Санкт–Петербургский государственный университет

МАКОЕВ Артур Аланович

Выпускная квалификационная работа

Моделирование распространения звука в замкнутом помещении с использованием вычислений на видеокарте

Уровень образования: бакалавриат

Направление: 02.03.02 «Фундаментальная информатика и информационные технологии

Основная образовательная программа СВ.5003.2017 «Программирование и информационные технологии»

Профиль «Автоматизация научных исследований»

Научный руководитель:
доцент, Кафедра компьютерного моделирования и многопроцессорных систем,
к.ф.-м.н. Ганкевич Иван Геннадьевич

Рецензент:
разработчик программного обеспечения,
Закрытое акционерное общество «БИОКАД»
Фатькина Анна Игоревна

Санкт-Петербург
2020 г.

Содержание

Введение

1. Актуальность темы

Задача моделирования акустики помещений актуальна при проектировании зданий и сооружений, особенно для концертных и лекционных залов. Также, с развитием технологий виртуальной реальности и с увеличивающийся реалистичностью виртуальных пространств растет потребность в качественном интерактивном звуке. В то же время растет интерес к использованию более мощных звуковых чипов с широкими возможностями в потребительской электронике. Так, например, компания Sony выпустила игровую приставку PlayStation 5 со звуковым чипом собственной разработки Tempest Engine, который осуществляет аппаратное ускорение в операциях со звуком, таких как применение свёрточной реверберации и HRTF (Head-related transfer function) – функций. Интерес представляет архитектура данного чипа, который состоит из множества SPU (Synergistic Processing Unit) – векторных со-процессоров в архитектуре Cell (Cell Broadband Engine Architecture), применявшихся в PlayStation 3. SPU является многоядерным векторным процессором класса SIMD (single instruction, multiple data), то есть обеспечивает параллелизм на уровне данных, что роднит принцип его работы с работой GPU. Данный факт показывает, что в настоящее время высокопроизводительные параллельные вычисления на процессорах с GPU-подобной архитектурой становятся актуальными не только для задач моделирования распространения звука с помощью точных, но не интерактивных методов, но и для интерактивных решений, работающих в реальном времени.

2. Цель работы

Цель работы – изучение потенциала использования вычислений на видеокарте для эффективного ускорения алгоритмов моделирования акустики помещений. Основными задачами являются

- изучение существующих методов решения задачи моделирования акустики,
- реализация прототипов программ для наиболее популярных методов,
- формулирование предложений по улучшению существующих методов в виде нового метода для достижения перцептивно реалистичной реверберации в помещении в реальном времени.

Входными данными для алгоритма являются положение источника звука, положение и ориентация слушателя, геометрия помещения. На выходе алгоритм представляет импульсную характеристику помещения.

Постановка задачи

Определение: Функция Дирака (дельта-функция) – функция $\delta(x)$, удовлетворяющая следующим условиям:

$$\delta(x) = \begin{cases} +\infty, x = 0 \\ 0, x \neq 0 \end{cases}, \quad (1)$$

$$\int_{-\infty}^{\infty} \delta(x)dx = 1. \quad (2)$$

Дельта-функция также обладает важным свойством (фильтрующее свойство):

$$\int_{-\infty}^{\infty} \delta(x-y)f(x)dx = f(y). \quad (3)$$

Для дискретной сетки с шагом h, функция Дирака аппроксимируется как

$$\delta(x) = \begin{cases} \frac{1}{h}, x = 0 \\ 0, x \neq 0 \end{cases}. \quad (4)$$

Определение: Импульсная переходная функция (импульсная характеристика) $h(t)$ – выходной сигнал динамической системы, как реакция на входной сигнал в виде дельта-функции Дирака.

Главное свойство импульсной характеристики, это то, что для линейных систем выходной сигнал связан с входным через выражение

$$y(t) \int_{-\infty}^{\infty} h(\tau)x(t-\tau)d\tau\,, \tag{5}$$

то есть через операцию свертки входного сигнала и импульсной характеристики.

Задачу моделирования акустики помещения можно задать так: Найти импульсную переходную функцию для системы из источника сигнала, приемника сигнала, среды (воздуха), в которой выполняется волновое уравнение, и помещения, на границе стен со средой которого выполняются граничные условия.

Данная система является линейной и стационарной, так как входящее в нее волновое уравнение является линейным дифференциальным уравнением в частных производных и положение источника и приемника сигнала не меняется во времени.

Волновое уравнение имеет вид

$$\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} - v^2\Delta u = f(x,t), \qquad \Delta u = \sum_{i=1}^{n} \frac{\partial^2 u}{\partial {x_i}^2}, \tag{6}$$

где

n – размерность,

Δu – оператор Лапласа,

v – скорость распространения волны, константная в однородной среде,

$b \in [0,1]$ – коэффициент затухания,

$f(x,t)$ – функция внешнего воздействия на систему, для точечного источника она равна $f(x,t) = f(t) \cdot \delta(x - x_0)$, то есть излучение происходит из одной точки.

Также заданы граничные условия вида Дирихле или Неймана

$u|_{\delta\Omega} = g(x)$ – условие на значение функции u на границе области

$\left.\frac{\partial u}{\partial n}\right|_{\delta\Omega} = g(x)$ – условие на значение производной функции u вдоль нормали на границе области

Импульсная характеристика находится из уравнения как $h(x_0, t) = u(x_0, t)$, при условии, что $f(x, t) = \delta(x - x_{source})$.

После нахождения импульсной характеристики системы мы можем эффективно вычислять выходной сигнал от входного только с помощью операции свертки входного сигнала с импульсной характеристикой системы. Эта импульсная переходная характеристика будет верна только для заданных положений источника и приемника, и для заданной геометрии помещения. Так как импульсная характеристика является реакцией системы на входной сигнал в виде дельта-функции Дирака, то получить её для реального помещения можно, записав звук громкого хлопка в помещении, либо воспроизвести звук возрастающей по частоте синусоидальной волны (sin sweep tone), а затем произвести деконволюцию полученного сигнала.

1. Обзор литературы

В настоящий момент, в большинстве интерактивных виртуальных пространств акустические свойства пространства задаются вручную дизайнером и являются статическими в своих секторах. Каждой зоне пространства присваивается импульсная характеристика, не зависящая от положения слушателя и реальной геометрии пространства. В данном разделе я рассматриваю различные варианты методов, применяемых для решения задачи моделирования акустики помещения.

1.1. Метод конечных разностей и метод конечных элементов

Рассмотрим метод конечных разностей. Зададим в пространстве сетку, на узлах которой будем хранить значения функции $u(x, t)$. По дифференциальному уравнению составим разностную схему, по которой можно вычислять значения функции в следующую итерацию в узле сетки (x_i, y_j) независимо от вычислений в соседних узлах (в пределах итерации) [1].

Разностные схемы строятся путем замены частных производных на дискретные выражения вида:

$$\frac{dy}{dx} = \frac{y(x_{i+1}) - y(x_{i-1})}{2h} + O(h^2), \tag{7}$$

$$\frac{\partial^2 y}{\partial x^2} = \frac{y(x_{i-1}) - 2y(x_i) + y(x_{i+1})}{h^2} + O(h^2). \tag{8}$$

В нашем случае мы рассматриваем линейную систему, что позволяет явно выразить из уравнения неизвестное значение в следующий момент времени как функцию нескольких текущих и предыдущих значений.

$$u(x_i, t_{n+1}) = F\big(x_i, \dots, x_j, t_n, t_{n-1}\big). \tag{9}$$

Этот метод легко распараллеливается, так как значение $u(x_i, t_{n+1})$ для следующей итерации алгоритма можно вычислять независимо от соседних значений на следующей итерации.

Метод конечных элементов разделяет пространство на конечное число маленьких треугольных или четырехугольных участков, внутри которых значения функции $u(x,t)$ аппроксимируются в виде параметрической функции $f(a,x)$, при этом на значения на границах областей задается уравнение, приравнивающее значения аппроксимирующих функций на общих границах. В итоге значения для следующей итерации оказываются связаны между собой через линейную систему алгебраических уравнений. Для нахождения этих значений надо решить систему с матрицами размера порядка $n \times n$, где n – число узлов, которое в случае двумерной или трехмерной задачи может быть велико. Облегчает вычисления тот факт, что матрица в большинстве случаев – разреженная, что экономит память и вычисления, но требует особых алгоритмов решения. Преимущество данного метода перед методом конечных разностей в его более высокой точности, и более точных граничных условиях, позволяющих, например задать нормаль к поверхности, на которой задано граничное условие, чего нельзя сделать на

сетке. Метод конечных элементов тоже можно распараллелить, но он более вычислительно сложный чем метод конечных разностей.

1.2. Метод мнимых источников (метод отражений)

Метод отражений – геометрический метод решения волнового уравнения, основанный на том, что волновой фронт (место точек с одинаковой фазой) двигается в однородной среде с постоянной скоростью, и отражается от идеально гладкой поверхности под углом, равным углу падения. Если геометрия помещения состоит из кусочно-плоских поверхностей, то для каждой грани можно рассчитать подпространство, из которого будет виден отраженный источник, и его координаты в отраженном пространстве, то есть, как если бы луч от источника не отразился от грани, а источник просто находился по другую её сторону. Добавим в этом положении источник, который будет уже называться «мнимым», и который виден из найденного подпространства. Такая операция проводится для всех граней, а затем рекурсивно и для всех мнимых источников. Каждый новый добавленный мнимый источник хранит в себе номер источника родителя и количество отражений. Также можно хранить и другие данные, например коэффициенты отражательной способности грани, относительно которой построен источник. В данном методе можно отделить расчёт положений мнимых источников от заданного положения приемника сигнала. В этом случае для приемника рассчитывается видимость мнимых источников из его позиции. По списку видимых источников строится импульсная характеристика, путем нанесения значений функции на $h(x) = f(source, r, v),\ x = r/v$. Если известны акустические характеристики грани, то вместо одного значения в точке $x = r/v$, можно наносить некое заранее известное ядро свертки.

Недостаток данного метода в том, что в простом варианте он не учитывает явления дифракции звука, которое в отличие от света, в звуке выражено сильнее и имеет весомое значение для перцептивного восприятия человеком. Так как каждая точка грани образует новый фронт волны, то на ребрах волна отражается всенаправленно, огибая углы. Для двумерного случая, в вершины углов добавляются мнимые источники, излучающие в пределах подпространства, образованного данным углом и хранящие в себе информацию об общей длине пути луча, которое он прошел, до того как попал в этот угол (для мнимых источников без дифракции нет смысла хранить это значение, так как длина общего пути луча от источника до приемника будет равна длине от приемника до мнимого источника). Для трехмерного случая при дифракции от ребра эти мнимые источники перестают быть точечными и становятся отрезками. Обычно, в алгоритмах допускают не больше фиксированного количества дифракционных мнимых источников в цепочке. Также недостатком метода является то, что границы не могут быть в виде кривых, так как от кривой нельзя построить точечный мнимый источник.

Примером реализации метода, поддерживающим дифракцию, является работа **Implementation and visualization of edge diffraction with image-source method**. **[2]**

1.3. Метод трассировки лучей

Стохастический метод трассировки лучей при стремящимся к бесконечности числу лучей сходится к стабильному решению, не учитывающему эффектов дифракции. Дифракцию можно аппроксимировать, меняя траекторию лучей вблизи ребер. Преимущество перед предыдущим методом в том, что позволяет учитывать поверхности с произвольной отражательной способностью, описываемой BRDF (двулучевой функции

отражательной способности), а не только идеально зеркальные, что позволяет получать диффузные и спекулярные отражения от материалов с разными параметрами шероховатости. Это делает его реалистичнее чем предыдущий метод. Он лучше всего работает для высоких частот, так как там эффекты дифракции наименее выражены. Этот метод хорошо изучен для задач оптического рендеринга, но в задаче акустики появляется дополнительное измерение – время. В качестве примера реализации метода я рассматривал статью **Interactive Sound Propagation with Bidirectional Path Tracing** [3], в которой используется двунаправленный поиск пути, одновременно от источника и от приемника.

Отраженный звук разделяют на ранние и поздние отражения по времени прошедшему до того, как они достигнут слушателя. Считается, что ранние отражения улавливаются человеком достаточно точно, именно они позволяют ему определить расстояния до стен и геометрию пространства. Громкость и длительность поздних отражений зависит от объема помещения и материалов стен. Поздние отражения тоже улавливаются человеком, но к этому моменту путь звука становится слишком сложным, чтобы можно было разобрать его точный путь с помощью органов чувств, только его общие параметры. Так как для поздних отражений звука количество отражений от поверхностей на пути луча велико, то для его вычисления нецелесообразно использовать метод трассировки лучей. Метод трассировки лучей хорошо подходит для вычисления ранних отражений. Для поздних отражений можно использовать метод **Acoustic Radiosity Transfer** [4] (метод переноса энергии), который менее точен, чем трассировка лучей, так как все поверхности в нем отражаются диффузно, то есть во все стороны, но для поздних отражений это имеет меньшее значение, так как лучи и так уже сильно рассеяны в пространстве и человек не способен точно их различать.

Существует программный пакет Odeon [5], включающий в себя геометрические методы моделирования акустики, такие как метод трассировки лучей, метод отражений, метод переноса энергии.

1.4. Efficient and accurate sound propagation using adaptive rectangular decomposition

Данный метод делит пространство на множество прямоугольных параллелепипедов (областей, доменов) размеров (l_x, l_y, l_z), в каждом из которых поле давления воздуха выражается через собственные функции оператора Лапласа $f: \Delta f + \lambda f = 0$, которая в прямоугольной области выражается через базисные функции вида Ф:

$$p(x, y, z, t) = \sum_{i=(i_x, i_y, i_z)} m_i(t)\, \Phi_i(x, y, z), \tag{10}$$

$$\Phi_i(x, y, z) = \cos\left(\frac{\pi i_x}{l_x} x\right) \cos\left(\frac{\pi i_y}{l_y} y\right) \cos\left(\frac{\pi i_z}{l_z} z\right), \tag{11}$$

где $m_i(t)$ – неизвестные модальные коэффициенты.

Первое уравнение по сути является Обратным Дискретным Косинусным Преобразованием (iDCT), поэтому оно имеет вид $P(t) = iDCT(M(t))$. Начальные значения M высчитываются через начальные значения $M(0) = DCT(P(0))$.

Так как мы используем сумму собственных функций оператора Лапласа, то можно перейти к волновому уравнению над значениями M:

$$\frac{\partial^2 M_i}{\partial t^2} + c^2 {k_i}^2 M_i = iDCT\big(F(t)\big), \tag{12}$$

$${k_i}^2 = \pi^2 \left(\frac{{i_x}^2}{{l_x}^2} + \frac{{i_y}^2}{{l_y}^2} + \frac{{i_z}^2}{{l_z}^2}\right). \tag{13}$$

Внешнее воздействие моделируется через

$$\tilde{F}(t) \equiv DCT\big(F(t)\big). \qquad (14)$$

Выражается правило вычисления значений M:

$$M_i^{n+1} = 2M_i^n \cos(\omega_i \Delta t) - M_i^{n-1} + 2\frac{\widetilde{F^n}}{{\omega_i}^2}(1 - \cos(\omega_i \Delta t)), \qquad \omega_i = ck_i. \quad (15)$$

Таким образом, задача сводится к вычислению модальных коэффициентов на следующие итерации, а решение волнового уравнения в этой прямоугольной области ищется на сетке с частотой Найквиста.

То есть распространение волн через область не накапливает вычислительной ошибки, в отличие от метода конечных разностей. Значения волновой функции на границе области переносится как граничное условие в смежную область. Подробнее в оригинальной статье [6]. Для самой области ее граница является гладкой плоскостью, если она соответствует стене, или PML (perfectly matched layer) – идеально согласованному слою, поглощающему входящие волны, моделируя таким образом открытое пространство [7].

