

**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

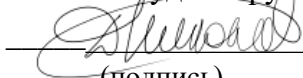
Направление подготовки: «Прикладная математика и информатика»
Магистерская программа: «Вычислительные методы и суперкомпьютерные
технологии»

ОТЧЕТ

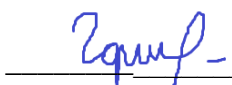
по проектно-технологической практике
на тему:

«Моделирование распространения света методом Монте Карло на GPU»

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
Введение.....	3
1. Постановка задачи	5
2. Обзор метода Монте-Карло	6
3. Обзор методов программирования на GPU	9
3.1. Технология Nvidia CUDA.....	9
Архитектура CUDA и язык CUDA C	9
Взаимодействия центрального и графического процессоров	9
Характеристики устройства.....	11
Средства синхронизации.....	17
3.2. Технология Intel OneAPI.....	19
Стандарт OneAPI и язык DPC++.....	19
Архитектура.....	20
Вычислительные единицы DPC++	21
Доступ к памяти между устройствами	22
4. Подсистема памяти многопоточного приложения	24
5. Применение типов пониженной точности.....	27
6. Генерация псевдослучайных чисел	28
Линейный конгруэнтный метод.....	28
7. Параллельная реализация метода Монте-Карло OneAPI.....	31
8. Результаты моделирования.....	34
Заключение	38
Литература	39

Введение

Одним из наиболее популярных численных методов решения сложных математических систем является метод Монте Карло, который находит свое применение в разных задачах, начиная от простого расчёта интеграла до изображения сцен компьютерной графики. В том числе в моделировании распространения света внутри биологических тканей. Реализация этого метода заключается в процессе многократного расчета случайного транспорта одного фотона в среде. Такой программный подход может быть эффективной заменой действительному облучению и дальнейшему сбору информации о распространении фотонов в многослойных биологических структурах.

Метод Монте Карло выполняет многократное повторение случайных независимых испытаний, обсчитывая сложную математическую модель. Предсказание её характеристик и поведения требует большого количества вычислительных ресурсов. Наиболее принципиальным методом повышения производительности является распараллеливание вычислений.

До недавнего времени наиболее доступными системами для параллельных расчетов были кластеры, в которых вычисления производились на центральных процессорах общего назначения (CPU, Central Processing Unit). Однако такие кластеры достаточно дороги и сложны в эксплуатации, поэтому проблема эффективных параллельных вычислений остается актуальной. С развитием индустрии трехмерной компьютерной графики и массивно-параллельных ускорителей, появились более конкурентные решения в виде графических процессоров (GPU, Graphics Processing Unit), которые специально разрабатывались как поточно-параллельные системы с большим количеством вычислительных блоков, конвейерной обработкой данных и памятью с максимальной пропускной способностью.

Безусловным лидером среди индустрии массивно-параллельных ускорителей является компания Nvidia. Эта компания предоставляет графические процессоры на базе архитектуры CUDA. Разработчикам ПО для GPU Nvidia [1] предоставляется набор инструментов CUDA Toolkit[3]. Важным фактом для программ, разработанных с применением CUDA Toolkit [4], является поддержка всей линейки видеокарт с той же архитектурой на протяжении уже нескольких лет.

Архитектура построения программ с массивно-параллельными вычислениями идейно очень похожа между разными вендорами графических ускорителей. Но, к сожалению, из-за

отсутствия единой стандартизации программно-аппаратного взаимодействия, программные реализации сильно зависимы от своего вендора и не могут быть взаимозаменяемы.

Ошибки, совершенные на этапе планирования, зачастую обходятся намного дороже в сравнение с ошибками последующих этапов. Архитекторы ПО и проектные менеджеры зачастую стоят перед сложным выбором, изучая все ограничения и сопутствующие затраты для каждой возможной архитектуры массивно параллельного ускорителя.

В попытке решить возникающие проблемы было создано несколько проектов, среди которых наиболее новым и современным является решение от компании Intel – открытый стандарт OneAPI [5], язык программирования DPC++ и сопутствующий набор библиотек для кросс платформенного программирования на вычислительных устройствах.

В данной работе проведен обзор технологий Nvidia CUDA и Intel OneAPI для программирования массивно-параллельных вычислений и описан опыт разработки решений с их применением. В частности, реализован параллельный алгоритм метода Монте Карло в задаче распространения света.

1. Постановка задачи

Длительный процесс разработки алгоритмов для массивно-параллельных вычислений и отсроченное получение конечных результатов заставляют глубоко анализировать все потребности разрабатываемого приложения и его зависимостей заранее, а также тщательно подходить к выбору технологического стека.

В рамках данной работы ставится задача изучения существующих решений и их преимуществ, а также их применение в практических задачах реализации параллельных алгоритмов на GPU. Данная задача может быть разделена на следующие подзадачи:

- 1) Провести обзор технологии Nvidia CUDA
- 2) Провести обзор технологии Intel OneAPI
- 3) Реализовать параллельный алгоритм метода Монте Карло на языке DPC++ в задаче распространения света на GPU.

