisotropy);

            }

        }

        if (w < input.Wth && !dead)

        {

            roulette();

        }

    }

    inline const LayerStruct& get\_current\_layer() const

    {

        return layerspecs[layer];

    }

    inline data\_type\_t get\_random()

    {

        return random.next();

    }

};

# Приложение 2. Программный код распределения параллельных вычислений

#include <CL/sycl.hpp>

#include <vector>

#include <iostream>

#include <fstream>

#include <iomanip>

#include "matrix.hpp"

#include "iofile.hpp"

#include "mcml.hpp"

iofile fmanager;

static auto exception\_handler = [](sycl::exception\_list e\_list)

{

    for (std::exception\_ptr const& e : e\_list)

    {

        try

        {

            std::rethrow\_exception(e);

        }

        catch (std::exception const& e)

        {

            std::cout << "exception: " << e.what() << std::endl;

            std::terminate();

        }

    }

};

template<class T>

struct sycl\_host\_allocator

{

    sycl::queue& queue;

    sycl\_host\_allocator<T>(sycl::queue& queue) :

        queue{ queue }

    {;}

    sycl\_host\_allocator<T>(sycl\_host\_allocator<T>& other) :

        queue{ other.queue }

    {;}

    T\* allocate(size\_t size)

    {

        return sycl::malloc\_host<T>(size, queue);

    }

    void free(T\* data)

    {

        assert(data);

        if (data)

        {

            sycl::free(data, queue);

        }

    }

};

struct sycl\_base\_allocator

{

    sycl\_base\_allocator(sycl::queue& queue) :

        \_\_queue{ queue }

    {;}

    void free(void\* data)

    {

        assert(data);

        if (data)

        {

            sycl::free(data, \_\_queue);

            stat\_free();

        }

    }

    ~sycl\_base\_allocator()

    {

        assert(\_\_stat\_allocs\_count == \_\_stat\_fries\_count);

        std::cout << "Allocator stats: " << \_\_stat\_allocs\_count << " / " << \_\_stat\_fries\_count << std::endl;

    }

protected:

    sycl::queue& \_\_queue;

    void stat\_allocate()

    {

        \_\_stat\_allocs\_count++;

    }

private:

    size\_t \_\_stat\_allocs\_count{ 0 };

    size\_t \_\_stat\_fries\_count{ 0 };

private:

    void stat\_free()

    {

        \_\_stat\_fries\_count++;

    }

};

struct sycl\_shared\_allocator : public sycl\_base\_allocator

{

    sycl\_shared\_allocator(sycl::queue& queue) :

        sycl\_base\_allocator(queue)

    {;}

    template<class T>

    T\* allocate(size\_t size)

    {

        auto pointer = sycl::malloc\_shared<T>(size, \_\_queue);

        if (pointer)

        {

            stat\_allocate();

        }

        return pointer;

    }

};

struct sycl\_device\_allocator : public sycl\_base\_allocator

{

    sycl\_device\_allocator(sycl::queue& queue) :

        sycl\_base\_allocator(queue)

    {;}

    template<class T>

    T\* allocate(size\_t size)

    {

#ifdef FEATURE\_ALIGNED\_DEVICE\_ALLOC

        auto pointer = sycl::aligned\_alloc\_device<T>(sizeof(T), size, \_\_queue);

#else

        auto pointer = sycl::malloc\_device<T>(size, \_\_queue);

#endif

        if (pointer)

        {

            stat\_allocate();

        }

        return pointer;

    }

};

template<class T>

struct sycl\_unique\_ptr

{

    template<class TAlloc>

    sycl\_unique\_ptr(size\_t size, TAlloc& allocator) :

        \_\_allocator { allocator }, \_\_ptr{ allocator. template allocate<T>(size) }

    {;}

    T\* get()

    {

        return \_\_ptr;

    }

    ~sycl\_unique\_ptr()

    {

        \_\_allocator.free(\_\_ptr);

    }

private:

    sycl\_base\_allocator& \_\_allocator;

    T\* \_\_ptr;

};

template<class T>

using host\_matrix\_view = memory\_matrix\_view<T, sycl\_host\_allocator<T>>;

bool has\_local\_memory(const sycl::device &device)