В оригинальной статье авторы проводили сравнения своего метода с методом конечных разностей, и показали, что распространение волны, вызванной импульсом дельта-функцией на 15 метров по прямой при одинаковом шаге сетки в их методе практически не меняло форму волны, в то время как метод разностей уже давал сильное искажение. Даже метод разностей, использующий сетку в 2,5 раз более плотную по каждой оси, не избавился от искажения полностью. Вычисление предложенным методом заняло 31 минуту и 0,5 ГБ памяти, методом конечных разностей (FDTD, finite-difference time-domain) заняло 11 минут и 0,4 ГБ памяти, а методом FDTD с в 2,5 раза более плотной сеткой заняло 356 минут и 6,1 ГБ памяти.

Так как области независимы, то в пределах общей симуляции можно комбинировать области с разными методами вычисления итераций, например, предложенный точный метод на прямоугольных областях для открытых пространств внутри помещения, и метод конечных разностей для областей со сложными объектами внутри. Также это позволяет делать параллелизм на уровне областей. Этот метод также подходит для вычисления на видеокарте, что продемонстрировано в работе **An efficient GPU-based time domain solver for the acoustic wave equation** [8].

1.5. Project Triton и Planeverb

Два рассматриваемых метода Project Triton [9] и Planeverb [10] были разработаны и используются компанией Microsoft. В отличие от предыдущих методов, в них результат расчета поведения волн не используется напрямую. В них расчет импульсной характеристики для данных позиций источника и приемника происходит в реальном времени в интерактивном режиме, и он отделен этапа вычислений импульсных характеристик для пар точек источник-приемник.

Рассмотрим Project Triton. Первый этап – в помещении расставляются точки, в которых будут источник и приемник. Для импульсных характеристик не имеет значения, в данной паре точек является ли первая источником или приемником, они взаимозаменяемые. Для каждой неупорядоченной пары вычисляются импульсные характеристики по алгоритму из статьи [6], а затем для каждой пары считаются и сохраняются наборы из 10 параметров, описывающие перцептивные свойства звука, такие как задержка основного звука, направление и громкость, время затухания, задержка ранних отражений, громкость реверберации с 6 сторон.

Во время непосредственно второго этапа работы алгоритма в реальном времени, данные значения линейно интерполируются между точками, и в

реальном времени генерируется импульсная характеристика с помощью канонических (оптимальных) фильтров и шумов Гаусса. Данная импульсная характеристика численно отличается от вычисленной на предыдущем этапе, но совпадает с ней по перцептивным параметрам.

Метод PlaneVerb состоит из тех же этапов – решение волнового уравнения, кодирования параметров и восстановления импульсной характеристики, но в данном случае первый этап проходит в реальном времени на компьютере пользователя с помощью алгоритма FDTD для двумерного пространства в радиусе 25 метров с частотой 275 Гц, d_x = 0,36 м. за менее чем 100 мс. Преимущество в сравнении с Project Triton – работает с динамичными сценами, а недостаток – пространство двумерно, и низкочастотные эффекты дифракции переносятся на высокие частоты.

1.6. NVIDIA VRWorks Audio и AMD TrueAudio Next

Компании NVIDIA и AMD представили свои технологии для работы со звуком на GPU, ускоряющие операции применения IR (Impulse response – импульсная характеристика), HRTF, FCT (Fourier cosine transform – косинусное преобразование Фурье), а также позволяющие использовать методы геометрической акустики. К сожалению, проект NVIDIA VRWorks [11] недоступен для тестирования, все ссылки на скачивания SDK и GitHub проекта с официальной страницы не работают. Технология AMD TrueAudio [12] вошла в состав программы Steam Audio. Эта технология позволяет ускорять операции со звуком на видеокартах AMD Radeon, а также использовать видеокарту для вычисления импульсной характеристики сцены геометрическим методом.

2. Модель программирования CUDA ядра

Рассмотренные в предыдущем разделе методы решения задачи подходят для эффективного распараллеливания в силу принципа работы. Метод конечных разностей можно применять параллельно для каждого

элемента сетки, а в методах, основанных на трассировке лучей, каждый луч является независимым. В данном разделе мы рассмотрим архитектуру модели вычислений на видеокарте.

С точки зрения использования видеокарты как вычислительной платформы, существуют следующие инструменты для выполнения вычислений общей направленности на видеокарте – general purpose computing on graphics processing units (GPGPU):

- Compute Shaders – использование программируемых шейдеров для неграфических задач. Расширение обычных графических шейдеров (программ, исполняемых на GPU, предназначенных для создания графических эффектов), позволяющая применять шейдеры за пределами графического пайплайна, на произвольном массиве данных. Вычислительные шейдеры поддерживаются стандартом OpenGL выше 4.3 на большом количестве видеокарт от разных производителей на платформах Windows, Linux, Mac OS. Для операционной системы Windows доступен API DirectCompute, входящий в состав DirectX и реализующий вычислительные шейдеры.
- OpenCL – фреймворк для написания параллельных алгоритмов, работающих на CPU и GPU. Является открытым стандартом и поддерживает видеокарты всех производителей.
- CUDA – платформа для параллельным вычислений, созданная компанией Nvidia, поддерживающая только видеокарты компании.

Для целей своей работы я выбрал платформу CUDA, потому что она легче в написании алгоритмов, обладает удобными инструментами разработки и немного выигрывает в производительности у OpenCL. В случае создания полноценного программного продукта есть смысл в портировании кода с архитектуры CUDA в код для выполнения в вычислительных шейдерах.

Параллельные вычисления на CPU позволяют запускать на разных ядрах процессора независимый код, что позволяет эффективно параллельно исполнять отдельные части сложного алгоритма, обмениваться данными между частями алгоритма, но не дает преимущества при исполнении одного и того же кода для различных данных.

В отличие от параллельных нитей выполнения программы для CPU, созданной, например, с помощью библиотеки OpenMP, нити программы, запускаемой на GPU, логически связаны друг с другом, и выполняются группами, называемыми в CUDA Warp. Одна группа в CUDA объединяет 32 нити выполнения. Внутри одной группы выполняются одни и те же инструкции на различных данных. Если в коде встречается ветвление, то инструкции выполняются на всех нитях, только результаты выполнения для неактивных нитей не будут сохранятся.

Группы объединяются в блоки размера до 1024 нитей, которые в свою очередь выполняются в сетке (grid) заданных размеров. Порядок выполнения нитей в блоке и порядок выполнения блоков в сетке является недетерминированным. Нити внутри блока, а также блоки внутри сетки могут располагаться в одномерном, двумерном, или трехмерном логическом массиве, то есть внутри кода ядра (Kernel) доступны одно-, дву-, или трехмерные индексы для нитей в блоке и блока в сетке. Синхронизация нитей внутри кода возможна только в пределах блока, потому что блоки в сетке выполняются хоть и параллельно, но не все разом. Синхронизация в блоке происходит с помощью команды __syncthreads().

Память видеокарты доступна как для чтения, так и записи, но мало пригодна для обмена данными между нитями. Эффективный обмен данных между нитями доступен в пределах блока в виде общей памяти блока (shared memory), ограниченного размера порядка 16-96kB. Обычно в пределах одного ядра выполнения алгоритм разделен на этапы, между которыми происходят синхронизация в блоке. Данные, необходимые на текущем этапе, загружаются в общую память на предыдущем этапе. Архитектура

вычислений предполагает (но не навязывает) прямое соответствие между нитями и областью в памяти, то есть для одной нити предполагается вычисление одного элемента. Поэтому рекомендуется выбирать размерность блоков и сетки в соответствии с размерностью задачи, при этом доступ к памяти желательно должен быть выравнен по отношению к памяти, иначе скорость доступа к данным резко падает.

CUDA версии выше 5.0 позволяет использовать динамический параллелизм, то есть производить запуск нового ядра с произвольным размером сетки прямо из кода устройства. Это позволяет как решать малые локальные подзадачи, так и запускать полноразмерные ядра, экономя время синхронизации между процессором и видеокартой при завершении выполнения ядра между итерациями алгоритма.

Во время выполнения ядра на GPU, можно продолжить выполнение кода CPU. Также можно запустить выполнение другого ядра во втором потоке выполнения, но взаимодействие двух потоков выполнения через средства языка программирования невозможно.

В модели программирования CUDA доступны следующие типы памяти:

- Register – память регистров, в которых хранятся созданные в ядре переменные.
- Constant – быстрая константная память для неизменяющихся во время исполнения ядра переменных. Значение в памяти можно поменять непосредственно перед запуском ядра.
- Global – глобальная память. Здесь хранятся входные данные. Ядра могут читать и писать в произвольные места глобальной памяти, но произвольный доступ имеет малую скорость.
- Local – область глобальной памяти, использующаяся для хранения переменных или созданных в ядре массивов, не помещающихся в память регистров. Доступ к такой памяти ненамного быстрее, чем к глобальной за счет того, что память упорядочена по индексу нитей.

Программист не может сам использовать эту память, так как логически она является продолжением памяти регистров, а за ее использование отвечает компилятор и спецификация конкретного устройства.

- L1/TEX, L2 Cache – два уровня быстрой кэш памяти для сохранения значений из глобальной и текстурной памяти.
- Shared – общая для блока память. Доступ к ней происходит быстрее чем к глобальной памяти. На момент запуска ядра ее содержимое не определено. Ядро само заполняет общую память, каждая нить может заполнить относящийся к ней элемент. Для сохранения элемента в общую память необходимо сделать запрос к глобальной памяти. Общую память выгоднее всего использовать, если для каждой нити требуется запрос не только к своему соответствующему элементу, но и к соседним. В данном случае выгодно чтобы каждая нить из блока скопировала в общую память свой соответствующий элемент, а потом производило чтение из общей памяти.
- Texture – текстурная память, доступная только для чтения. Позволяет хранить одно-, дву-, и трехмерные плотные массивы из данных типа integer или float (и различного числа каналов), оптимизированные для хранения текстур. Так, кроме обычного доступа к элементу по его индексу, есть доступ к произвольным координатам, проводящий интерполяцию между значениями, также доступно MIP-текстурирование — сохранение вместе с текстурой дополнительных убывающих в размере текстур, каждая из которых в два раза меньше предыдущей по обеим осям. Обычно в этих дополнительных текстурах значение пикселя является средним из значений четырех пикселей на уровне ниже. Элементы текстуры расположены в памяти не в линейном порядке, а в некотором незадокументированном порядке на одной из плотно заполняющей

пространство кривой, например в порядке обхода по Кривой Пеано. Точный способ не указан в документации, потому что может изменяться в разных поколениях архитектур видеокарт. Подобный способ укладывания данных улучшает локальность хранения данных и обеспечивая повышенный процент попадания в кэш при чтении локально близких значений из текстуры.

- Surface – тип памяти, похожий на текстурную память, с отличием в том, что эта память доступна для записи, но для нее недоступна интерполяция и MIP-текстурирование.

Texture и Surface память расположена в глобальной памяти, но имеет преимущество перед ней за счет логического устройства хранения данных. Подробнее о модели программирования в источнике [13].

3. Описание используемых алгоритмов

В рамках работы были написаны программы, реализующие следующие методы: метод конечных разностей, метод трассировки лучей, метод отражений. Все алгоритмы написаны на языке C++ для платформы CUDA. Программы, реализующие методы трассировки и отражения, запускаются через Matlab с помощью плагина Parallel Computing Toolbox или обертки MexFunction – API для выполнения C++ функций в Matlab. Программа, реализующая метод конечных разностей запускается из C++ приложения. Исходный код методов трассировки и отражений расположен в репозитории GitHub[1].

Вычисления проводились на компьютере характеристиками: процессор AMD Ryzen 5 4600H 3.00 GHz, видеокарта Nvidia GeForce GTX 1650Ti.

[1] https://github.com/zarond/NIR

3.1. Метод трассировки лучей

Выпускаются множество лучей, которые зеркально отражаются от стен, каждый из которых дает конечный вклад в импульсную характеристику, если он попадает в источник менее чем за максимальное число отражений (в этом примере оно равняется 40). Реализация в файле kernel_Geom.cu через функцию

__global__ void
kernelRayTracing(const float *Edges_f, const unsigned int N, float *IR, const unsigned int SampleCount, const int maxReflections, const int T,const float v, const float d_t, const float x_ir, const float y_ir, const float x_s, const float y_s, const float s_r=0.1),

принимающую в себя массив ребер в формате (x_1, y_1, x_2, y_2), где нормаль ребра считается как $(-y_2+, y_1, x_2 - x_1)$, то есть слева от вектора ребра. Результат записывается в массив IR.

Пример использования:
[ImpulseResponse_c]=feval(k,Edges_c, size(Edges,2)/4, ImpulseResponse_c, SampleCount, maxReflections, T, v, timestep, x_ir, y_ir, x_s, y_s, s_r);
На Рис. 1 и Рис. 2 изображены примеры импульсных характеристик, полученных методом трассировки лучей.

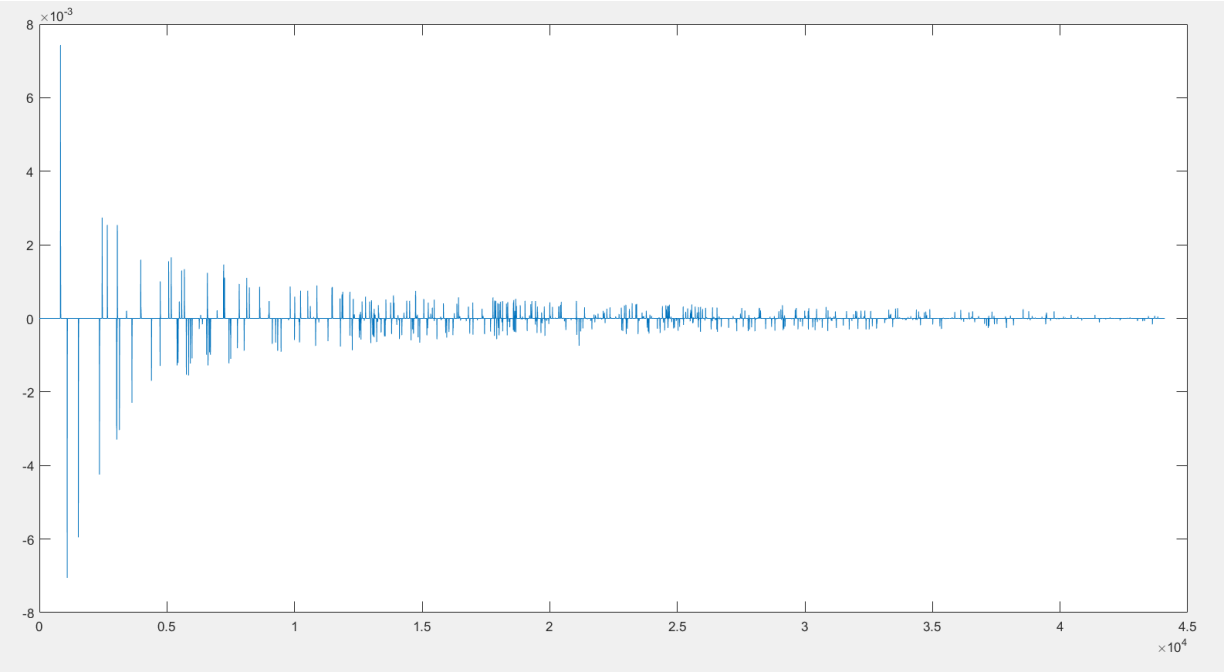

Рис. 1 – Результат трассировки лучей в комнате room.png

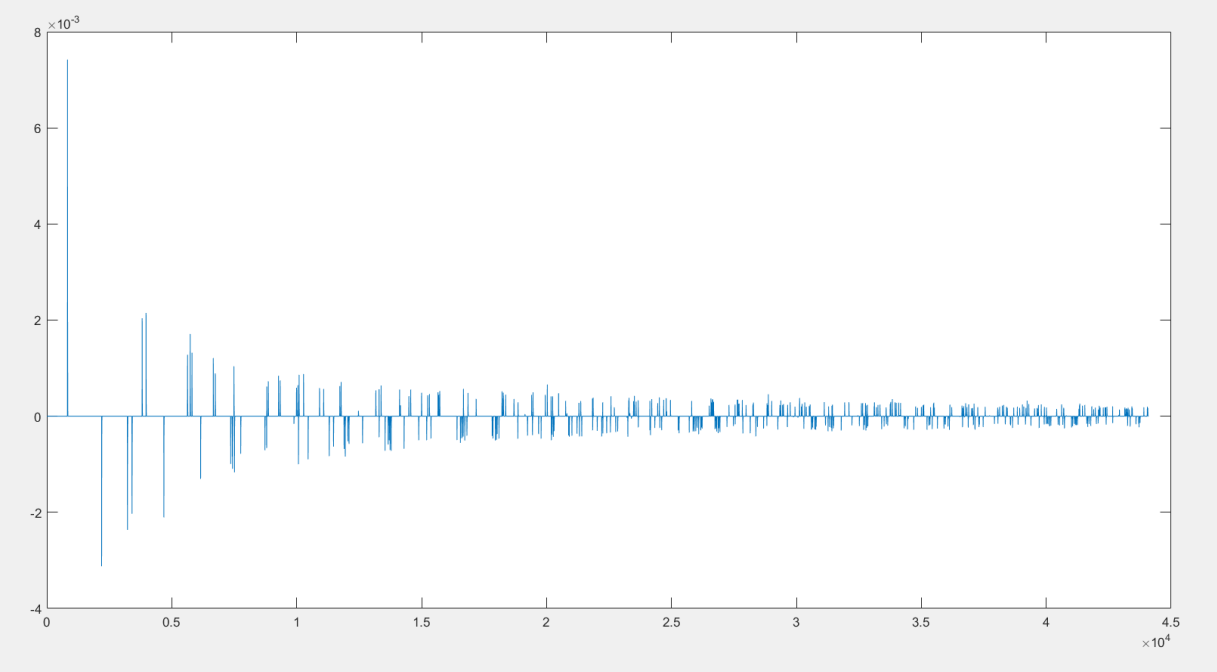

Рис. 2 – Результат трассировки лучей для квадратной комнаты

Количество лучей SampleCount = 24000000, максимальное число отражение MaxReflections = 40, время работы 2,6563 секунд.

