**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ  
ФЕДЕРАЦИИ**

Федеральное государственное автономное образовательное учреждение  
высшего образования

**«Национальный исследовательский**

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

**Институт информационных технологий, математики и механики**

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

**МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ**

**«Моделирование распространения света**

**с применением технологии Intel OneAPI»**

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

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

Нижний Новгород

2023

# Оглавление

[Оглавление 2](#_Toc133776189)

[Введение 3](#_Toc133776190)

[1. Постановка задачи 5](#_Toc133776191)

[2. Метод решения (об алгоритме распространения фотонов) 6](#_Toc133776192)

[3. Обзор метода Монте-Карло 9](#_Toc133776193)

[4. Обзор технологий параллельных вычислений 11](#_Toc133776194)

[5. Обзор методов программирования на GPU 12](#_Toc133776195)

[3.1. Технология Nvidia CUDA 12](#_Toc133776196)

[Архитектура CUDA и язык CUDA C 12](#_Toc133776197)

[~~Взаимодействия центрального и графического процессоров~~ 12](#_Toc133776198)

[~~Характеристики устройства~~ 14](#_Toc133776199)

[5.2. Технология Intel OneAPI 16](#_Toc133776200)

[Стандарт OneAPI и язык DPC++ 16](#_Toc133776201)

[Архитектура 17](#_Toc133776202)

[Вычислительные единицы DPC++ 18](#_Toc133776203)

[Доступ к памяти между устройствами 19](#_Toc133776204)

[6. Подсистема памяти многопоточного приложения 21](#_Toc133776205)

[7. Программная оптимизация 25](#_Toc133776206)

[8. Генерация псевдослучайных чисел 26](#_Toc133776207)

[Линейный конгруэнтный метод 26](#_Toc133776208)

[9. Параллельная реализация метода Монте-Карло OneAPI 29](#_Toc133776209)

[Результаты моделирования 29](#_Toc133776210)

[Заключение 31](#_Toc133776211)

[Литература 32](#_Toc133776212)

# Введение

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

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

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

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

# Постановка задачи

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

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

Решение поставленной задачи разбивается на следующие этапы:

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

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

# Метод решения (об алгоритме распространения фотонов)

Метод распространения света биологических тканей описан в оригинальной работе группы ученых Техасского университета в США в 1993 году [6]. Оригинальные тексты последовательной программы были опубликованы в глобальной сети Интернет.

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

Распространение фотона в среде описывается в декартовых координатах. Положение фотона определяется координатами 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.

Каждое испытание проводится независимо друг от друга, таким образом каждое испытание может быть выполнено параллельно.

# Обзор метода Монте-Карло

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

Так, предлагаемый параллельный метод Монте Карло применяемый для моделирования

Идея метода Монте-Карло в приложение к описанной выше задаче состоит в случайной трассировке набора фотонов в биологической ткани. Общая трассировка фотона осуществляется на основании правил, описанных в оригинальной работе [6].

# Обзор технологий параллельных вычислений

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

Существует несколько технологий для параллельных вычислений:

1. Многопоточность – технология, которая позволяет выполнять несколько потоков вычислений в рамках одного процессора. Она используется в многоядерных процессорах для увеличения производительности.
2. OpenMP – стандартная технология для параллельных вычислений на общей памяти. Она позволяет разрабатывать многопоточные программы на C, C++ и Fortran, используя директивы препроцессора и библиотеки OpenMP.
3. MPI – технология, которая позволяет выполнять параллельные вычисления на распределенных системах. Она используется для обмена данными между процессами, работающими на различных компьютерах. MPI может быть использована в кластере, если они подключены к сети.
4. Nvidia CUDA – проприетарная технология компании Nvidia, которая позволяет использовать графические процессоры (GPU) для параллельных вычислений.
5. AMD HIP API – проприетарная технология компании AMD, которая позволяет использовать графические процессоры (GPU) для параллельных вычислений.
6. OpenCL – технология, которая позволяет использовать как GPU, так и центральные процессоры (CPU) для параллельных вычислений. Она используется для разработки кроссплатформенных параллельных приложений.
7. Intel OneAPI – это единый набор инструментов и библиотек для создания приложений, который был разработан компанией Intel для поддержки различных архитектур процессоров и ускорителей, включая центральные процессоры (CPU), графические процессоры (GPU) и программируемые ускорители (FPGA).