{

    auto is\_host = device.is\_host();

    if (!is\_host)

    {

        return device.get\_info<sycl::info::device::local\_mem\_type>() != sycl::info::local\_mem\_type::none;

    }

    return is\_host;

}

void print\_device\_info(const sycl::device& device)

{

    std::cout << "Name:                  " << device.get\_info<sycl::info::device::name>()                << std::endl;

    std::cout << "Version:               " << device.get\_info<sycl::info::device::version>()             << std::endl;

    std::cout << "Vendor:                " << device.get\_info<sycl::info::device::vendor>()              << std::endl;

    std::cout << "Driver version:        " << device.get\_info<sycl::info::device::driver\_version>()      << std::endl;

    std::cout << "                                                                                 "     << std::endl;

    std::cout << "Max work group size:   " << device.get\_info<sycl::info::device::max\_work\_group\_size>() << std::endl;

    std::cout << "Global memory size:    " << device.get\_info<sycl::info::device::global\_mem\_size>()     << std::endl;

    bool has\_local\_mem = has\_local\_memory(device);

    if (has\_local\_mem)

    {

        auto local\_mem\_size = device.get\_info<sycl::info::device::local\_mem\_size>();

        std::cout << "Local memory size:     " << local\_mem\_size << std::endl;

    }

    else

    {

        std::cout << "Device has no avaible local memory" << std::endl;

    }

    std::cout << std::endl;

}

int main(int argc, char\* argv[])

{

#if defined(MCML\_CPU\_SINGLE\_THREAD) || defined(MCML\_CPU\_MULTIPLE\_THREADS)

    sycl::cpu\_selector d\_selector;

#else

    sycl::gpu\_selector d\_selector;

#endif

    print\_device\_info(d\_selector.select\_device());

    constexpr size\_t N\_x = LAYER\_OUTPUT\_SIZE\_X;

    constexpr size\_t N\_y = LAYER\_OUTPUT\_SIZE\_Y;

    constexpr size\_t N\_z = LAYER\_OUTPUT\_SIZE\_Z;

    constexpr size\_t N\_l = LAYER\_OUTPUT\_COUNT;

    constexpr size\_t random\_seed = SIMULATION\_RANDOM\_SEED;

    constexpr size\_t number\_of\_layers = SIMULATION\_LAYERS\_COUNT;

    constexpr size\_t N\_repeats = SIMULATION\_REPEATS\_COUNT;

    constexpr size\_t work\_group\_size = CONFIGURATION\_WORK\_GROUP\_THREADS\_COUNT;

    constexpr size\_t num\_groups = CONFIGURATION\_WORK\_GROUP\_SIZE;

    constexpr size\_t total\_threads\_count = num\_groups \* work\_group\_size;

    constexpr size\_t total\_photons\_runs = N\_repeats \* work\_group\_size \* num\_groups;

    try

    {

        sycl::queue queue(d\_selector, exception\_handler);

        sycl\_host\_allocator<data\_type\_t> allocator(queue);

        sycl\_shared\_allocator shared\_allocator(queue);

        sycl\_device\_allocator device\_allocator(queue);

        host\_matrix\_view<data\_type\_t> host\_view(N\_x, N\_y, N\_z, N\_l, allocator);

        size\_t N = host\_view.properties().length();

        sycl\_unique\_ptr<data\_type\_t> data(N, device\_allocator);

        sycl\_unique\_ptr<data\_type\_t> group\_data\_pool(N \* num\_groups, device\_allocator);

        sycl\_unique\_ptr<LayerStruct> layerspecs(number\_of\_layers, shared\_allocator);