3.2. Метод отражений

Для каждой пары источника и ребра рекурсивно строятся его производные мнимые источники. На рисунке 6 приведены примеры расчета источников и видимости для 64 источников для двух комнат.

Реализация в файлах kernel_IS.cu и kernel_visibility.cu. Функция вызывается через оболочку Mexfunction и имеет вид:

```
__global__
void kernelIS(const float4 * Edges, ISource* d_Sources, const unsigned int N,
const unsigned int MaxSources, const float x_s, const float y_s),
```

где информация об источниках возвращается в формате:

```
struct ISource{
	float2 pos;
	float4 window;
	int edgeid;
	int parent_source;
	int reflections;
};
```

Пример использования:

```
sources = kernel_IS(Edges,size(Edges,2)/4, MaxSources, x_s, y_s);
sources_c = gpuArray(cast(sources(1:9,:),'single'));
[Visibility_c,ImpulseResponse_c] = feval(k, Edges_c, size(Edges,2)/4, sources_c,
Visibility_c, ImpulseResponse_c, MaxSources, T, v, timestep, x_ir, y_ir);
```

На Рис. 3 и Рис. 4 визуализирован результат работы метода мнимых источников. Символом «*» обозначены мнимые источники, символом «o» обозначен слушатель. Зеленым цветом обозначаются видимые из положения слушателя мнимые источники, а красным – невидимые с позиции слушателя.

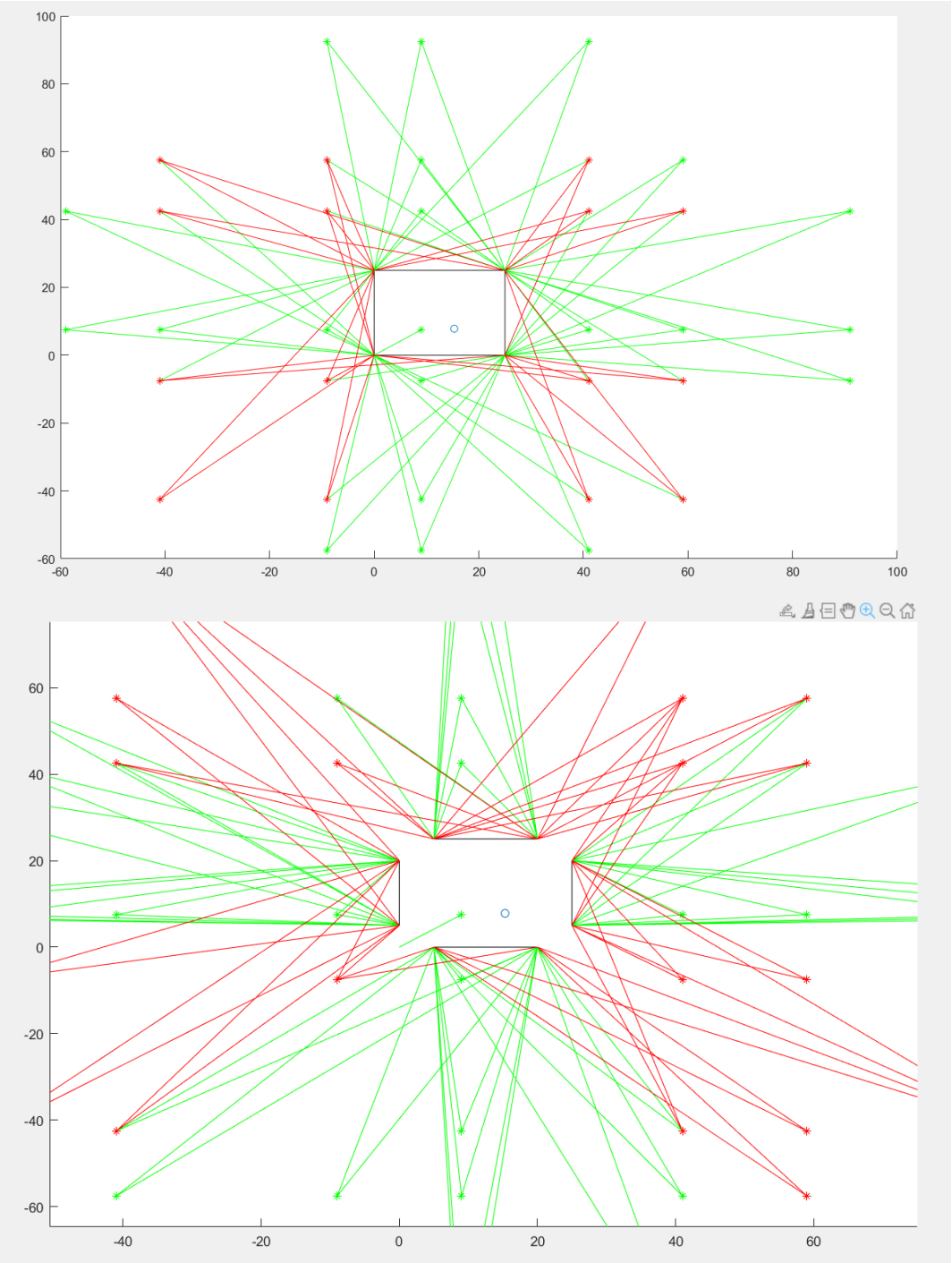

Рис. 3 и 4 – Визуализация работы метода мнимых источников, 64 источника

Для 50000 источников в квадратной комнате метод работает 15,6 миллисекунд, для комнаты room.png работает за 62,5 миллисекунды.

4. Метод конечных разностей и сравнение производительности

Цель раздела – провести сравнение производительности методов решения задачи моделирования акустики помещений, выполняемых на центральном процессоре и на видеокарте. В рамках работы были написаны два алгоритма, каждый из которых представлен в нескольких вариантах. Было написано приложение, которое показывает работу алгоритмов в режиме реального времени с помощью графической библиотеки GLUT, и позволяет проводить бенчмарки для различных конфигураций симуляции и сохранять результаты симуляции в файл. Были проведены тесты программы на личном компьютере и на вычислительной платформе СПБГУ. Исходный код и результаты выполнения программы размещены в репозитории GitHub[2].

4.1. Описание программы

Программа написана полностью на языке C++ для платформы с поддержкой CUDA. Использовалась среда разработки Microsoft Visual Studio 2019, с помощью которой я создал исполняемый файл для операционной системы Windows 10. Для запуска программы на вычислительной платформе СПБГУ был создан файл Makefile, который помогает компилировать программу с помощью программы make вне зависимости от платформы. Таким образом была создана кроссплатформенная программа, которая также запустилась на вычислительном кластере от СПБГУ, работающем под управлением операционной системы на основе ядра Linux.

Программа реализует несколько разновидностей методов конечных разностей на двумерной пространственной сетке. Сравниваемые варианты методов для выполнения на CPU и GPU имеют минимальные различия в логике вычислений и дают равные результаты. В работе рассматриваются воздействие на производительность платформо-зависимых приемов

[2] https://github.com/zarond/NIP

оптимизации кода. Для оптимизации кода на CPU используется библиотека OpenMP, облегчающая работу с распараллеливанием алгоритма на несколько потоков, а для оптимизации кода на GPU используется работа с общей памятью блока (Shared memory) и динамический параллелизм. Список методов:

1. WaveIteration – однопоточный метод на CPU, выполняющий одну итерацию симуляции.
2. WaveIterationOMP – многопоточный метод на CPU, выполняющий одну итерацию симуляции.
3. WaveIterationOMPMultipleFrames – многопоточный метод на CPU, выполняющий несколько итераций симуляции за раз.
4. WaveIterationKernel – метод на GPU, выполняющий одну итерацию симуляции.
5. MultipleIterationsKernel – метод на GPU, выполняющий несколько итераций симуляции за раз с помощью динамического параллелизма. В этом случае запускается малое основное ядро на сетке с только одним потоком, которое уже запускает основную большую сетку для расчета итерации. В этом случае синхронизация выполнения работы не требует обращения к CPU, что потенциально увеличивает производительность.
6. WaveIterationKernelSM – метод на GPU, выполняющий одну итерацию симуляции, использующий общую память. В этом случае необходимые данные сначала копируется в память, доступную одному блоку. Производительность повышается за счет того, что обращение к элементу в общей памяти блока происходит быстрее, чем обращение в глобальную область памяти, и за счет того, что один элемент используется в нескольких вычислительных нитях ядра.

7. WaveIterationAltOMP – многопоточный метод на CPU, выполняющий одну итерацию симуляции для альтернативного алгоритма №2.
8. WaveIterationKernelAlt – метод на GPU, выполняющий одну итерацию симуляции для альтернативного алгоритма №2.

Все методы, кроме последних двух, выполняют одни и те же вычисления. Два последних метода выполняют альтернативный алгоритм (далее – Алгоритм №2). Методы для CPU описаны в файле functionCPU.cpp, а методы для GPU – в файле functionGPU.cu.

Входные данные для программы – число, имя файла конфигурации. Если не указаны данные, то в качестве числа берется 0, а в качестве имени файла конфигурации – «config.model». Если файл конфигурации не найден, то используются параметры по умолчанию. Входное число указывает режим работы программы. Если число от 0 до 7, то включается графический режим, и выбирается один из восьми описанных выше методов.

Число -1 включает режим бенчмарка, сравнивающий производительность, то есть время одной итерации симуляции, для всех методов на разных размерах поля $N \times N$ для N от 500 до 5000 с шагом в 500.

Число -2 включает режим бенчмарка, сравнивающий производительность метода WaveIterationOMP на различном числе потоков.

Число -3 включает режим бенчмарка, сравнивающий производительность метода WaveIterationOMPAlt на различном числе потоков.

Число -4 включает режим бенчмарка, сравнивающий производительность метода WaveIterationKernel на сетке с блоками размером в $N \times N$ нитей, для N от 1 до 32.

Число -5 включает режим бенчмарка, сравнивающий производительность метода WaveIterationKernelAlt на сетке с блоками размером в $N \times N$ нитей, для N от 1 до 32.

Для замера времени одной итерации берется среднее арифметическое от 100 итераций. Кроме того, для этих 100 итераций высчитывается среднеквадратичное отклонение, максимальное и минимальное значения.

В файле конфигурации указаны параметры симуляции, такие как частота симуляции hz и скорость распространения волны v, из которых высчитывается пространственный и временной шаг сетки, размеры симулируемой области, положение источника и приемника сигнала, T – общее количество итераций, room – имя файла изображения с описанием геометрии пространства, F – имя файла с сигналом для источника, Threads – количество OpenMP потоков, b – коэффициент поглощения среды для первого алгоритма и коэффициент отражательной способности стен для второго алгоритма. Булевы параметры JustRunSim, RunSimMode, WriteSimResults определяют дополнительный режим работы программы. Если JustRunSim=1, то программа работает в режиме без графики методом под номером RunSimMode, и, в случае если WriteSimResults=1, она записывает результаты всех итераций в файл. В ином случае WriteSimResults=0, в файл записываются в только данные из точки приемника. К сожалению, этому режиму было уделено наименьшее время разработки, поэтому он позволяет записывать все результаты только для методов WaveIteration, WaveIterationOMP, WaveIterationKernel, вдобавок результаты записываются в текстовом виде, что порождает файлы больших размеров и негативно сказывается на времени работы. Однако, этот режим полезен тем, что он показывает время вычисления для конкретной задачи.

Для работы с данными использовались массивы из библиотеки Blitz++ [14]. Сами массивы хранят данные в непрерывной области памяти, в которой матрицы хранятся построчно, что позволило в случае необходимости обращаться к данным напрямую с помощью указателя.

Для загрузки данных геометрии комнаты из изображения использовал однофайловую библиотеку stb_image.h[3]. В коде задал собственное наименование для вещественно числа: typedef float Real. Я протестировал производительность для типов double и float. Чтобы поменять тип данных, используемых в программе, необходимо в нескольких файлах заменить typedef float Real на typedef double Real. Я проводил сравнения для разных типов данных, потому что заявленная пиковая производительность для float и double на видеокарте отличается в несколько раз в пользу float. Также я обнаружил, что в CUDA константы, заданные в коде, не конвертируют свой тип на этапе компиляции, а делают это во время выполнения, поэтому даже константы вида 0.0 и 0.5 нагружали конвейер операций над типом double, что сильно сказывалось на производительности ядра. В методах для CPU такой проблемы не возникало. Проблема решалась, когда я прописал всем константам явное преобразование в тип Real. Читаемость кода ухудшилась, но программа больше не использовала операции над double при работе с данными в формате float. При работе над кодом CUDA использовал профайлер NsightCompute, который помог мне распознать несколько проблем, в том числе описанную выше.

4.2. Описание используемых алгоритмов

Алгоритм №1 и альтернативный алгоритм №2, оба относятся к классу методов конечных разностей (finite-difference time-domain (FDTD)).

Алгоритм №1

Волновое уравнение задается в виде:

$$\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} - v^2 \Delta u = f(x,t), \qquad \Delta u = \sum_{i=1}^{n} \frac{\partial^2 u}{\partial {x_i}^2} \tag{16}$$

[3] https://github.com/nothings/stb

В этом методе используется нулевое граничное условие, то есть $u = 0$ внутри стен.

В этом алгоритме используется трёхслойная разностная схема с приближением второго порядка:

$$u_{ij}^{t+1} = \frac{2u_{ij}^{t} - u_{ij}^{t-1}\left(1 - \frac{\Delta t}{2}b\right) + v^2\frac{\Delta t^2}{\Delta x^2}\left(u_{i-1j}^{t} - 2u_{ij}^{t} + u_{i+1j}^{t} + u_{ij-1}^{t} - 2u_{ij}^{t} + u_{ij+1}^{t}\right)}{\left(1 + \frac{\Delta t}{2}b\right)}. \quad (17)$$

Данный метод позволяет учитывать затухание волны в среде, но отражения от стен сохраняют 100% энергии. Также, при отражении меняется фаза волны, что соответствует поведению для расчета волнового уравнения, но может не подходить для задач акустики, где, как правило, фаза волны не меняется при отражении. Данный метод успешно приближенно решает волновое уравнение для нулевого граничного условия.