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | OpenMP | MPI | CUDA | AMD HIP | OpenCL | Intel OneAPI |
|  |  |  |  |  |  |  |
| Поддержка GPU AMD |  |  |  |  |  |  |
| Поддержка GPU Nvidia |  |  |  |  |  |  |
| Поддержка GPU Intel Arc |  |  |  |  |  |  |

# Обзор методов программирования на GPU

## 3.1. Технология Nvidia CUDA

### Архитектура CUDA и язык CUDA C

Язык программирования CUDA C является по существу языком С / С++ с некоторыми дополнениями для программирования видеокарт с архитектурой CUDA. Использование данного языка позволяет избежать применения технологий OpenGL и переноса задач в плоскость компьютерной графики с использованием графических ускорителей.

Как и в других стандартах распараллеливания программ, архитектура CUDA предоставляет вычислительные единицы - блоки, на которые будет распределена нагрузка главным потоком, выполняемым на CPU. Задачи будут переданы к исполнению доступными блоками, в распоряжении которых есть нити (thread) и разделяемая память.

Нитями (thread) называются потоки, которые разделяют время вычислительного блока (графического процессора). Во многом они являются точно такими же потоками, как и потоки ядра операционной системы, но во избежание путаницы в русскоязычном сообществе разработчиков их принято называть нитями. Количество нитей внутри блока ограничено. Нити внутри одного блока имеют доступ к общей памяти ограниченного объёма, которая называется разделяемой. Эти величины определены конкретной моделью графического процессора.

Одной из особенностей архитектуры CUDA является абстракция параллельных вычисления. Её можно сравнить с параллельным выполнением кода на некоторой абстрактной сетке в -мерном пространстве. Можно сказать, что каждый узел такой сетки будет являться вычислительным блоком, который будет выполнять предоставленный ему код. Внутри каждого блока можно однозначно определить принадлежность к соответствующему узлу этой абстрактной сетки координат (x, y, z) и использовать их для вычислений.

### ~~Взаимодействия центрального и графического процессоров~~

~~Тесная интеграция с принятой моделью компиляции программ на языке С требует разделения областей не только выполнения инструкций между CPU и GPU, но и хранения данных. С помощью новых квалификаторов функций и переменных удается разрешить эти конфликты в CUDA C.~~

~~Квалификаторы функций:~~

* ~~\_\_device\_\_ вызывается и выполняется только на графическом ускорителе.~~
* ~~\_\_global\_\_ выполняется на графическом ускорителе, но может быть вызвана как из GPU, так и из CPU. Возвращаемый тип такой функции допускается только void.~~
* ~~\_\_host\_\_ вызывается и выполняется только на центральном процессоре.~~

~~Квалификаторы переменных:~~

* ~~\_\_device\_\_ переменная размещается внутри видеокарты.~~
* ~~\_\_constant\_\_ может быть использована вместе с \_\_device\_\_, указывает, что переменная размещена в статической памяти и доступна из любой нити.~~
* ~~\_\_shared\_\_ может быть использована вместе с \_\_device\_\_, указывает, что к переменной имеют нити одного блока.~~

~~Устройство обладает собственной памятью, в которой могут быть размещены необходимые данные, переменные. Так, например, для обработки графики эта память используется для хранения текстур и быстрого доступа к ним. В случае нехватки этого объёма может быть предусмотрен механизм вытеснения данных в оперативную память, подконтрольную центральному процессору.~~

~~Для маршрутизации данных используются следующая функция, которая выполняет копирование данных:~~

~~cudaError\_t cudaMemcpy(void \*dst, const void \*src, size\_t count, enum cudaMemcpyKind kind)~~

~~Тип перечисления cudaMemcpyKind определяет «направление» копируемых данных.~~

~~enum cudaMemcpyKind~~

~~{~~

~~cudaMemcpyHostToHost, // копирование внутри оперативной памяти~~

~~cudaMemcpyHostToDevice, // из оперативной памяти в память устройства~~

~~cudaMemcpyDeviceToHost, // из памяти устройства в оперативную память~~

~~cudaMemcpyDeviceToDevice // копирование внутри памяти устройства~~

~~};~~

~~Указатели определяют количество байт count, которые лежат по адресу источника (dst) и адрес назначения (src). Причем по указанному адресу src должен быть выделен необходимый объём памяти.~~