2. 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 метода распространения света внутри биологических тканей основан на методах, описанных в оригинальной работе группы ученых Техасского университета в США в 1993 году [7]. Оригинальные тексты последовательной программы были опубликованы в глобальной сети Интернет.

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 [7].

Каждый слой характеризуется следующими оптическими параметрами: коэффициентом рассеяния μ_s , коэффициентом поглощения μ_a , параметром анизотропии g или фазовой функцией рассеяния $p(s, s')$, 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 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 [7].

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

$$P(l) = \frac{1}{\langle l \rangle} e^{-\frac{l}{\langle l \rangle}}$$

где средняя длина свободного пробега определяется как

$$\langle l \rangle = \frac{1}{\mu_s + \mu_a}$$

Случайная длина свободного пробега определяется в соответствии со следующей формулой:

$$l = -\ln(1 - \xi)\langle l \rangle,$$

где ξ - случайная величина, равномерно распределенная на интервале (0,1), генерируется машинным генератором случайных чисел.

Далее рассчитывается изменение направления движения фотона, при каждом акте рассеяния определяемое фазовой функцией рассеяния

$$p(s, s') = p(\theta)p(\varphi)$$

Рассеиватели обычно считаются сферически симметричными, в связи с чем, величина φ считается равномерно распределенной на отрезке $[0, 2\pi]$, а угол θ рассчитывается в соответствии с задаваемой фазовой функцией единичного рассеивателя.

Направляющие косинусы вектора скорости при рассеянии изменяются следующим образом:

$$\gamma_x = \frac{\sin(\theta)}{\sqrt{1 - \gamma_z^2}} (\gamma_x \gamma_z \cos(\varphi) - \gamma_y \sin(\varphi)) + \gamma_x \cos(\theta)$$

$$\gamma_y = \frac{\sin(\theta)}{\sqrt{1 - \gamma_z^2}} (\gamma_y \gamma_z \cos(\varphi) + \gamma_x \sin(\varphi)) + \gamma_y \cos(\theta)$$

$$\gamma_z = -\sin(\theta) \cos(\varphi) \sqrt{1 - \gamma_z^2} + \gamma_z \cos(\theta)$$

Если угол движения фотона близок к нормальному:

$$|\gamma_z| > 0.99999,$$

то изменение направляющих косинусов вычисляется по следующим формулам:

$$\gamma_x = \cos(\varphi) \sin(\theta)$$

$$\gamma_y = \sin(\varphi) \sin(\theta)$$

$$\gamma_z = \text{sign}(\gamma_z) \cos(\theta)$$

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

$$x = x_0 + l \gamma_x$$

$$y = y_0 + l \gamma_y$$

$$z = z_0 + l \gamma_z,$$

где x_0, y_0, z_0 – 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

$$P = P_0 \frac{\mu_a}{\mu_s + \mu_a},$$

где P_0 – текущий вес фотона.

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

3. Обзор методов программирования на GPU

3.1. Технология Nvidia CUDA

Архитектура CUDA и язык CUDA C

Язык программирования CUDA C является по существу языком C / C++ 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 R^3 -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 C требует разделения областей не только выполнения инструкций между CPU и GPU, но и хранения данных. С помощью новых квалификаторов функций и переменных удастся 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 R^3 . Размерность каждого подпространства из R^3 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.

Язык CUDA C предоставляет следующие механики для такого распределения задач:

// Структура, определяющая количество задействованных блоков

```
dim3 blocks(X, Y, Z);
```

```
size_t threads;
```

```
function << < blocks, threads >> > ()
```

В общем случае, можно считать, что это дополнительными параметрами функций, исполняемых на GPU.

С другой стороны, разработчику предоставляется возможность узнать тройку номеров блока, исполняющего текущий участок кода на устройстве:

```
__global__ void function()  
{  
    const int x = blockIdx.x;  
    const int y = blockIdx.y;  
    const int z = blockIdx.z;  
}
```

Это является особенностью архитектуры CUDA, которая позволяет идентифицировать каждый экземпляр функции, что может быть полезно для обработки данных.