Алгоритм №2

В предыдущем алгоритме используются только данные $u(x, y, t)$, в этом алгоритме кроме данных о значении отклонения давления в точке используются данные о частных производных $v_x = u'_x$ и $v_y = u'_y$. Сам метод и схема его вычисления была взята из источника [10]. Данный метод дает корректные (с акустической точки зрения) отражения и позволяет задавать коэффициент поглощения энергии для материала. Каждая точка может обладать своим коэффициентом, но в рамках данной работы был рассмотрен только вариант с задаваемым одним коэффициентом на все поверхности. Значения в итерациях задается через уравнения:

$$p^+ = B(p - C\nabla \cdot \vec{v}), \quad (18)$$

$$v_x^+ = BB_<(v_x - C\nabla_x p^+) + (B_< - B)(BY_< + B_<Y)(Bp^+ + B_<p_<^+), \quad (19)$$

$$v_y^+ = BB_{\vee}\left(v_y - C\nabla_y p^+\right) + (B_{\vee} - B)(BY_{\vee} + B_{\vee}Y)(Bp^+ + B_{\vee}p_{\vee}^+). \quad (20)$$

p, v_x, v_y – значения на текущем шаге в текущей точке, $*^+$ – знак плюс обозначает значения переменных на следующем шаге, $*_<$ и $*_\vee$ обозначают значения в положениях $\{(x - \Delta x, y), (x, y - \Delta y)\}$ соответственно. C – константа, равная $C = c\frac{\Delta t}{\Delta x}$, где c – скорость распространения волны, Δt и Δx – шаги сетки. При моей стратегии выбора шага сетки она равняется $\mathrm{C} = 0.5$. Для устойчивости алгоритма необходимо $C < \frac{1}{\sqrt{2}}$. B – индикатор твердой поверхности, равен нулю если в точке твердая граница, единице для свободного пространства. $Y = (1 - R)/(1 + R)$, где R – коэффициент отражательной способности поверхности. Для среды считается равным нулю.

Градиент вычисляется следующим образом:

$$\nabla_x p^+ = p^+ - p^+_< \tag{21}$$

$$\nabla_y p^+ = p^+ - p^+_\vee \tag{22}$$

$$\nabla \cdot \bar{v} = (v_x(x + \Delta x, y) - v_x(x, y)) + (v_y(x, y + \Delta y) - v_y(x, y)) \tag{23}$$

Получается зависимость данных в точке (x, y) от значений переменных в точках $\{(x - \Delta x, y), (x, y - \Delta y)\}$. При стандартном проходе по циклу по $y = \overline{0, N}$ и $x = \overline{0, M}$ проблем не возникает, но при распараллеливании зависимость данных мешает. В алгоритме для CPU данные разбиваются на блоки размера 64×64, и блоки обрабатываются параллельно с учетом зависимости данных. Идет цикл по блокам по диагоналям, так как диагональ может обрабатываться одновременно, так как блоки на диагонали не зависят от друг друга.

Для алгоритма на GPU вышеописанная схема распараллеливания не подходит. Я заметил, что для расчета v_x^+ и v_y^+ необходимы значения $p^+_<$ и $p^+_\vee$, но значения $v^+_{x<}$, $v^+_{y\vee}$ не нужны, поэтому провести вычисления можно в два прохода по данным. При первом проходе считаем p^+, а на втором v_x^+ и v_y^+, избавляясь таким образом от зависимости по данным от порядка вычисления. Получается, что оба этих проходов можно делать полностью параллельно на GPU.

4.3. Результаты

На Рис. 5 – 7 и Рис. 9 - 11 представлены скриншоты работы программы. Красные области – положительное отклонение значений давления, синие – отрицательное. Зеленым обозначены твердые стены. На Рис. 8 представлен пример вывода программы в консоль, в котором указано среднее время итерации алгоритма на основе измерения времени 100 итераций, среднеквадратичное отклонение, минимальное и максимальное значения времени итерации.

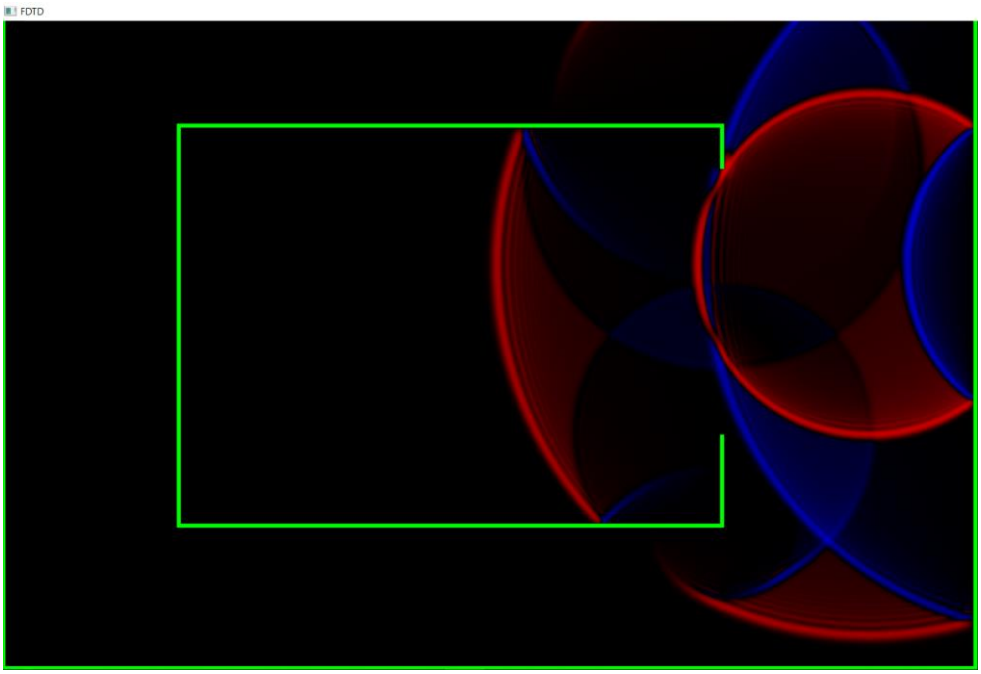

Рис. 5 Визуализация работы алгоритма №1. Клавиши + и – управляют яркостью, а клавиши w,a,s,d меняют положение источника.

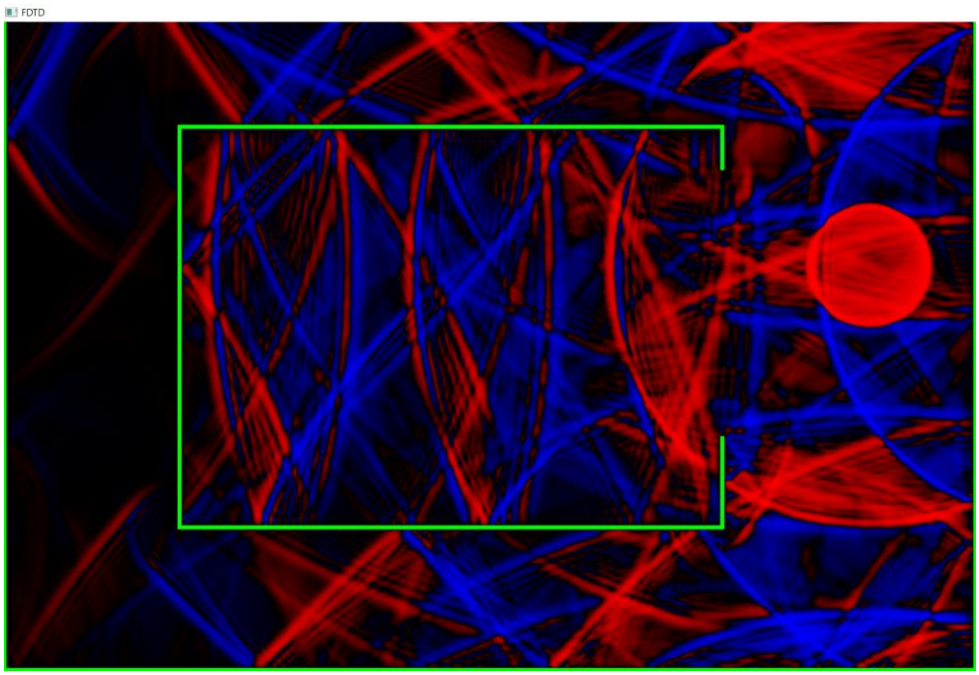

Рис. 6 Визуализация работы алгоритма №1 в комнате, описываемой файлом room.png

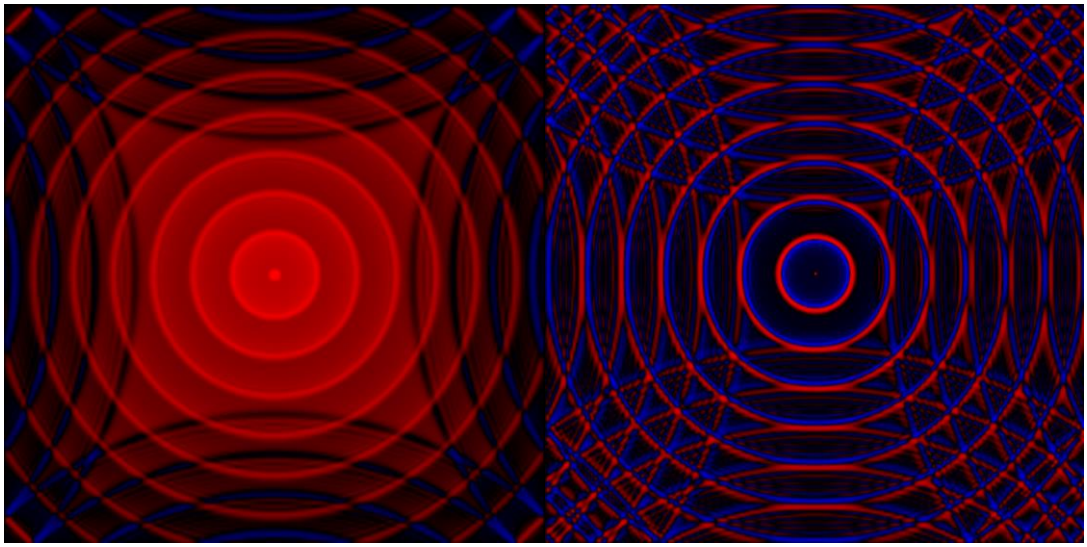

Рис. 7 Слева - алгоритм №1, справа - алгоритм №2

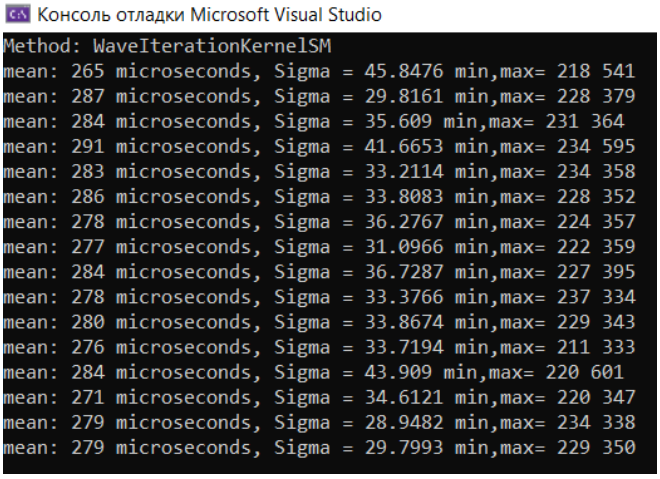

Рис. 8 Вывод программы в консоль в режиме визуализации

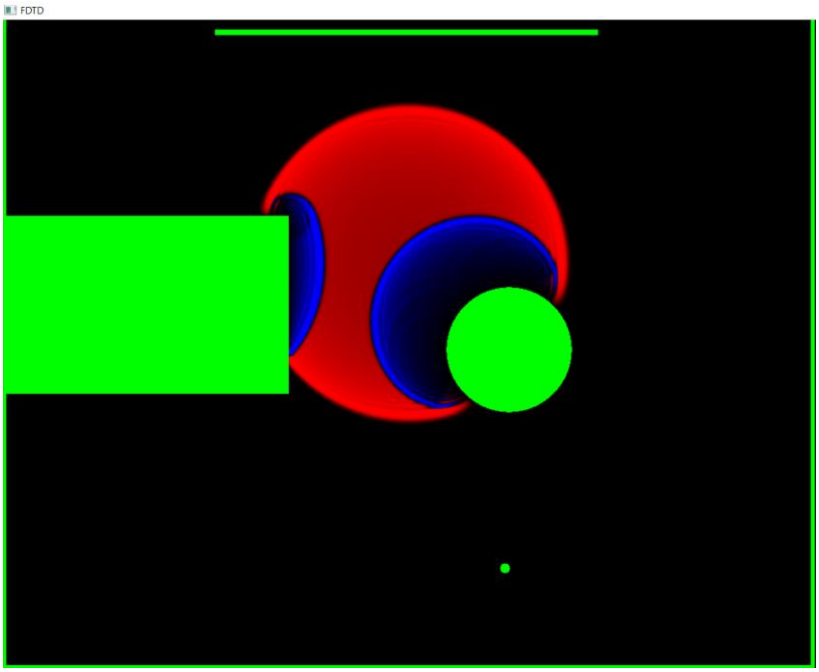

Рис. 9 Визуализация работы алгоритма №1

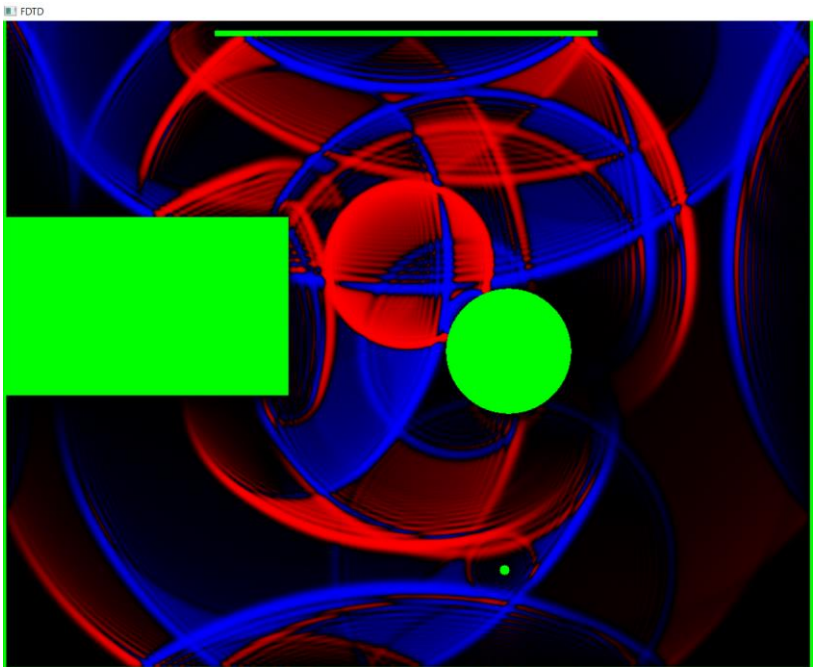

Рис. 10 Визуализация работы алгоритма №1

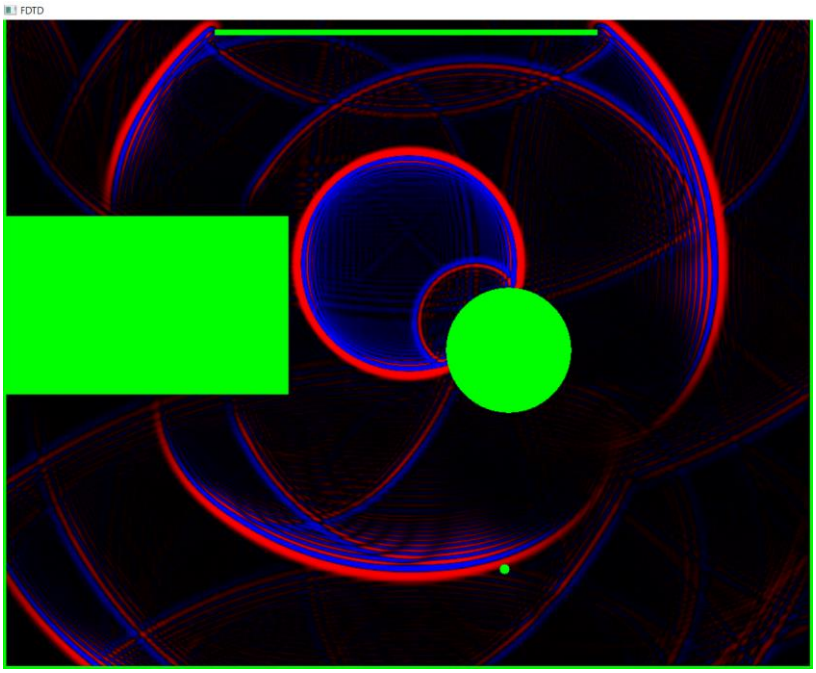

Рис. 11 Визуализация работы алгоритма №2. Малый коэффициент отражения поверхности.