~~Выделение памяти на устройстве может быть сделано так:~~

~~cudaError\_t cudaMalloc(void \*\*p, size\_t count)~~

~~В переданный адрес указателя будет записан адрес в памяти устройства объёмом count байт.~~

### ~~Характеристики устройства~~

~~Знание точных характеристик устройства может быть полезным для разработки и отладки параллельных программ.~~

~~Следующая программа предоставляет подробную информацию о ресурсах всех доступных графических ускорителей:~~

~~#include "cuda\_runtime.h"~~

~~#include "device\_launch\_parameters.h"~~

~~int getSPcores(cudaDeviceProp devProp)~~

~~{~~

~~int cores = 0;~~

~~int mp = devProp.multiProcessorCount;~~

~~switch (devProp.major)~~

~~{~~

~~case 2: // Fermi~~

~~if (devProp.minor == 1) cores = mp \* 48;~~

~~else cores = mp \* 32;~~

~~break;~~

~~case 3: // Kepler~~

~~cores = mp \* 192;~~

~~break;~~

~~case 5: // Maxwell~~

~~cores = mp \* 128;~~

~~break;~~

~~case 6: // Pascal~~

~~if (devProp.minor == 1) cores = mp \* 128;~~

~~else if (devProp.minor == 0) cores = mp \* 64;~~

~~else printf("Unknown device type\n");~~

~~break;~~

~~case 7: // Volta~~

~~if (devProp.minor == 0) cores = mp \* 64;~~

~~else printf("Unknown device type\n");~~

~~break;~~

~~default:~~

~~printf("Unknown device type\n");~~

~~break;~~

~~}~~

~~return cores;~~

~~}~~

~~void print\_cuda\_device\_info(cudaDeviceProp& prop)~~

~~{~~

~~printf("Device name: %s\n", prop.name);~~

~~printf("Global memory available on device: %zu\n", prop.totalGlobalMem);~~

~~printf("Shared memory available per block: %zu\n", prop.sharedMemPerBlock);~~

~~printf("Count of 32-bit registers available per block: %i\n", prop.regsPerBlock);~~

~~printf("Warp size in threads: %i\n", prop.warpSize);~~

~~printf("Maximum pitch in bytes allowed by memory copies: %zu\n", prop.memPitch);~~

~~printf("Maximum number of threads per block: %i\n", prop.maxThreadsPerBlock);~~

~~printf("Maximum size of each dimension of a block[0]: %i\n", prop.maxThreadsDim[0]);~~

~~printf("Maximum size of each dimension of a block[1]: %i\n", prop.maxThreadsDim[1]);~~

~~printf("Maximum size of each dimension of a block[2]: %i\n", prop.maxThreadsDim[2]);~~

~~printf("Maximum size of each dimension of a grid[0]: %i\n", prop.maxGridSize[0]);~~

~~printf("Maximum size of each dimension of a grid[1]: %i\n", prop.maxGridSize[1]);~~

~~printf("Maximum size of each dimension of a grid[2]: %i\n", prop.maxGridSize[2]);~~

~~printf("Clock frequency in kilohertz: %i\n", prop.clockRate);~~

~~printf("totalConstMem: %zu\n", prop.totalConstMem);~~

~~printf("Major compute capability: %i\n", prop.major);~~

~~printf("Minor compute capability: %i\n", prop.minor);~~

~~printf("Number of multiprocessors on device: %i\n", prop.multiProcessorCount);~~

~~printf("Count of cores: %i\n", getSPcores(prop));~~

~~}~~

~~int main()~~

~~{~~

~~int count;~~

~~cudaDeviceProp prop;~~

~~cudaGetDeviceCount(&count);~~

~~printf("Count CUDA devices = %i\n", count);~~

~~for (int i = 0; i < count; i++)~~

~~{~~

~~cudaGetDeviceProperties(&prop, i);~~

~~print\_cuda\_device\_info(prop);~~

~~}~~

~~return 0;~~

~~}~~

Как было сказано ранее выполнение управляющего кода происходит параллельно на выделенных абстрактных вычислительных блоках определенных в . Размерность каждого подпространства из ограничена какой-то константой для каждой модели графической карты. Если для выполнения задачи требуется выполнить какое-то множество подзадач, то разработчик должен наиболее точным образом указать определить количество блоков, выделив «прямоугольный параллелепипед» из этих блоков по каждой компоненте X, Y, Z.