        InputStruct input;

        input.configure(number\_of\_layers, N\_l, layerspecs.get());

        std::cout << "Image dimensions of x: " << N\_x                 << std::endl;

        std::cout << "Image dimensions of y: " << N\_y                 << std::endl;

        std::cout << "Image dimensions of z: " << N\_z                 << std::endl;

        std::cout << "Image count of layers: " << N\_l                 << std::endl;

        std::cout << "                                              " << std::endl;

        std::cout << "Random seed:           " << random\_seed         << std::endl;

        std::cout << "Inner layes:           " << number\_of\_layers    << std::endl;

        std::cout << "                                              " << std::endl;

        std::cout << "Repeats per item:      " << N\_repeats           << std::endl;

        std::cout << "Work group size:       " << work\_group\_size     << std::endl;

        std::cout << "Number of groups:      " << num\_groups          << std::endl;

        std::cout << "                                              " << std::endl;

        std::cout << "Total work item count: " << total\_threads\_count << std::endl;

        std::cout << "Total photon runs:     " << total\_photons\_runs  << " ( log10 = " << sycl::log10<data\_type\_t>(total\_photons\_runs) << " )" << std::endl;

        std::cout << "Total memory used:     " << (N \* num\_groups + N) \* sizeof(data\_type\_t) << " bytes" << std::endl;

        std::cout << "                                              " << std::endl;

        data\_type\_t\* device\_data = data.get();

        data\_type\_t\* device\_group\_data\_pool = group\_data\_pool.get();

        queue.parallel\_for(num\_groups,

            [=](auto group\_index)

            {

                data\_type\_t\* data = device\_group\_data\_pool + N \* group\_index;

                for (int i = 0; i < N; ++i)

                {

                    data[i] = 0;

                }

            });

        auto time\_start = std::chrono::high\_resolution\_clock::now();

        queue.submit(

            [&](sycl::handler& handler)

            {

                handler.parallel\_for\_work\_group<class PhotonKernel>(sycl::range<1>(num\_groups), sycl::range<1>(work\_group\_size),

                    [=](sycl::group<1> group)

                    {

                        size\_t gid = group.get\_group\_id(0); // work group index

                        data\_type\_t\* group\_data\_pool = device\_group\_data\_pool + N \* gid;

#if defined(FEATURE\_USE\_LOCAL\_MEMORY)

                        data\_type\_t memory[N\_x \* N\_y \* N\_z \* N\_l]{ 0.0 };

#else

                        data\_type\_t\* memory = group\_data\_pool;

#endif // FEATURE\_USE\_LOCAL\_MEMORY

                        group.parallel\_for\_work\_item(

                            [&](sycl::h\_item<1> item)

                            {

                                uint64\_t thread\_global\_id = item.get\_global\_id();

                                mcg59\_t random\_generator(random\_seed, thread\_global\_id, work\_group\_size \* num\_groups);

                                raw\_memory\_matrix\_view<data\_type\_t> view(memory, N\_x, N\_y, N\_z, N\_l);

                                PhotonStruct photon(random\_generator, view, input);

                                auto rsp = Rspecular(input.layerspecs);

                                for (size\_t j = 0; j < N\_repeats; ++j)

                                {

                                    photon.init(rsp);

                                    do

                                    {

                                        photon.hop\_drop\_spin();

                                    }

                                    while (!photon.dead);

                                }

                            });

#if defined(FEATURE\_USE\_LOCAL\_MEMORY) || defined(FEATURE\_USE\_ATOMIC\_SUMMATOR)

                        group.parallel\_for\_work\_item(

                            [&](sycl::h\_item<1> item)

                            {

                                static\_assert(N\_x \* N\_y \* N\_z \* N\_l % work\_group\_size == 0);

                                constexpr auto batch\_size = N\_x \* N\_y \* N\_z \* N\_l / work\_group\_size;

                                auto thread\_local\_id = item.get\_local\_id();

                                auto thread\_local\_offset = batch\_size \* thread\_local\_id;

                                for (int i = 0; i < batch\_size; ++i)

                                {

#if defined(FEATURE\_USE\_ATOMIC\_SUMMATOR)

                                    sycl::atomic\_ref<data\_type\_t, sycl::memory\_order::relaxed, sycl::memory\_scope::work\_group, sycl::access::address\_space::ext\_intel\_global\_device\_space>

                                        atomic(device\_data[thread\_local\_offset + i]);

                                    atomic.fetch\_add(memory[thread\_local\_offset + i]);

#else

                                    group\_data\_pool[thread\_local\_offset + i] = memory[thread\_local\_offset + i];

#endif

                                }

                            });

#endif // FEATURE\_USE\_LOCAL\_MEMORY || FEATURE\_USE\_ATOMIC\_SUMMATOR

                    });