Параметры симуляции взяты таким образом:

максимальная частота hz = 44100 герц, временной шаг timestep = 1.0/hz = 2,26 * 10^-5 сек, v = 331 метр/секунду, шаг сетки в пространстве d_x = 2 * v * timestep = 0,015 м = 15,5 мм, размеры комнаты (X_size ; Y_size) = (18 ; 15) метров, размер сетки симуляции (N , M)=(1000 ; 1200).

На Рис. 12 – 15 представлены полученные в симуляциях импульсные характеристики. На графиках заметны несколько явных пиков от ранних отражений. На Рис.15 видно, что шумовая компонента довольно быстро начинает перевешивать сигнал.

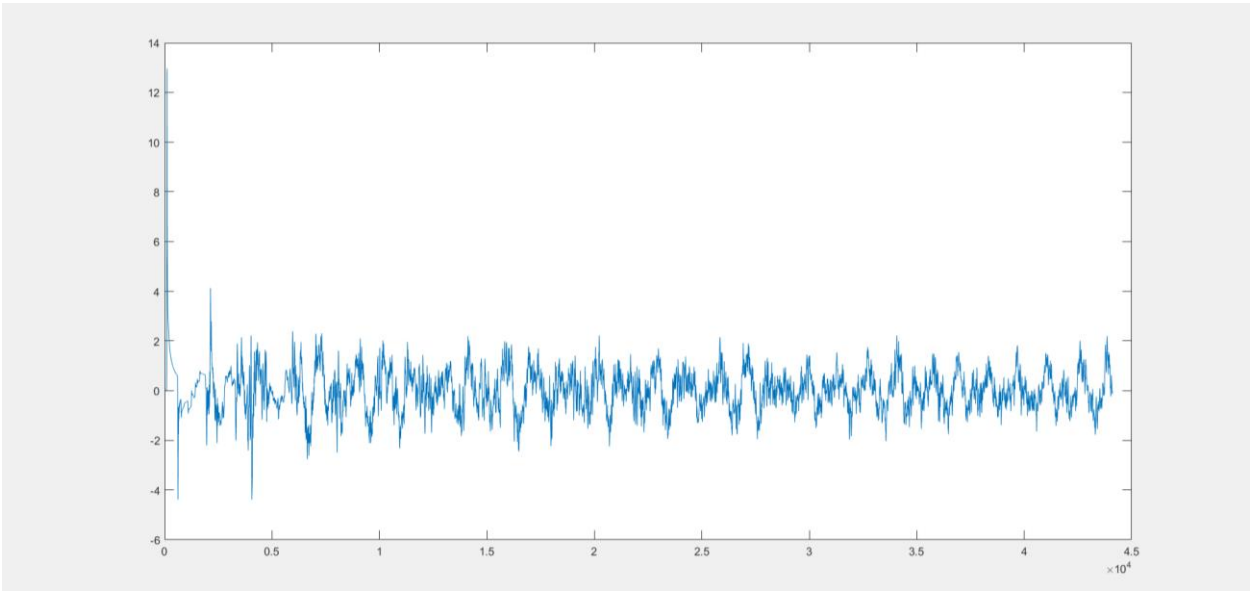

Рис. 12 – Импульсная характеристика, полученная для комнаты, описанной в файле room.png

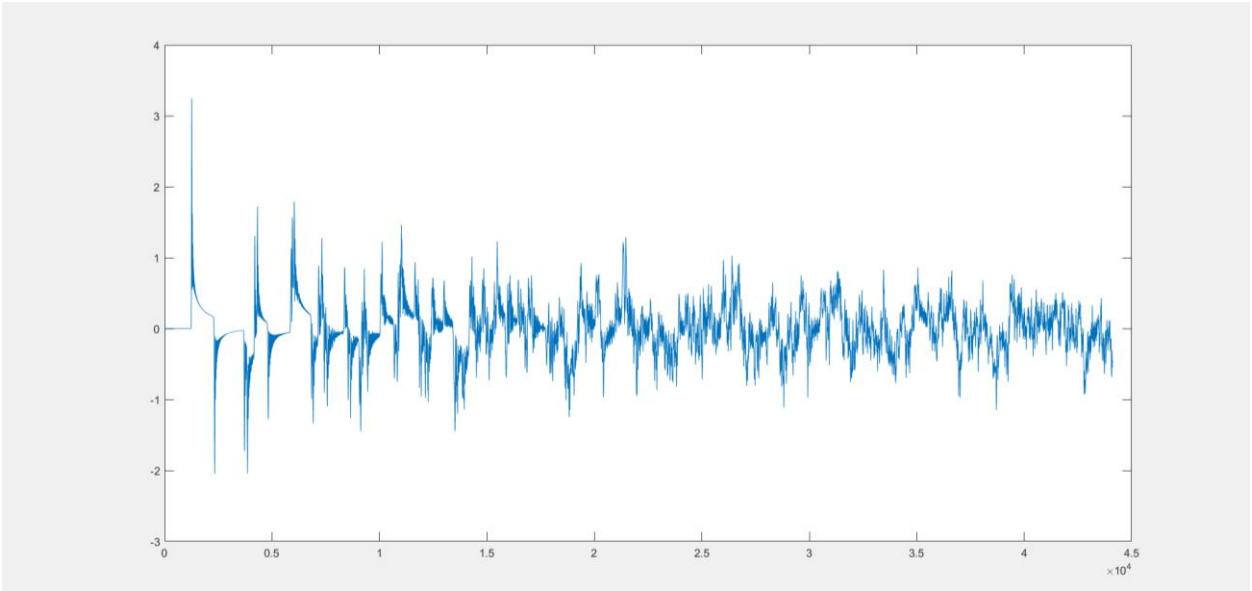

Рис. 13 – Импульсная характеристика квадрата размерами 25 на 25 метров

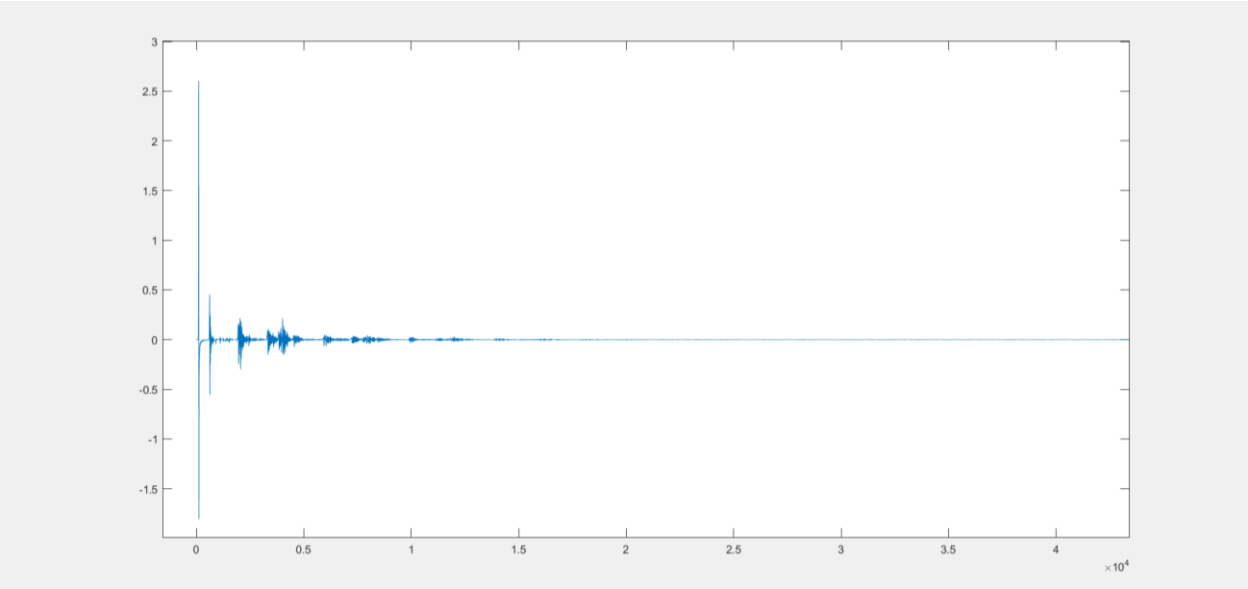

Рис. 14 – Импульсная характеристика, полученная для комнаты, описанной в файле room.png для алгоритма №2 с параметром b = 0.9

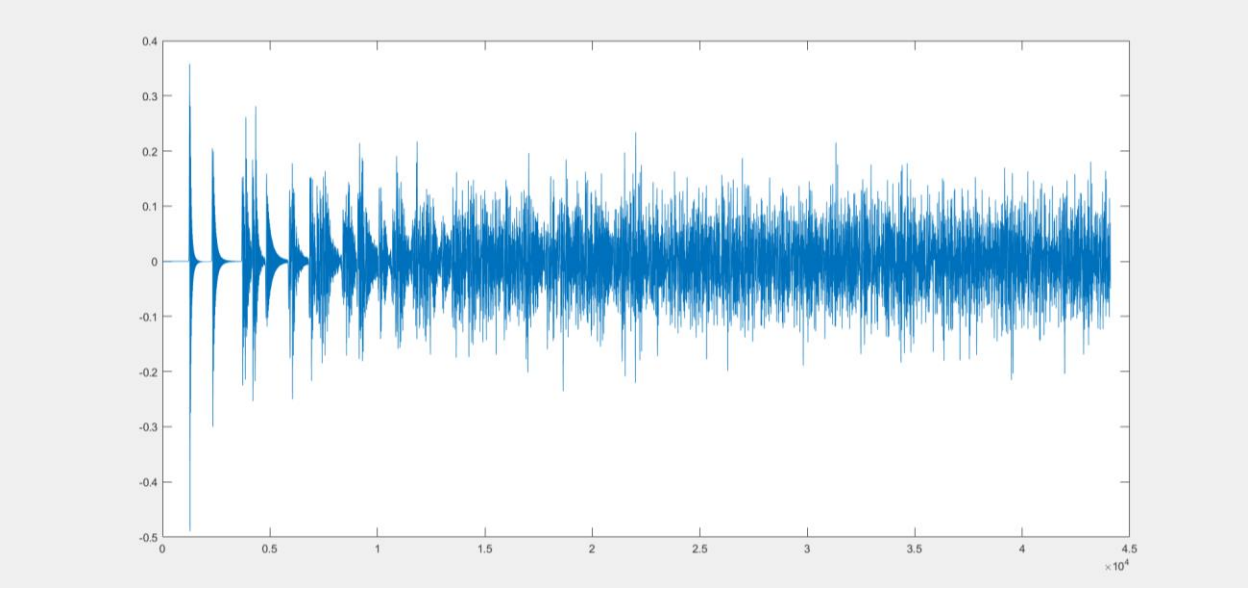

Рис. 15 – Импульсная характеристика квадрата для алгоритма №2 с параметром b = 0.9

Расчёт 44100 итераций алгоритмом №1 на сетке размером 1000 x 1200 из файла room.png прошел за процессорное время в 9993 миллисекунды, то есть одна итерация в среднем заняла 227 микросекунд.

Этот же расчёт алгоритмом №2 прошел за 35528 миллисекунд, одна итерация в среднем заняла 806 микросекунд.

Расчёт 44100 итераций алгоритмом №1 в квадрате размерами 25 на 25 метров занял 24216 миллисекунд, то есть одна итерация в среднем заняла 549 микросекунд.

Расчёт с помощью алгоритма №2 прошел за 79323 миллисекунды, то есть одна итерация в среднем заняла 1799 микросекунд.

При пониженной частоте симуляции в 4410 Гц и 4410 итерациях, симуляция алгоритмом №1 для квадрата размерами 25 на 25 метров занимает 222 миллисекунды, около 50 микросекунд на итерацию.

Далее проведем сравнение производительности на основе данных, полученных от бенчмарков. Выводы программы для бенчмарков находятся в папках floatres, doubleres, floatrescluster, doublerescluster в файлах benchmark.txt, benchmarkOMP.txt, benchmarkKernel.txt, benchmarkOMPAlt.txt, benchmatkKernelAlt.txt. На основе данных из этих файлов был создан Excel файл “Benchmark Results.xlsx”, в котором сохранены данные, а также построены графики сравнения производительности.

Рассмотрим графики результатов выполнения бенчмарков на моем персональном компьютере. Характеристики компьютера: процессор AMD Ryzen 5 4600H 3.00 GHz, видеокарта Nvidia GeForce GTX 1650Ti. Число OMP потоков было выбрано равным 12, а размеры блока для CUDA выбраны 32 на 32. MF в названии метода значит Multiple Frames – метод считает несколько итераций подряд, SM означает Shared Memory – метод использует общую для блока память. В некоторых из графиков линиями разных цветов обозначаются данные, полученные на методе, примененном на различных размерах квадратных сеток, с размером оси от 500 до 5000 с шагом в 500.

На Рис. 16 указано время итерации для всех сравниваемых методов. На оси X указан размер сетки стороны квадрата, на котором проходит симуляция. По оси Y отложено время выполнения одной итерации в микросекундах в логарифмическом масштабе.

На Рис.17, 18 построена зависимость ускорения методов по сравнению с однопоточным CPU методом для соответствующих размеров сетки.