Так, например, вычисления фрактала Julia требует точной идентификации каждой точки в пространстве. Следующая программа раскрывает эти особенности:

```
struct Image  
{  
    struct Pixel  
    {  
        unsigned char R : 8;  
        unsigned char G : 8;  
        unsigned char B : 8;  
    };  
  
    size_t width;  
    size_t height;  
  
    Pixel* data;  
  
    Image(const size_t width, const size_t height)  
        : width(width), height(height)  
    {  
        data = new Pixel[width * height];  
    }  
};
```

```

        for (size_t idx = 0; idx < width * height; ++idx)
        {
            data[idx].R = 0;
            data[idx].G = 0;
            data[idx].B = 0;
        }
    }

Image(const Image& other)
    : width(other.width), height(other.height), data(nullptr)
{
    data = new Pixel[width * height];

    std::copy(other.data, other.data + other.size(), data);
}

~Image()
{
    delete[] data;
}

size_t size() const
{
    return width * height;
}

size_t mem_size() const
{
    return size() * sizeof(Pixel);
}

bool save(const std::string filename) const
{
    CImg<unsigned char> img((unsigned int)width, (unsigned int)height, 1, 3);

    for (unsigned int x = 0; x < width; ++x)
    {
        for (unsigned int y = 0; y < height; ++y)
        {
            const unsigned int index = x * height + y;

            img.draw_point(x, y, reinterpret_cast<unsigned char*>(data + index));
        }
    }

    unsigned char pix[3] = { 255, 0, 0 };

    img.draw_point(50, 50, pix);

    img.save(filename.c_str());
}

```

```

        return true;
    }
};

__device__ cuComplex operator+(const cuComplex& a, const cuComplex& b)
{
    return cuCaddf(a, b);
}

__device__ cuComplex operator*(const cuComplex& a, const cuComplex& b)
{
    return cuCmulf(a, b);
}

__device__ float magnetude2(const cuComplex& a)
{
    return a.x * a.x + a.y * a.y;
}

__device__ int julia(const int width, const int height, const int x, const int y)
{
    const float scale = 1.5;
    const float jx = scale * static_cast<float>(width / 2 - x) / static_cast<float>(width / 2);
    const float jy = scale * static_cast<float>(height / 2 - y) / static_cast<float>(height / 2);

    cuComplex c(make_cuComplex(-0.8f, 0.156f));
    cuComplex a(make_cuComplex(jx, jy));

    for (int i = 0; i < 200; ++i)
    {
        a = a * a + c;

        if (magnetude2(a) > 1000)
        {
            return 0;
        }
    }

    return 1;
}

__global__ void kernel(const int width, const int height, Image::Pixel* data)
{
    const int x = blockIdx.x;
    const int y = blockIdx.y;

    const int index = x + y * gridDim.x;
    const int value = julia(width, height, x, y);

```

```

        data[index].R = 255U * value;
        data[index].G = 0;
        data[index].B = 0;
    }

class CudaJulia
{
    Image& target;

    const size_t& width;
    const size_t& height;

    char* dev_data;

public:
    CudaJulia(Image& target)
        : target(target), width(target.width), height(target.height), dev_data(nullptr)
    {}

    void main()
    {
        size_t size = target.mem_size();

        cudaMalloc((void**)&dev_data, size);

        dim3 grid(width, height);

        kernel<<<grid,1>>>(width, height, reinterpret_cast<Image::Pixel*>(dev_data));

        cudaMemcpy(reinterpret_cast<char*>(target.data), dev_data, size,
cudaMemcpyDeviceToHost);

        cudaFree(dev_data);

        target.save("bitmap.bmp");
    }
};

int main()
{
    Image img(1000, 1000);
    CudaJulia gpu_coder(img);

    gpu_coder.main();

    return 0;
}

```


Данная программа выполняет итеративные вычисления для каждого пикселя не более 200 раз. С учетом размера изображения (1000 x 1000 точек) становится очевидным, что общее количество вычислений слишком велико, чтобы последовательный алгоритм сделал это так же незаметно быстро, как его параллельная реализация. Стоит добавить, что общее время выполнения программы длится не более полутора секунд и большая часть времени задействована в последовательном сохранении каждого пикселя в файл.

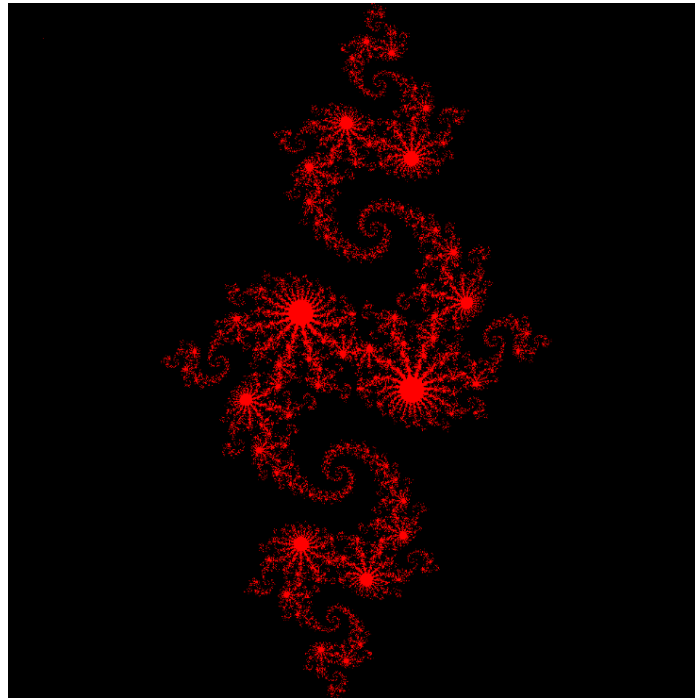


Рис. 1. Результат вычисления фрактала Julia.