## Технология Intel OneAPI

### Стандарт OneAPI и язык DPC++

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

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

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

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

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

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

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

### Архитектура

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

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

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

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

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

### Вычислительные единицы DPC++

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

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

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

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

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

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

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

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

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

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

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

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

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

### Доступ к памяти между устройствами

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

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

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

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

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

int main()

{

constexpr size\_t N = 1024;

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

// выделение памяти происходит на куче

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

// local scope

{

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

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

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

gpu\_queue.submit(

[&](sycl::handler& cgh)

{

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

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

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

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

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

{

// Kernel code

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

}

);

}

);

}

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

// выделенные на стеке, разрущаются => передают доступ

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

for (const auto& v : c)

{

assert(v == 3.0);

}

}

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

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

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

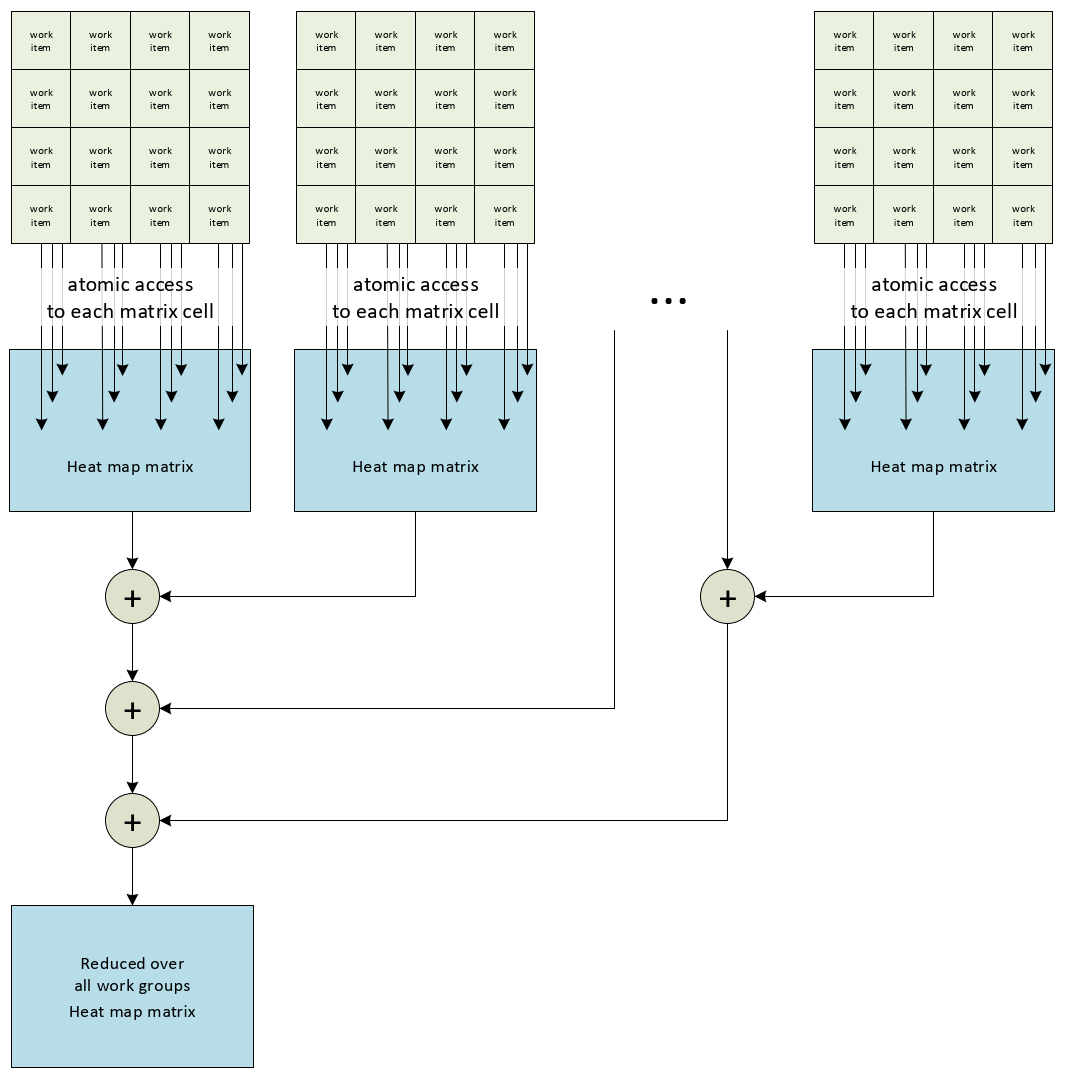
# Подсистема памяти многопоточного приложения

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 condition. Data condition 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.