Результаты для программы, основанной на типе данных float:

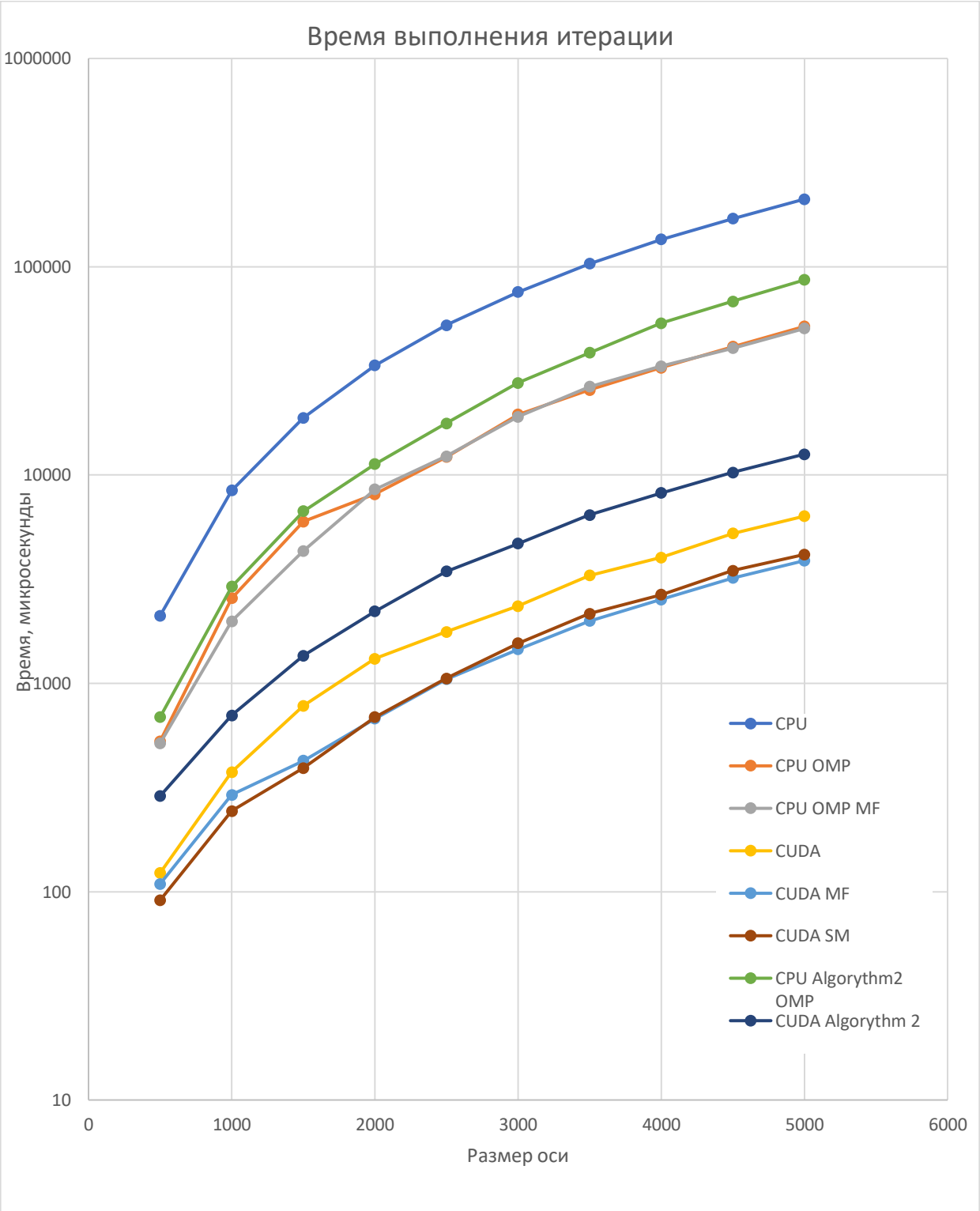

Рис. 16 Время выполнения одной итерации алгоритма в микросекундах в зависимости от размера сетки, логарифмический масштаб по вертикальной оси.

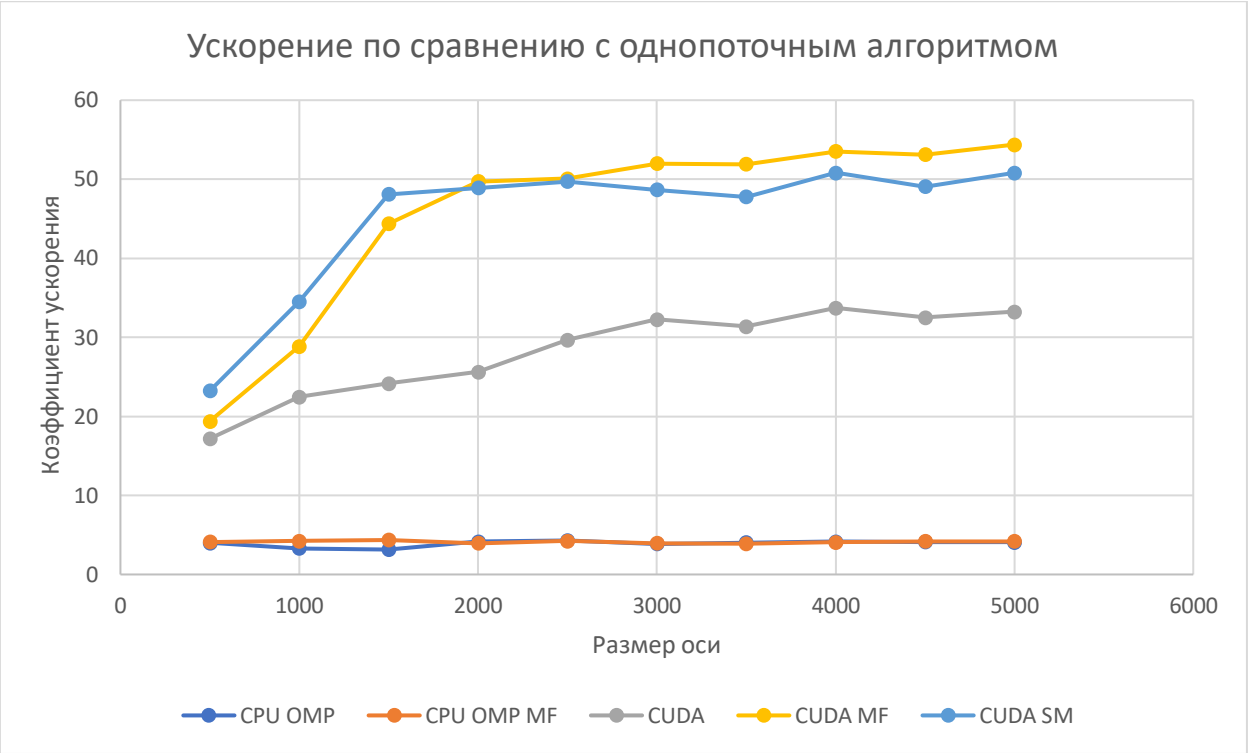

Рис. 17 Сравнение с однопоточным алгоритмом

По Рис. 17 можно наблюдать, что производительность лучшего метода на CPU в 12 раз меньше, чем производительность лучшего метода на GPU для размера оси сетки от 2000. Из графика также видно, что проведение нескольких итераций за проход в алгоритмах на CPU не дает прироста производительности, в то же время подобная стратегия в методах на GPU дает прирост производительности.

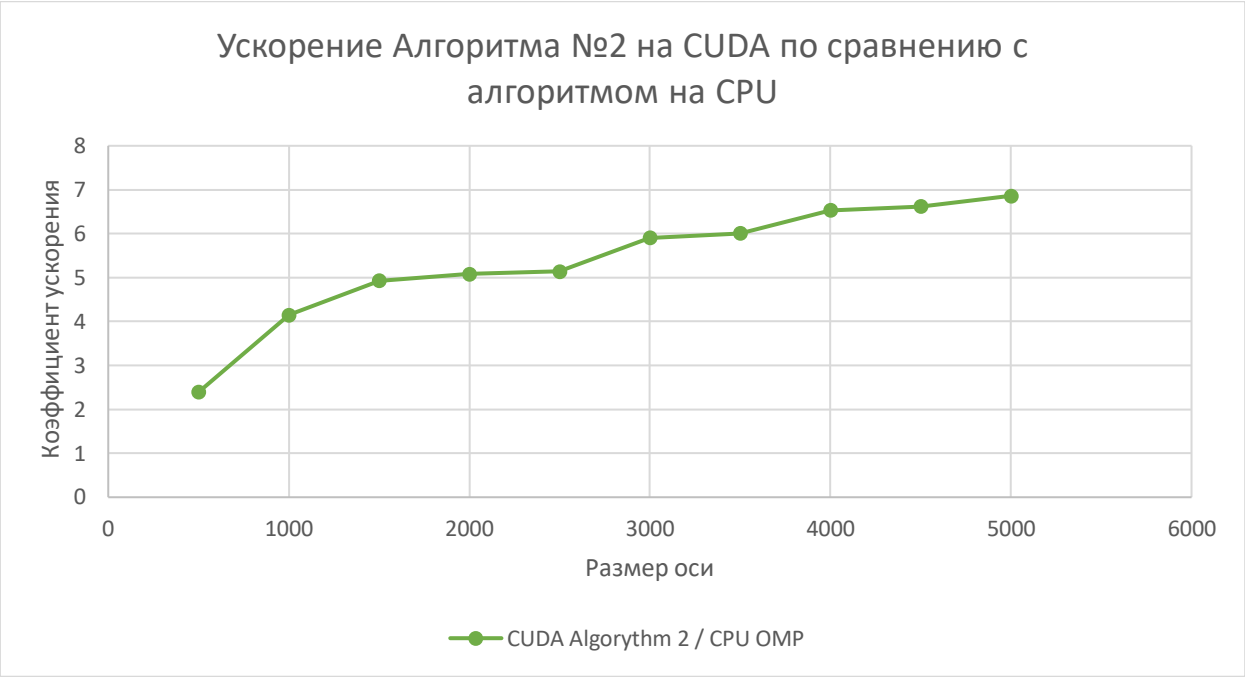

Рис. 18 Сравнение GPU алгоритма с CPU версией для алгоритма №2

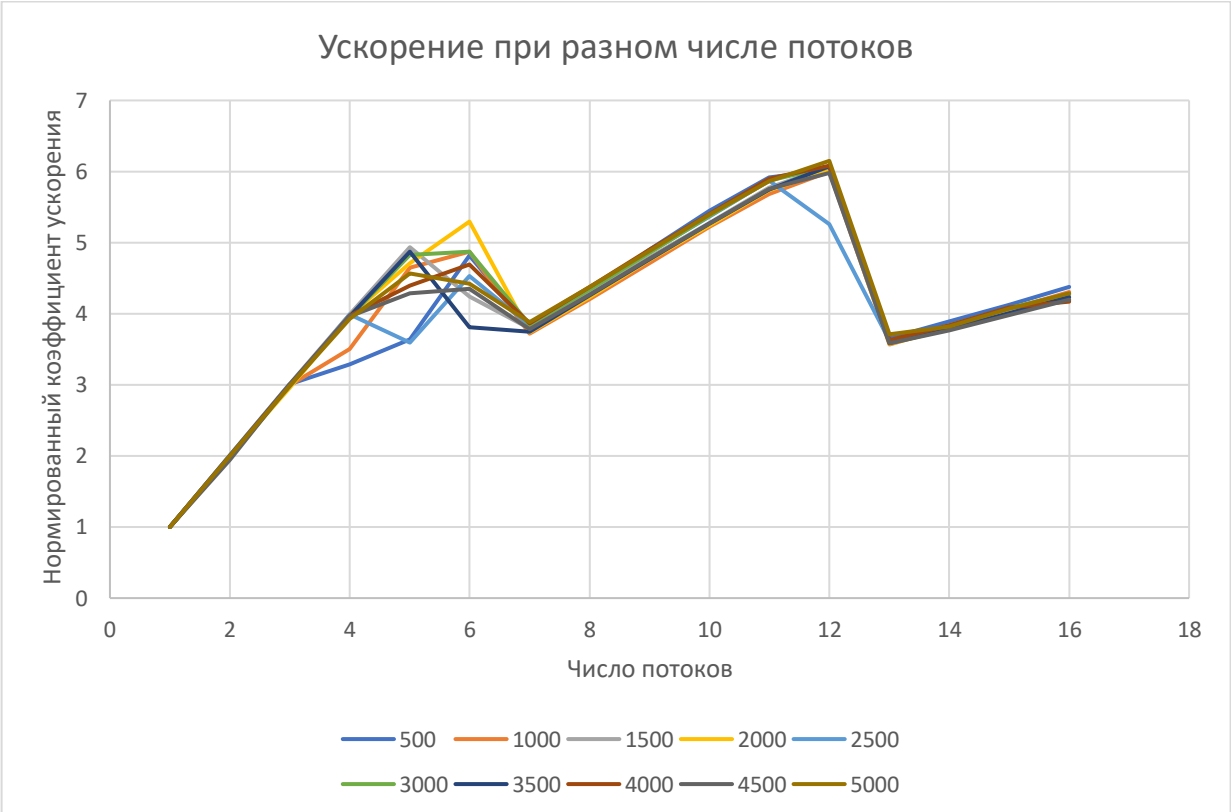

Рис. 19 Зависимость ускорения CPU алгоритма от числа потоков.

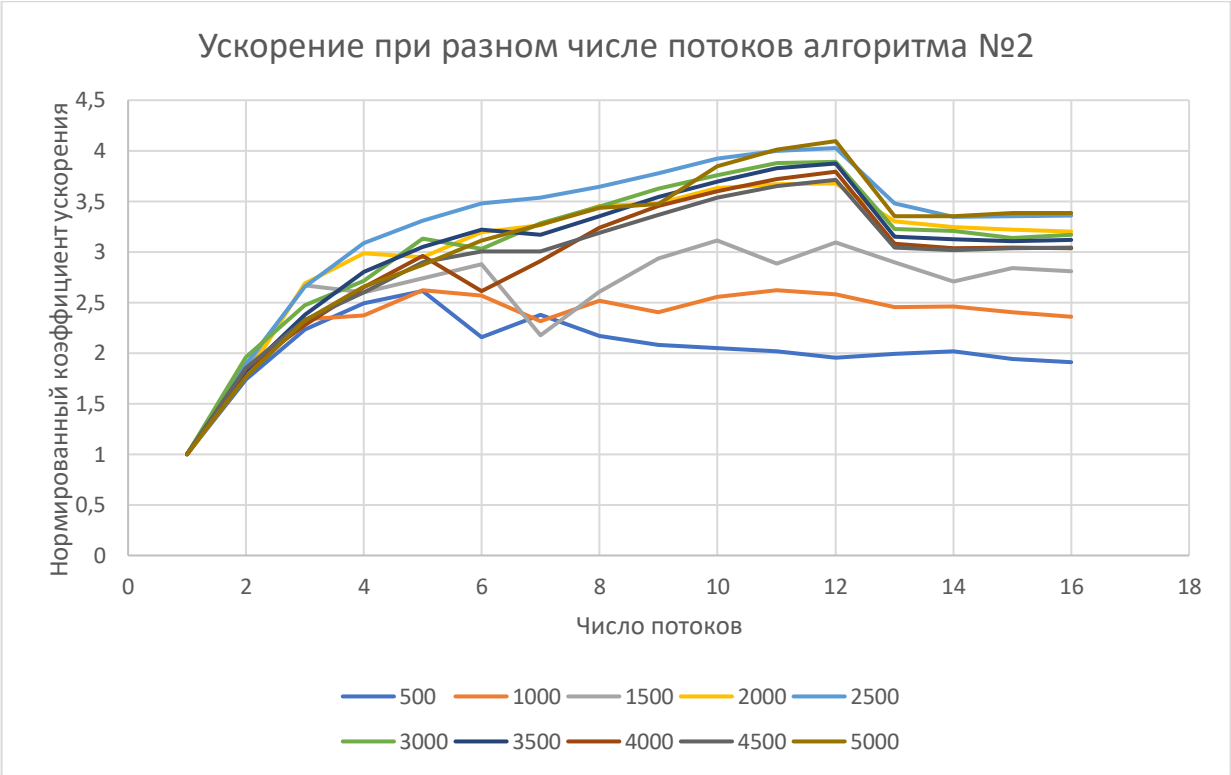

Рис. 20 Ускорение для алгоритма №2.

На Рис. 19, 20 изображены графики зависимости ускорения алгоритмов для CPU на различном числе потоков. Линия на графике – производительность метода на заданном размере сетки. Производительность нормирована относительно времени выполнения на одном потоке. На моем центральном процессоре 6 ядер и 12 логических процессоров, поэтому на Рис. 19, 20 наблюдается максимум ускорения, равный 6, в случае 12 потоков. Для алгоритма №2 зависимость сохраняется, но, вероятно, способ распараллеливания алгоритма не является оптимальным, из-за чего максимальное ускорение равно 4, а не 6.