Средства синхронизации

Параллелизм выполнения задач нуждается в примитивах синхронизации. Такие механизмы заложены и в архитектуре CUDA. Следующая процедура вызывается на устройстве:

```
void __syncthreads(void)
```

Каждый поток из одного блока, который вызвал эту процедуру, переходит в режим ожидания перед выполнением следующих функций. Ожидание продолжается до тех пор, пока оставшиеся потоки не вызовут эту процедуру.

Так, например, следующий пример кода выполняет многонитевое (многопоточное) сложение компонент вектора таким известным приёмом, как редукция:

```
__shared__ float cache[threadPerBlock];
```

```

int cacheIndex = threadIdx.x;

int i = blockDim.x / 2;
while (i != 0)
{
    if (cacheIndex < i)
        cache[cacheIndex] += cache[cacheIndex + i];

    __syncthreads();

    i /= 2;
}

if (cacheIndex == 0)
    c[blockIdx.x] = cache[0];

```

За каждую компоненту вектора отвечает свой блок. Значение каждой компоненты будет вычислено множеством потоков. Значения будут поочередно суммироваться потоками, записывая промежуточные вычисления в разделяемую память блока. В результате выполнения нулевой поток запишет значение в соответствующую компоненту.

3.2. 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 [8]. 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 CPU, 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.

```
int main()
{
    // host code

    sycl::queue gpu_queue{ sycl::gpu_selector{} };
    sycl::queue cpu_queue{ sycl::cpu_selector{} };

    gpu_queue.submit([&](C) { /* device code for GPU */ });
    cpu_queue.submit([&](C) { /* device code for CPU */ });

    // host code
}
```

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.

Лямбда функция `kernel` захватывает все данные, копируя их (на `device`), что очень органично вписывается в концепцию стандарта C++:

```
gpu_queue.submit(
    [&](sycl::handler& cgh)
    {
        cgh.parallel_for<class Add>(sycl::range<1>(N), [=](sycl::id<1> index) { /* Kernel code */ });
    }
);
```

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 выполняется.

```

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 C++ 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)` с 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, разрушаются => 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);
```

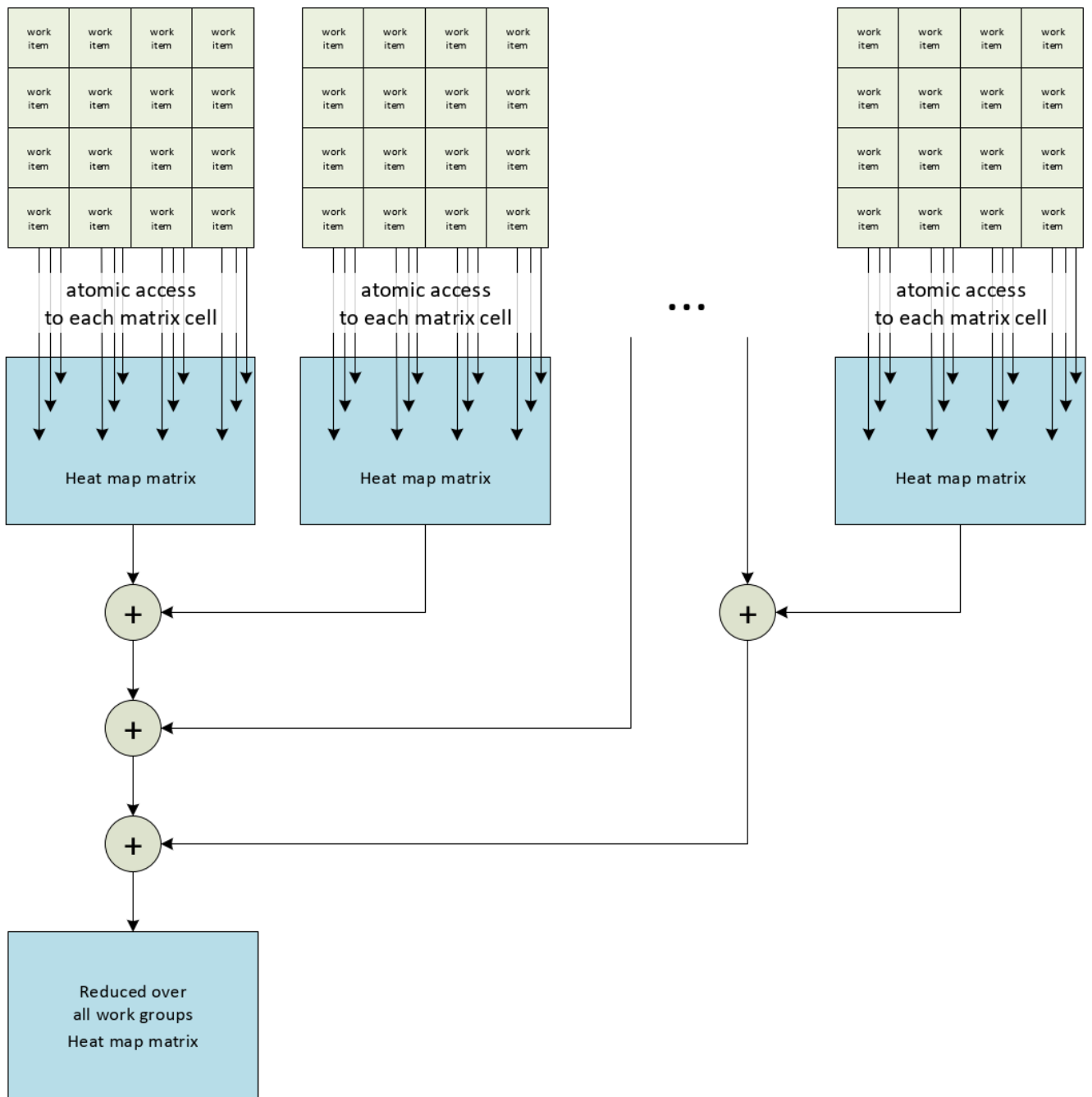
Таким образом данные могут храниться внутри *work-group* с более быстрым и защищенным (от других *work-group*) доступом к ним.