Параллельная реализация метода Монте-Карло предполагает одновременный доступ к глобальной памяти, в которой хранится матрица потерянной энергии фотонов в каждой точке пространства. Учитывая особенность реализации, невозможно предсказать какой следующий адрес будет использован для сохранения результатов, поэтому применить какой-то потокобезопасный и, одновременно, эффективный алгоритм записи данных не представляется возможным. При обновлении значений предполагается эксклюзивный доступ для чтения и записи значений с помощью применения атомарных операций над областью памяти. Доступ к одним и тем же ячейкам памяти для разных ядер осуществляется поочередно, 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. Снижение количества

Несмотря на это, рост времени ожидания доступа может быть снижен путем разделения областей памяти и балансировки вычислений на них.

Наиболее приятной особенностью метода Монте-Карло является независимость работы вычислителей от результатов работы других вычислителей. С учетом этого, может быть предложена следующая архитектура параллельной реализации метода Монте-Карло с разделением памяти между группами вычислителей:



Основные положения такой реализации:

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

Так же стоит отметить возможность использования локальной памяти внутри группы вычислителей для малых размеров памяти, что должно положительно сказываться на скорости доступа. К сожалению, матрицы большого размера не получится оптимизировать таким образом, т.к. объем локальной памяти сильно ограничен, и как правило составляет десятки килобайт.

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.

Таким образом получается увеличить скорость вычислений распространения фотонов в плоскопараллельной среде.

**Локальная память**

Локальная память – это тип памяти, доступный для использования в графических ускорителях, который расположен непосредственно рядом с ядрами исполнения. С одной стороны, это даёт ей более высокую пропускную способность. С другой стороны, её объем, как правило, существенно меньше. 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.

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

# 

# Программная оптимизация

1. Распараллеливание вычислений
2. Подсистема памяти, атомарный доступ, минимизация взаимных блокировок, алгоритмы суммирования в локальной памяти и вытекающие проблемы
3. Проблема дивергенции
4. Применение векторных инструкций (актуально на CPU и старых GPU)

# Генерация псевдослучайных чисел

Метод Монте-Карло предполагает использование случайных равномерно распределенных чисел в интервале [0, 1]. В случае, если распределение не будет равномерным, результат решения статистическим методом будет искажен, и следовательно, задача будет решена неверно. Эта особенность задачи накладывает некоторые требования на генератор случайных чисел.

В данной работе будет применяться мультипликативный конгруэнтный генератор случайных чисел, описание которого приведено ниже.

## Линейный конгруэнтный метод

Линейный конгруэнтный метод является одной из простейших и наиболее употребительных в настоящее время процедур, имитирующих случайные числа. В этом методе используется операция mod(x, y), возвращающая остаток от деления первого аргумента на второй. Каждое последующее случайное число рассчитывается на основе предыдущего по следующей формуле:

где:

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

Последовательность случайных чисел, полученных с помощью данной формулы, называется линейной конгруэнтной последовательностью. Линейную конгруэнтную последовательность при называют мультипликативным конгруэнтным методом, а при — смешанным конгруэнтным методом.

Для качественного генератора требуется подобрать подходящие коэффициенты. Необходимо, чтобы число M было довольно большим, так как период не может иметь больше M элементов. С другой стороны, деление, использующееся в этом методе, является довольно медленной операцией, поэтому для двоичной вычислительной машины логичным будет выбор , поскольку в этом случае нахождение остатка от деления сводится внутри процессора к двоичной логической операции 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 .

Одним из требований к линейным конгруэнтным последовательностям является как можно большая длина периода. Длина периода зависит от значений , и . Теорема, которая приведена ниже, 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.