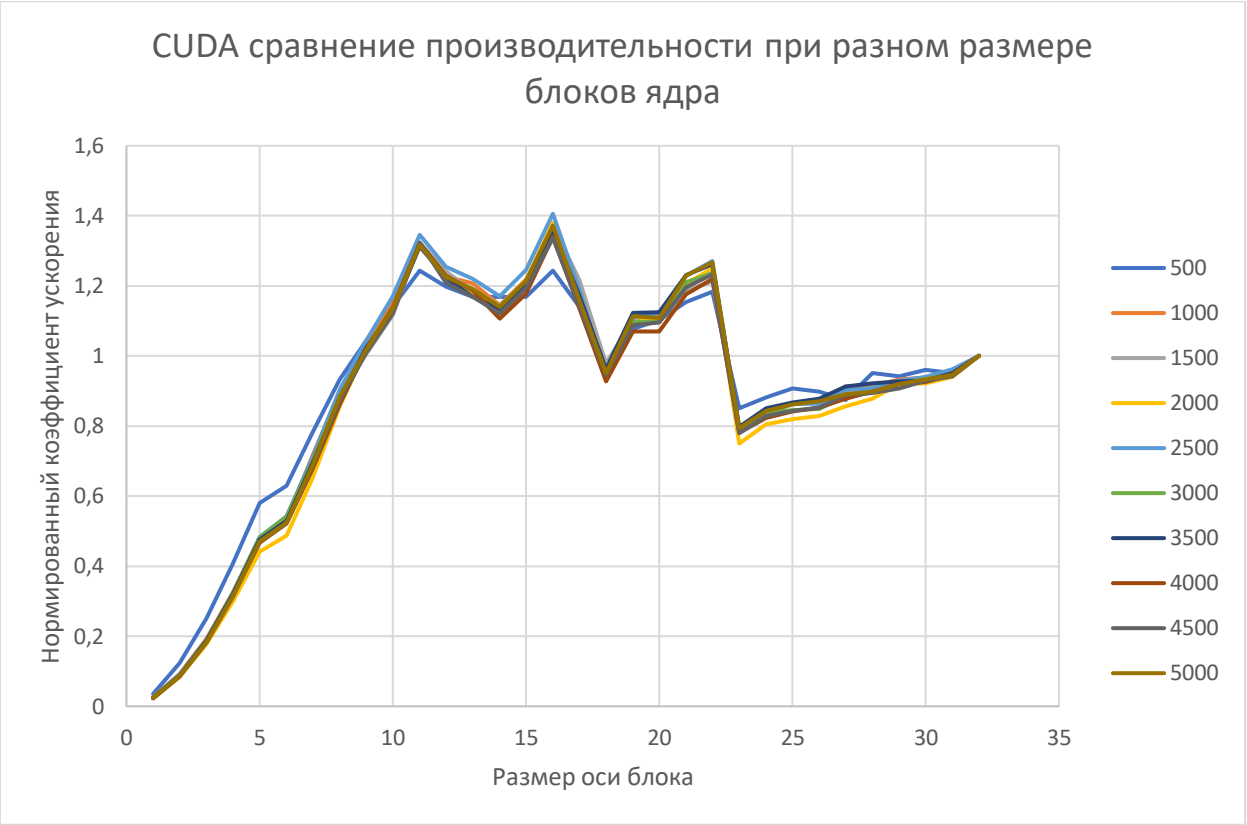

Рис. 21 CUDA сравнение производительности Алгоритма № 1 при разном размере блоков ядра.

На Рис. 21, 22 изображено сравнение производительности CUDA методов при разном размере стороны двумерного блока. Графики на различных размерах сетки нормированы относительно времени с размером блока, равного 32. Видно, что в случае выбора блока с размером в 16 для алгоритма №1, производительность повышается в 1.36 раз. Для алгоритма №2 размер блока имеет меньшее влияние на производительность.

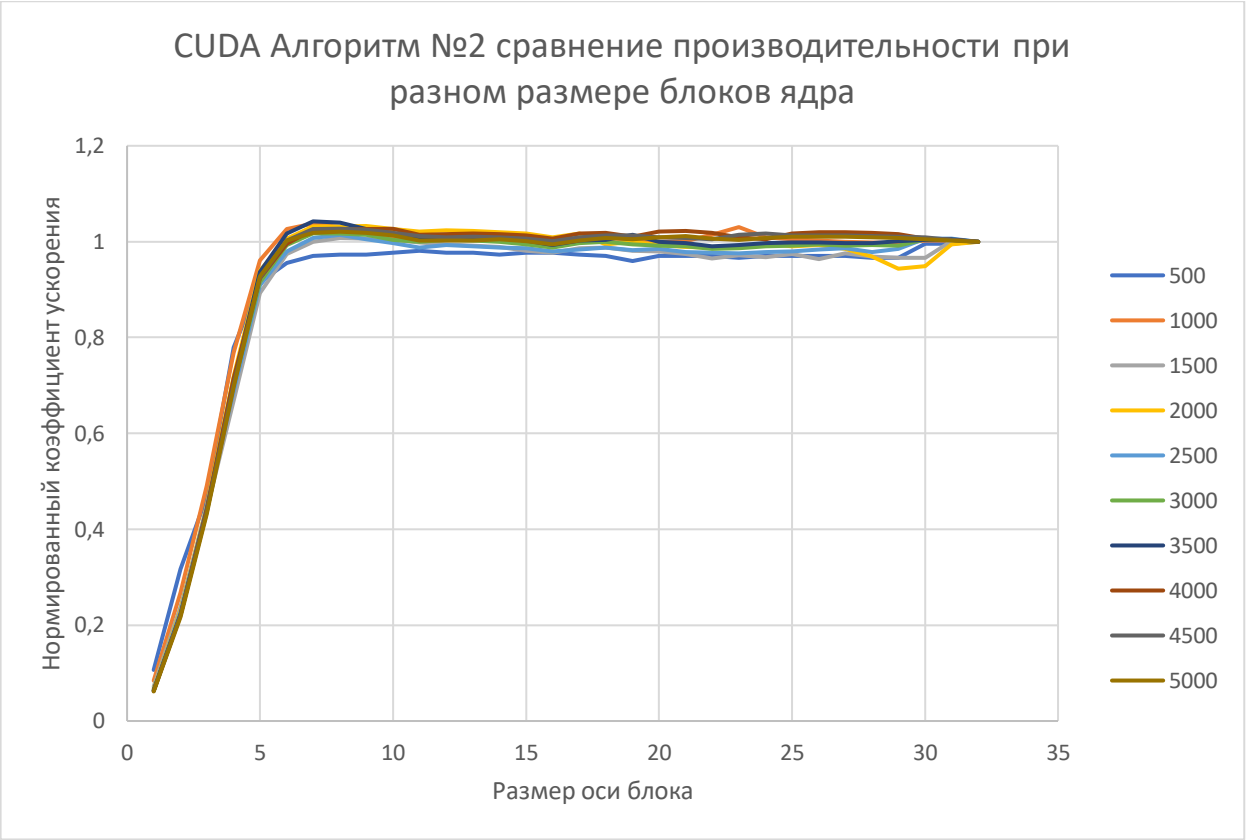

Рис. 22 CUDA сравнение производительности Алгоритма № 2 при разном размере блоков ядра.

Далее рассмотрим результаты бенчмарков для типа данных double. На Рис. 23 указано время выполнения всех методов, работающих на основе типа данных двойной точности Double.

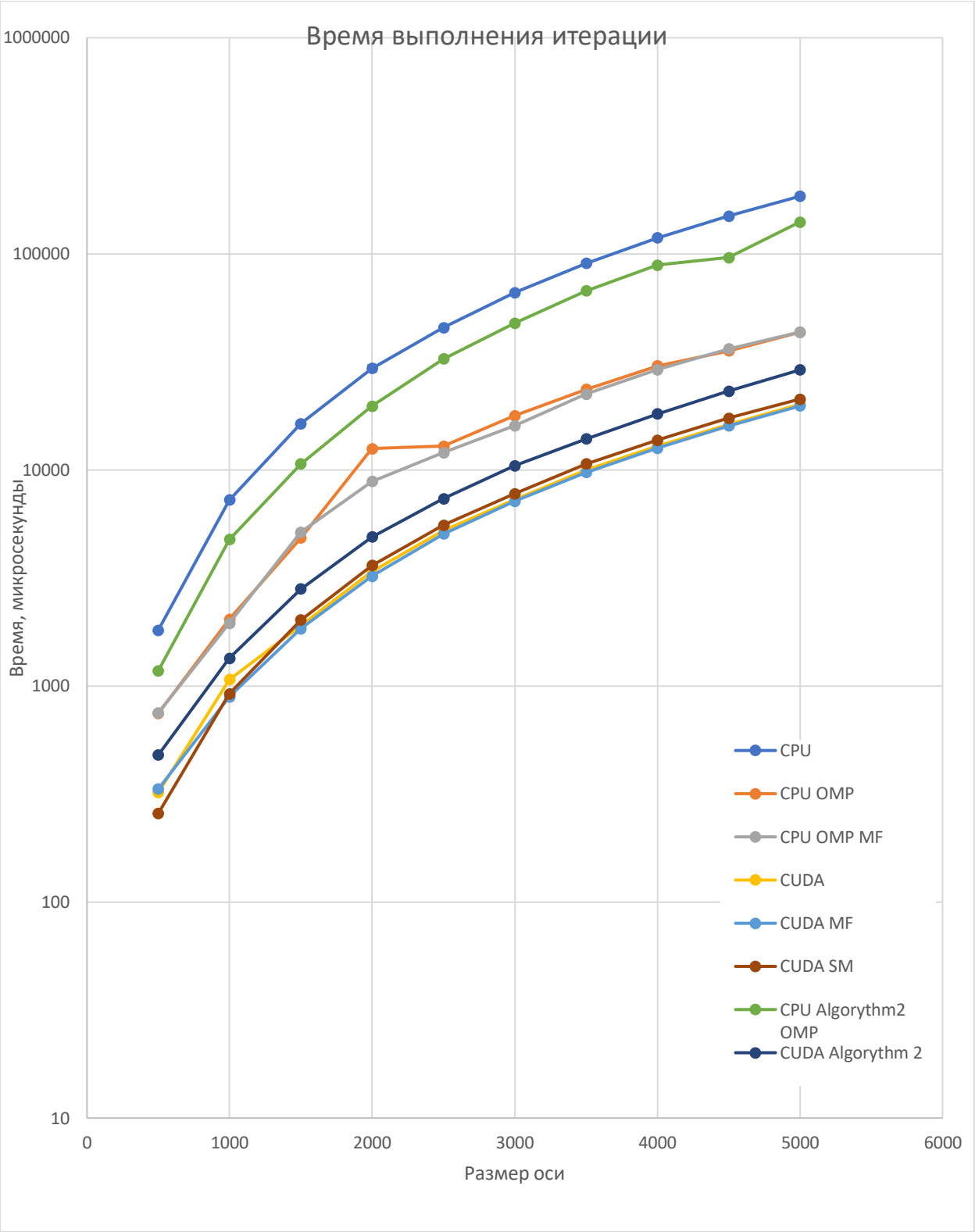

Рис. 23 Производительность методов, работающих с типом данных Double.

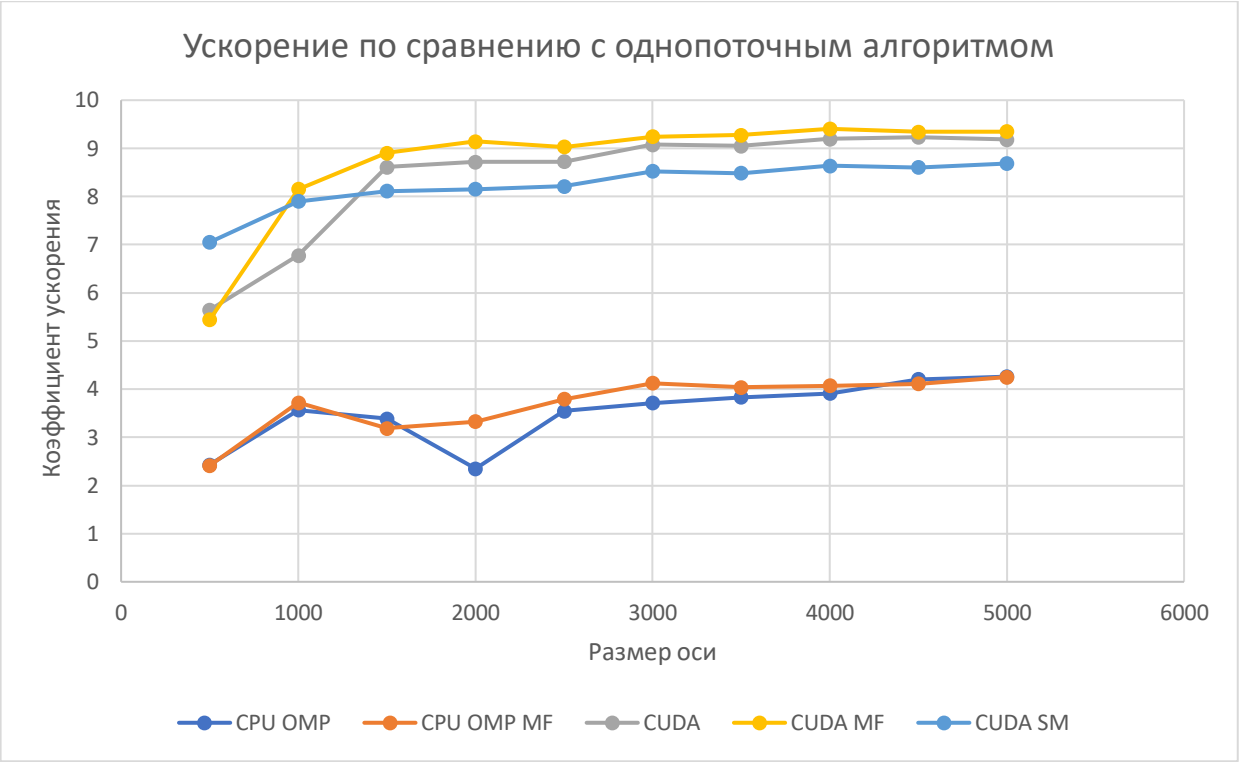

Рис. 24 Сравнение с однопоточным алгоритмом

Сравнивая график на Рис. 24 с графиком на Рис.17, замечаем, что при смене типа данных на double производительность методов на GPU сильно падает. Кроме того, при использовании типа данных float метод CUDA SM выигрывал по производительности у простого CUDA, а для типа данных double, он уже проигрывает. Аналогично, на Рис. 25 ускорение для метода, исполняемого на видеокарте, по сравнению с методом для CPU ниже, чем на Рис. 18.