            });

        queue.wait();

#if !defined(FEATURE\_USE\_ATOMIC\_SUMMATOR)

#if defined(FEATURE\_USE\_GROUP\_SUMMATOR)

        queue.submit(

            [&](sycl::handler& handler)

            {

                constexpr auto work\_group\_size = N\_y;

                constexpr auto work\_group\_count = N\_x;

                static\_assert(work\_group\_count <= 256);

                static\_assert(work\_group\_size <= 256);

                handler.parallel\_for\_work\_group(sycl::range<1>(work\_group\_count), sycl::range<1>(work\_group\_size),

                    [=](sycl::group<1> group)

                    {

                        constexpr auto batch\_size = (N\_x \* N\_y \* N\_z \* N\_l) / (work\_group\_size \* work\_group\_count);

                        constexpr auto work\_group\_data\_size = work\_group\_size \* batch\_size;

                        auto work\_group\_index = static\_cast<size\_t>(group.get\_group\_id());

                        auto work\_group\_data\_offset = work\_group\_index \* work\_group\_data\_size;

                        group.parallel\_for\_work\_item(

                            [&](sycl::h\_item<1> item)

                            {

                                auto item\_local\_index = static\_cast<size\_t>(item.get\_local\_id());

                                auto item\_local\_offset = batch\_size \* item\_local\_index;

                                data\_type\_t thread\_local\_summ\_batch[batch\_size]{ 0.0 };

                                for (size\_t data\_group\_index = 0; data\_group\_index < num\_groups; ++data\_group\_index)

                                {

                                    auto device\_group\_data = device\_group\_data\_pool + N \* data\_group\_index;

                                    auto group\_batch\_data = device\_group\_data + work\_group\_data\_offset + item\_local\_offset;

                                    for (int i = 0; i < batch\_size; ++i)

                                    {

                                        thread\_local\_summ\_batch[i] += group\_batch\_data[i];

                                    }

                                }

                                auto device\_batch\_data = device\_data + work\_group\_data\_offset + item\_local\_offset;

                                for (int i = 0; i < batch\_size; ++i)

                                {

                                    device\_batch\_data[i] = thread\_local\_summ\_batch[i];

                                }

                            });

                    });

            });

#else

        queue.parallel\_for(N,

            [=](auto idx)

            {

                data\_type\_t summ = 0;

                for (size\_t data\_group\_index = 0; data\_group\_index < num\_groups; ++data\_group\_index)

                {

                    data\_type\_t\* data = device\_group\_data\_pool + N \* data\_group\_index;

                    summ += data[idx];

                }

                device\_data[idx] = summ;

            });

#endif // !FEATURE\_USE\_GROUP\_SUMMATOR

        queue.wait();

#endif

        auto time\_end = std::chrono::high\_resolution\_clock::now();

        auto time\_duration = std::chrono::duration\_cast<std::chrono::milliseconds>(time\_end - time\_start);

        std::cout << "Elapsed time: " << time\_duration.count() << " ms" << std::endl;

        queue.memcpy(host\_view.data(), device\_data, host\_view.size\_of\_data());

        queue.wait();

        //

        matrix\_utils::normalize\_v2(host\_view);

        auto [size\_x, size\_y, size\_z, size\_l] = host\_view.properties().size();

        if (size\_x < 11 && size\_y < 11 && size\_z < 11)

        {

            for (size\_t y = 0; y < size\_y; ++y)

            {

                for (size\_t x = 0; x < size\_x; ++x)

                {

                    data\_type\_t v = 0.0F;

                    for (size\_t z = 0; z < size\_z; ++z)

                    {

                        for (size\_t l = 0; l < size\_l; ++l)

                        {

                            v += host\_view.at(x, y, z, l);

                        }

                    }

                    std::cout << std::setw(8) << v << ' ';

                }

                std::cout << '\n';

            }

        }

        {

            auto file = fmanager.open();

            fmanager.export\_file(host\_view, file);

        }

    }

    catch (const sycl::exception& e)

    {

        std::cout << "An exception is caught: " << e.what() << std::endl;

        std::terminate();

    }

    return 0;

}