В случае, если псевдослучайные последовательности будут повторяться в разных потоках, псевдослучайные числа могут не обладать свойством равномерно распределенной с.в. В данной работе реализован алгоритм мультипликативного конгруэнтного генератора MCG59, где в качестве выбирается число . Данный генератор предоставляет возможность генерировать псевдослучайные последовательности чисел для многопоточных систем, а так же имеет достаточно простую реализацию:

struct mcg59\_t

{

constexpr static uint64\_t MCG59\_C = 302875106592253;

constexpr static uint64\_t MCG59\_M = 576460752303423488;

constexpr static uint64\_t MCG59\_DEC\_M = 576460752303423487;

uint64\_t value;

uint64\_t offset;

mcg59\_t(uint64\_t seed, unsigned int id, unsigned int step)

{

uint64\_t value = 2 \* seed + 1;

uint64\_t firstOffset = RaiseToPower(MCG59\_C, id);

value = (value \* firstOffset) & MCG59\_DEC\_M;

this->value = value;

this->offset = RaiseToPower(MCG59\_C, step);

}

double next()

{

this->value = (this->value \* this->offset) & MCG59\_DEC\_M;

return (double)(this->value) / MCG59\_M;

}

uint64\_t RaiseToPower(uint64\_t argument, unsigned int power)

{

uint64\_t result = 1;

while (power > 0)

{

if ((power & 1) == 0)

{

argument \*= argument;

power >>= 1;

}

else

{

result \*= argument;

--power;

}

}

return result;

}

};

# Параллельная реализация метода Монте-Карло OneAPI

В оригинальной работе [1] результаты численного эксперимента записываются в матрицы отражений, проникновений и поглощений. На основе этих данных рассчитываются характеристики исследуемого объекта.

В основном алгоритме могут быть выделены основные шаги:

1. Загрузка параметров симуляции.
2. Создание фотонов, инициализация данными в соответствии с параметрами симуляции.
3. Последовательное выполнение вычислений траектории распространения фотона в биологической ткани.
4. Регистрация значения в матрице отражений.
5. Регистрация значения в матрице проникновений.
6. Регистрация значения в матрице поглощений.
7. Если вес фотона достаточно мал или фотон покинул область исследования, то осуществляется переход к шагу №8, иначе к шагу №3.
8. Уменьшается количество фотонов на один, если их больше нуля, иначе вычисления завершаются.
9. Результат записывается в бинарный файл.