4. 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. При обновлении значений предполагается эксклюзивный доступ для чтения и записи значений. С увеличением количества вычислителей неизбежно растет время ожидания такого доступа к общему ресурсу, поэтому общий прирост производительности снижается. 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.

5. 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. 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.

6. 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 $\text{mod}(x, y)$, возвращающая остаток от деления первого аргумента на второй. Каждое последующее случайное число рассчитывается на основе предыдущего по следующей формуле:

$$r_{i+1} = \text{mod}(k * r_i + b, M)$$

где:

- M – модуль ($M > 0$);
- k – множитель ($0 \leq k < M$)
- b – приращение ($0 \leq b < M$)
- r_0 – начальное значение ($0 \leq r_0 < M$)

Последовательность случайных чисел, полученных с помощью данной формулы, называется линейной конгруэнтной последовательностью. Линейную конгруэнтную последовательность при $b = 0$ называют мультипликативным конгруэнтным методом, а при $b \neq 0$ — 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 $M = 2^N$, поскольку в этом случае нахождение остатка от деления сводится внутри процессора к двоичной логической операции AND. Также широко распространен выбор наибольшего

простого числа M , меньшего, чем 2^N . Можно доказать, что в этом случае младшие разряды получаемого случайного числа r_{i+1} 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 $2^{31} - 1$, и таким образом, константа определяется как $M = 2^{31} - 1$.

Одним из требований к линейным конгруэнтным последовательностям является как можно большая длина периода. Длина периода зависит от значений M , k и b . Теорема, которая приведена ниже, позволяет определить, возможно ли достижение периода максимальной длины для конкретных значений M , k и b .

Теорема. Линейная конгруэнтная последовательность, определенная числами M , k , b и r_0 , имеет период длиной M тогда и только тогда, когда:

- числа b и M взаимно простые;
- $k - 1$ кратно p для каждого простого p , являющегося делителем M ;
- $k - 1$ кратно 4, если M 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 M выбирается число 2^{59} . 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;
}
};

```

7. 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*.

С учетом вышеуказанной архитектуры запуск *kernel* будет выглядеть следующим образом:

```
sycl::queue q(d_selector, exception_handler);  
sycl_host_matrix_allocator<float> allocator(q);  
  
constexpr size_t N_x = 5;  
constexpr size_t N_y = 5;  
constexpr size_t N_z = 5;  
  
constexpr size_t N_l = 1;  
  
constexpr size_t N_repeats = 0.25 * 1000 / 10;  
  
constexpr size_t work_group_size = 64;  
constexpr size_t num_groups = 128;
```

```

std::cout << "N_x dimensions of x: " << N_x << std::endl;
std::cout << "N_y dimensions of y: " << N_y << std::endl;
std::cout << "N_z dimensions of z: " << N_z << std::endl;
std::cout << "N_l count of layers: " << N_l << std::endl;
std::cout << "  Repeats per tread: " << N_repeats << std::endl;
std::cout << "  Work group size: " << work_group_size << std::endl;
std::cout << "  Number of groups: " << num_groups << std::endl;

host_matrix_view<float> host_view(N_x, N_y, N_z, N_l, allocator);

size_t N = host_view.properties().length();

float* device_data = sycl::malloc_device<float>(N, q);
float* device_group_data_pool = sycl::malloc_device<float>(N * num_groups, q);