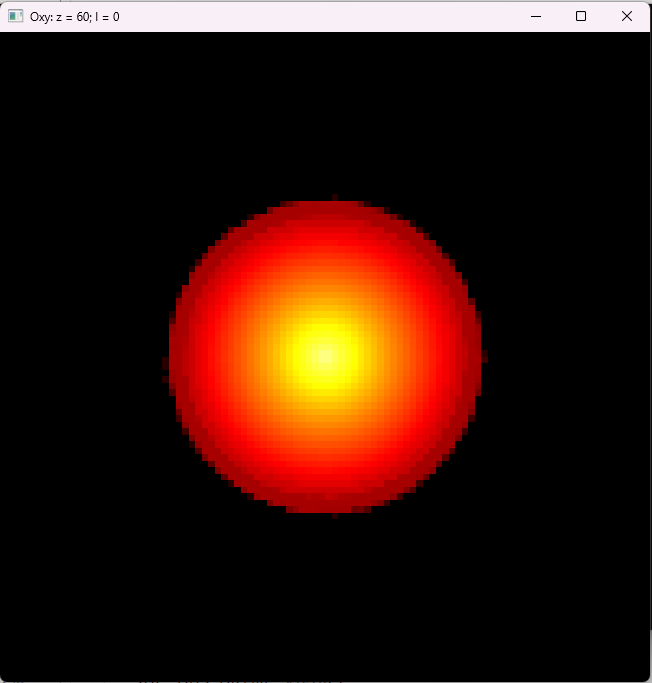
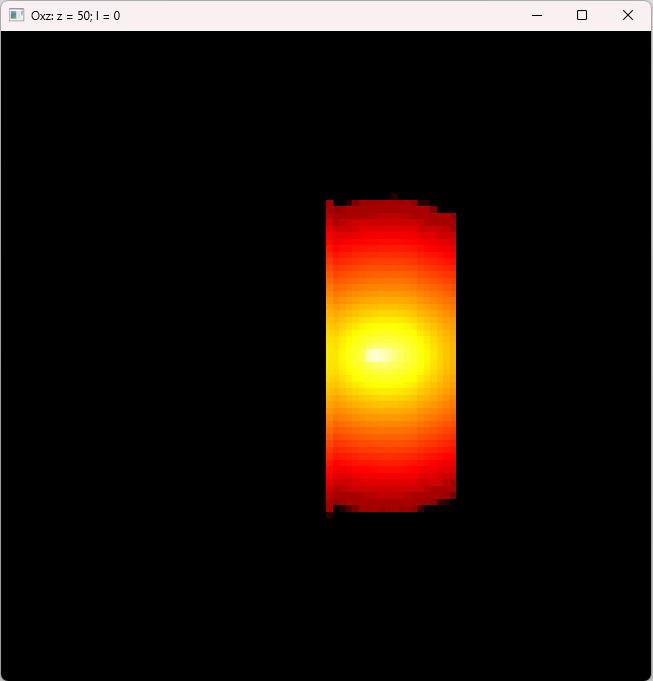
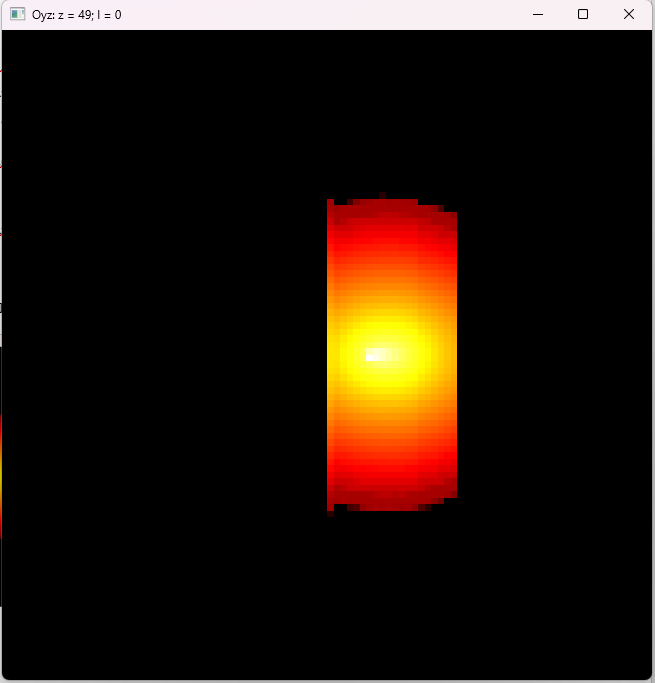
Шаги 2-7 могут быть выполнены независимо друг от друга, следовательно могут быть выполнены параллельно.

Оригинальная программа была адаптирована под современный стандарт C++, был произведен рефакторинг оригинального кода.

Запись итоговых значений выполняется в три матрицы, к которым следует предоставить совместный доступ на запись и чтение из *kernel*.

## Результаты моделирования

В качестве интерпретатора результатов моделирования используется дополнительно разработанное приложение для визуализации тепловой карты потерянного веса фотонов. Данное приложение предоставляет двумерное изображении сечения трехмерной матрицы:

На продемонстрированных рисунках можно наблюдать рассеивание света непосредственно от источника излучения. Наиболее белые участки соответствуют наиболее теплым участкам карты, т.е. наибольшему количеству потерянного веса в этой точке пространства.

# Заключение

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

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

# Литература

1. ~~NVIDIA Corporation: CUDA C Programming Guide – 2012 г. – 185с.~~
2. Сандерс Д., Кэндрот Э.: Технология CUDA в примерах. Введение в программирование графических процессов – 2018 г. – 232 с.
3. Боресков А. В., Харламов А.: Основы работы с технологией CUDA – 2010 г. – 231с.
4. Intel OneAPI – официальный сайт: <https://www.intel.com/oneapi/overview.html>
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. Wang L.V., Jacques S.L., Zheng L.Q. MCML – Monte Carlo modeling of light transport in multi-layered – 1995 г. – 186 с.
7. OpenCL – официальный сайт: <http://www.khronos.org/opencl/>
8. 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.
9. 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