InputStruct input;

input.layerspecs = sycl::malloc_shared<LayerStruct>(N_l, q);

configure_input(input);
configure(input.layerspecs, q);

q.parallel_for(num_groups,
    [=](auto group_index)
    {
        float* data = device_group_data_pool + N * group_index;

        for (int i = 0; i < N; ++i)
        {
            data[i] = 0;
        }
    });

q.submit(
    [&](sycl::handler& h)
    {
        sycl::stream output(1024, 256, h);

        h.parallel_for_work_group<class PhotonKernel>(sycl::range<1>(num_groups),
            sycl::range<1>(work_group_size),
            [=](sycl::group<1> g)
            {
                size_t gid = g.get_group_id(0); // work group index

                // select from allocated memory on device
                float* data = device_group_data_pool + N * gid;

                // float local_mem[N_x * N_y * N_z * N_l]{ 0.0 };

                g.parallel_for_work_item(
                    [&](sycl::h_item<1> item)
                    {
                        uint64_t thread_global_id = item.get_global_id();
                        mcg59_t random_generator(42, thread_global_id, work_group_size * num_groups);

                        raw_memory_matrix_view<float> view(data, N_x, N_y, N_z, N_l);
                        // raw_memory_matrix_view<float> view(local_mem, N_x, N_y, N_z, N_l);

                        PhotonStruct photon(random_generator, view, input, input.layerspecs);

                        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);
                        }
                    }
                );
            }
        );
    }
);

```



```

        });
    });
q.wait();
q.parallel_for(N,
    [=](auto idx)
    {
        int summ = 0;

        for (size_t gid = 0; gid < num_groups; ++gid)
        {
            float* data = device_group_data_pool + N * gid;

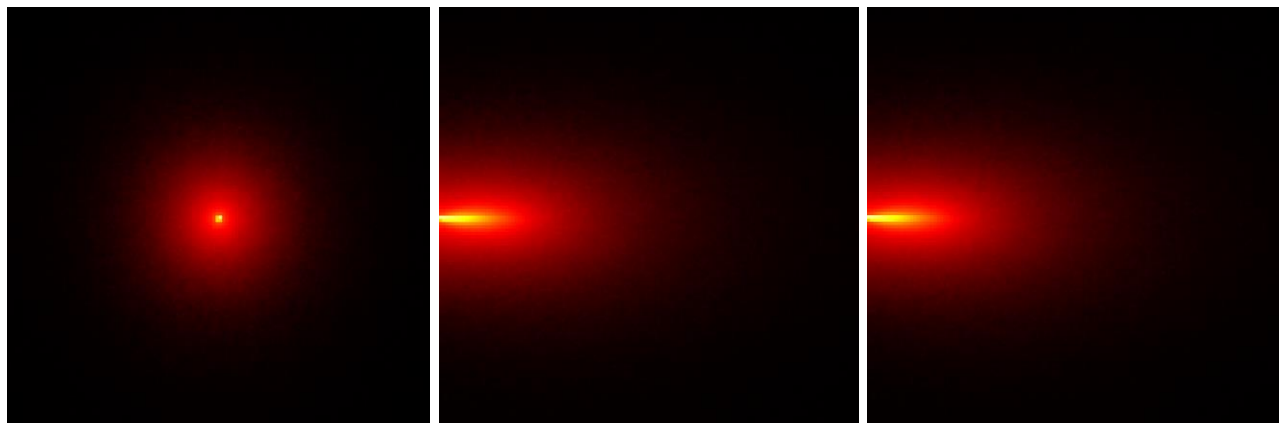
            summ += data[idx];
        }

        device_data[idx] = summ;
    });
q.wait();
q.memcpy(host_view.data(), device_data, host_view.size_of_data());
q.wait();

```

8. Результаты моделирования

В качестве интерпретатора результатов моделирования используется дополнительно разработанное приложение для визуализации тепловой карты потерянного веса фотонов. Данное приложение предоставляет двумерное изображение 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.

Ниже приведены результаты экспериментов разработки последовательного алгоритма для CPU.

Характеристики испытательного стенда:

Операционная система: Windows 11, 64-bit.

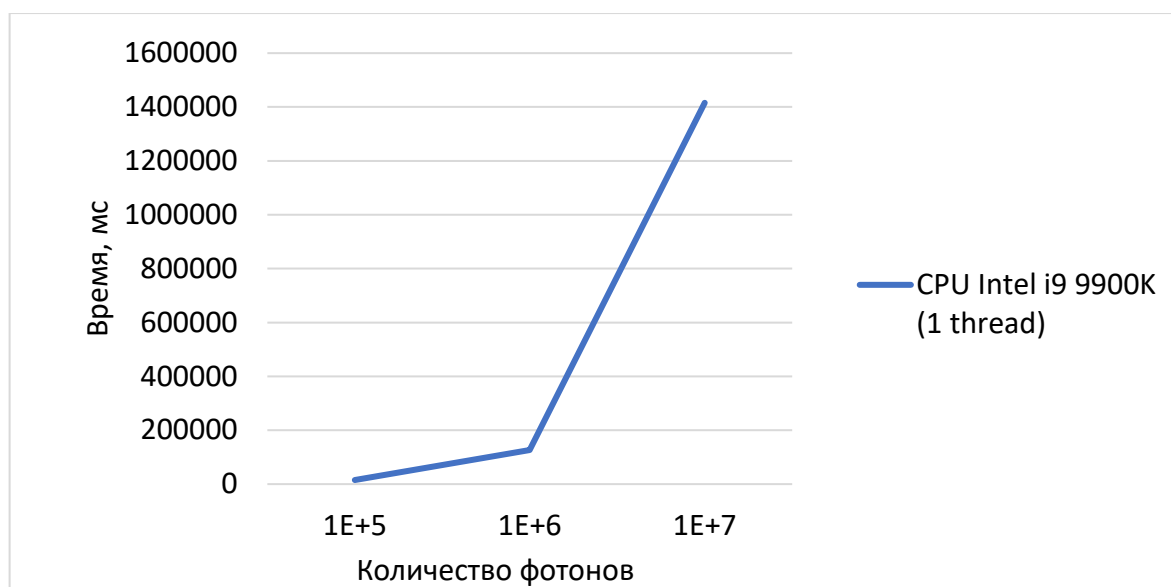
Процессор: Intel Core i9 9900K 4.48 GHz.

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



На графике можно заметить, что время моделирования испытания напрямую зависит от количества фотонов.

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

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.

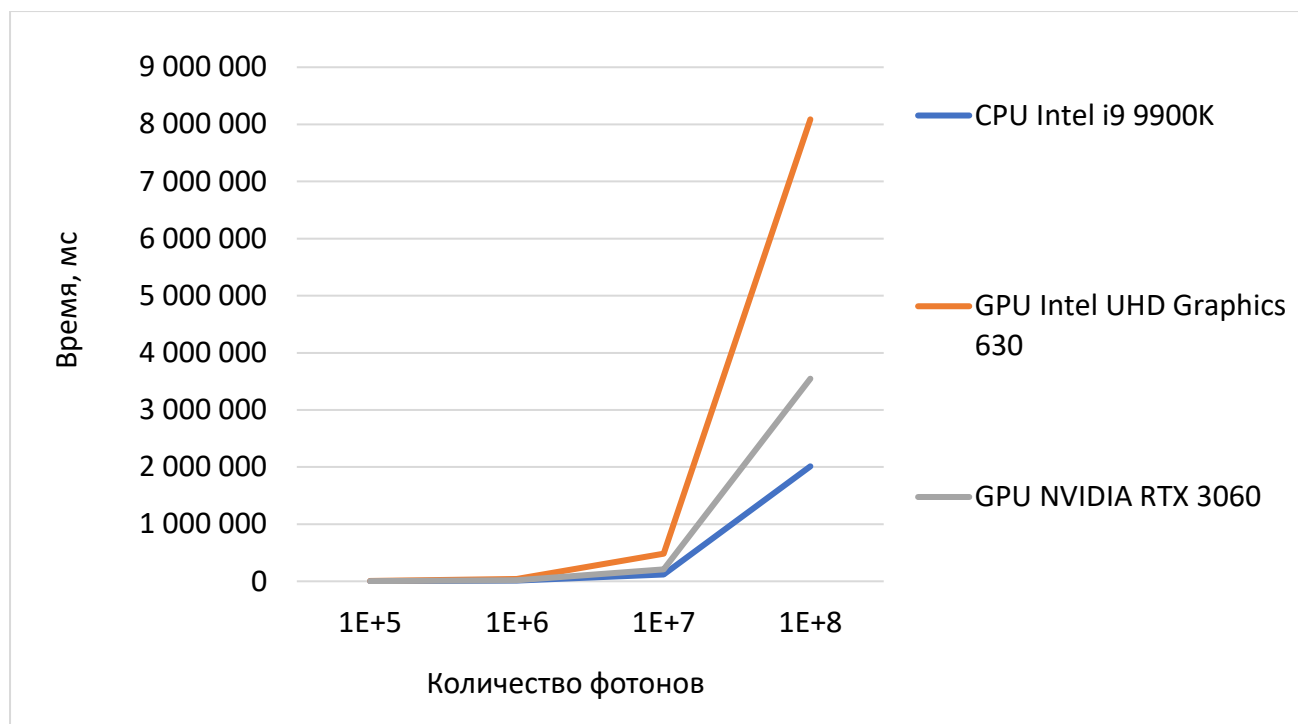
Объем оперативной памяти: 24GB.

Графический ускоритель: Intel UHD Graphics 630.

Графический ускоритель: NVIDIA RTX 3060 12GB.

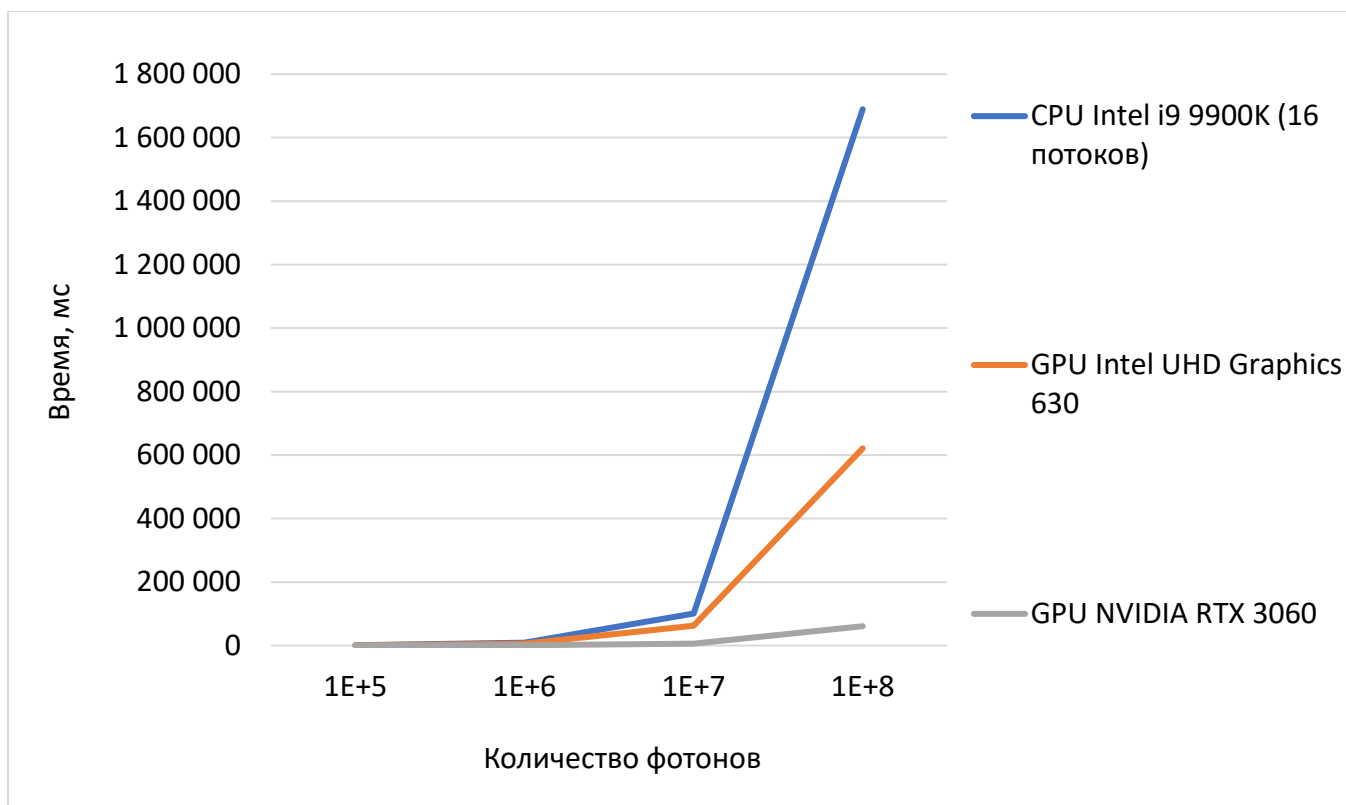
Ниже приведена таблица с результатами экспериментов до выполнения оптимизаций.

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



После выполнения вышеописанных оптимизаций было выполнено повторное сравнение производительности. Результаты представлены в таблице ниже.

Вычислительное устройство	Число фотонов			
	100 000	1 000 000	10 000 000	100 000 000
CPU Intel i9 9900K (16 потоков)	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



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

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 на GPU, и составляет более 220 раз относительно неоптимизированной последовательной версии для CPU, что говорит о значительном улучшении.

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

Литература

1. Reinders J., Ashbaugh B., Brodman J.: Data Parallel C++. Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL. – 2021 г. – 548 с.
2. NVIDIA Corporation: CUDA C Programming Guide – 2012 г. – 185с.
3. Сандерс Д., Кэндрот Э.: Технология CUDA в примерах. Введение в программирование графических процессов – 2018 г. – 232 с.
4. Боресков А. В., Харламов А.: Основы работы с технологией CUDA – 2010 г. – 231с.
5. Intel OneAPI – официальный сайт: <https://www.intel.com/oneapi/overview.html>
6. 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.
7. Wang L.V., Jacques S.L., Zheng L.Q. MCML – Monte Carlo modeling of light transport in multi-layered – 1995 г. – 186 с.
8. OpenCL – официальный сайт: <http://www.khronos.org/opencl/>
9. 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.
10. 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