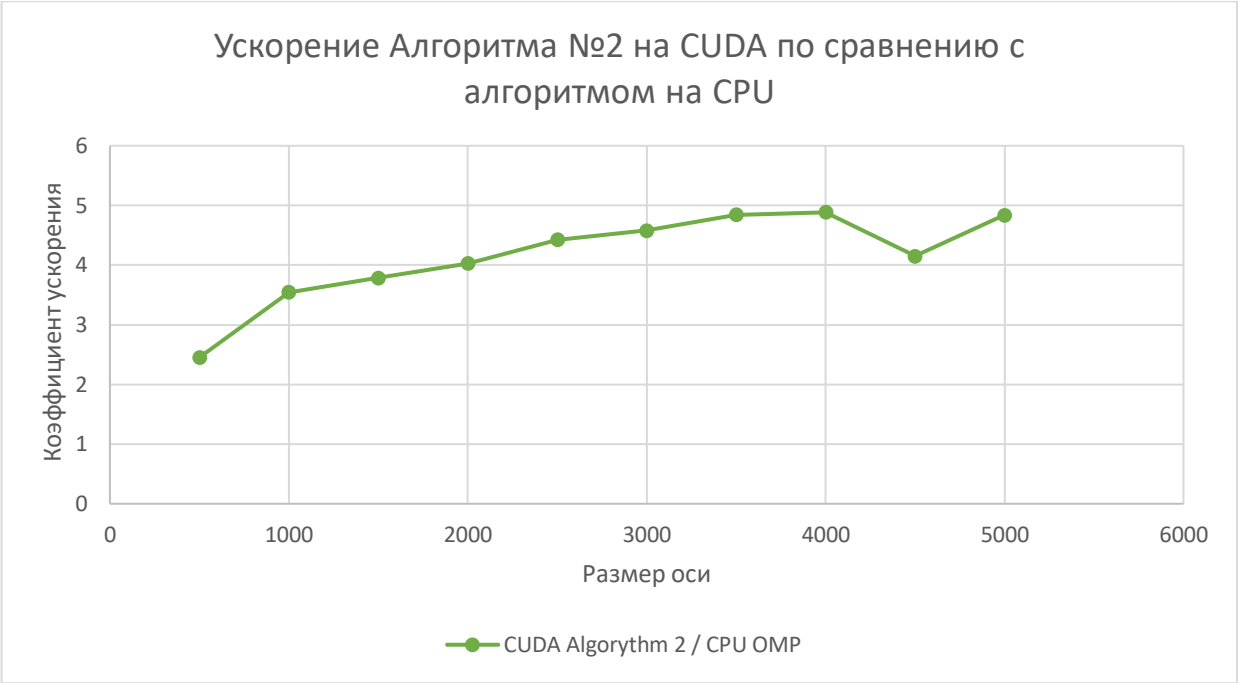

Рис. 25 Сравнение GPU алгоритма с CPU версией для алгоритма №2

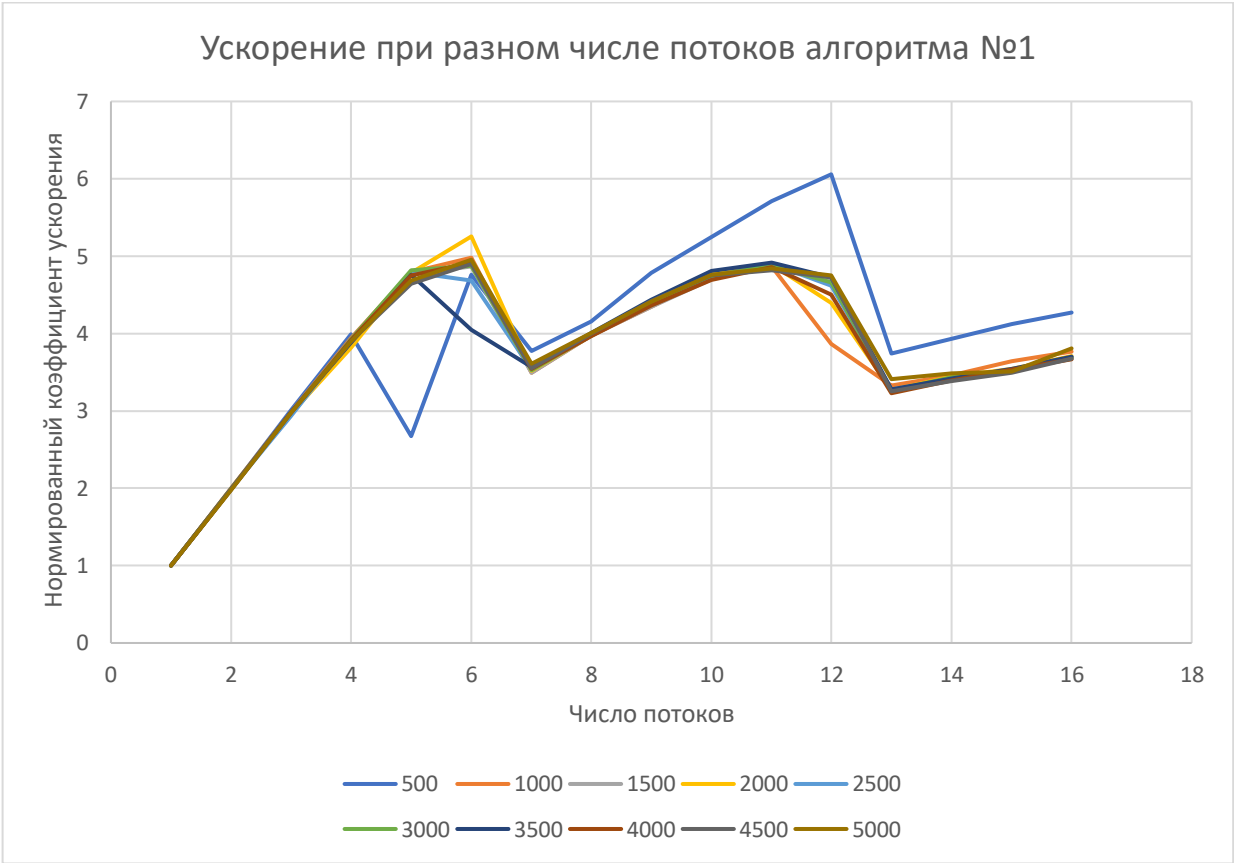

Рис. 26 Зависимость ускорения CPU алгоритма № 1 от числа потоков

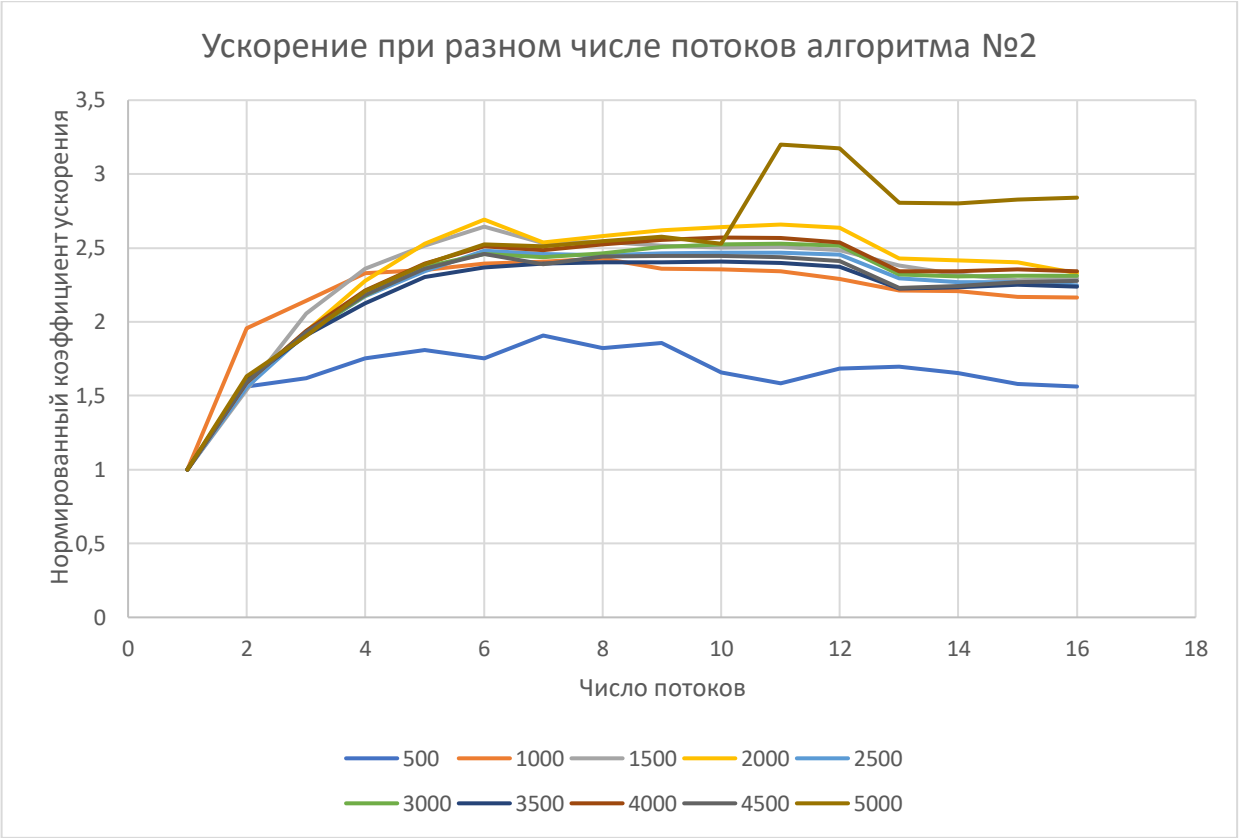

Рис. 27 Зависимость ускорения CPU алгоритма от числа потоков для алгоритма №2

На Рис. 26, 27 изображено ускорение алгоритмов на CPU при различном числе потоков. По сравнению с ускорением алгоритмов на Рис. 19, 20, работавших с типом Float, максимальное ускорение для типа Double ниже.

На Рис. 28, 29 изображено сравнение производительности CUDA методов для типа данных Double при разном размере стороны двумерного блока. Графики на различных размерах сетки нормированы относительно времени с размером блока, равного 32. Наблюдаются локальные экстремумы на значениях 8 и 16, но изменение размера с 32 не дает существенного прироста производительности.

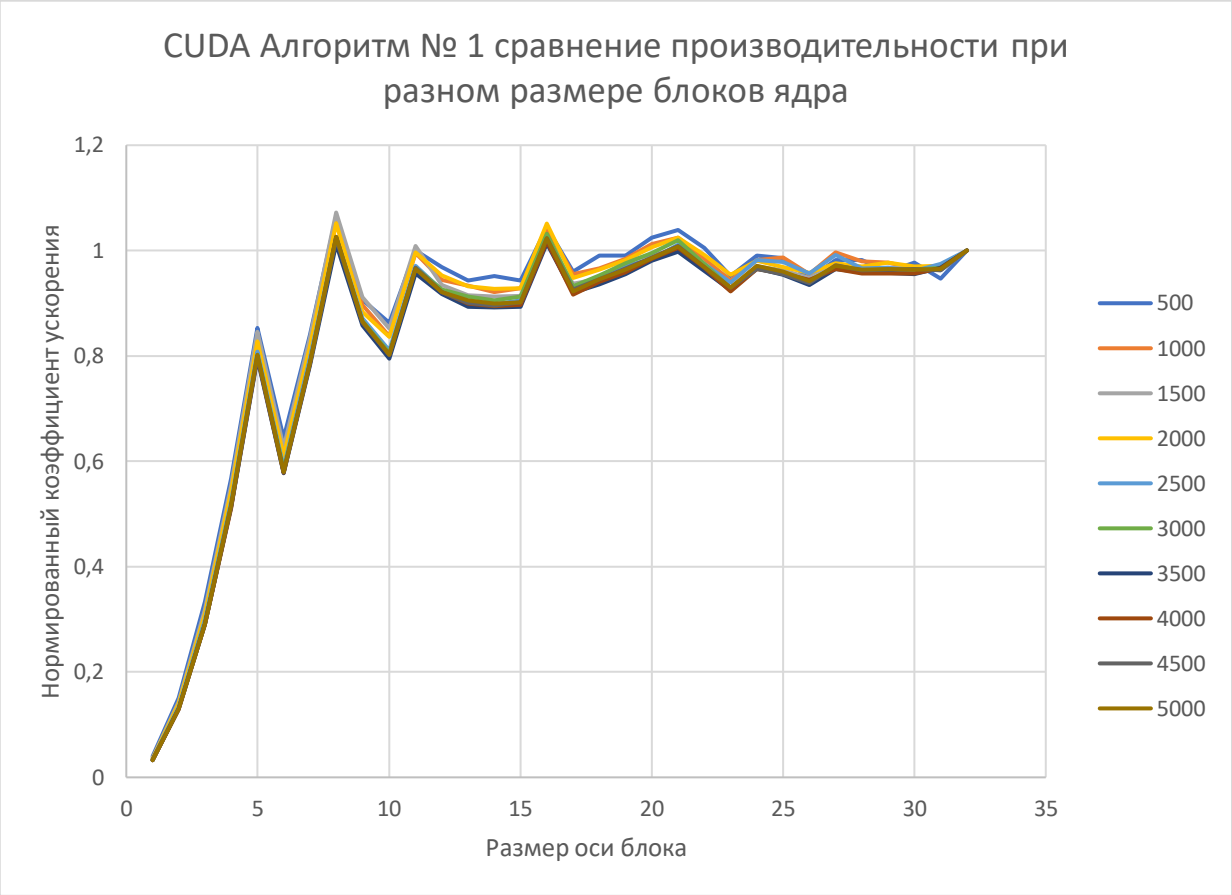

Рис. 28 Зависимость от размера блока ядра для первого алгоритма.

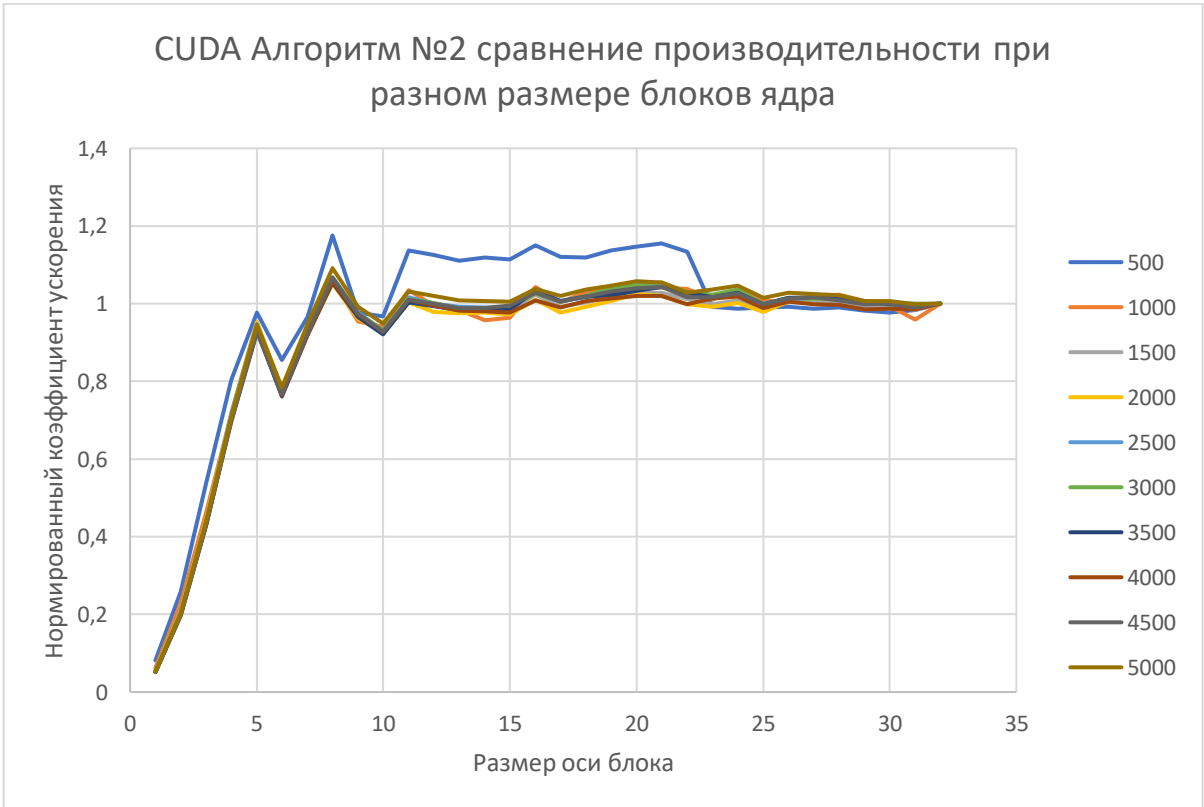

Рис. 29 Зависимость от размера блока ядра для алгоритма второго типа.

Сравним напрямую во сколько раз производительность методов на типе данных float выше, чем на типе данных double. По Рис. 30 видно, что методы, реализующие алгоритм №1, работающие на CPU, быстрее работают с типом данных double, а методы, реализующие алгоритм №2, несколько выигрывают от типа float. С другой стороны видно, что видеокарта гораздо менее эффективна при работе с типом данных double, чем при работе с типом float.

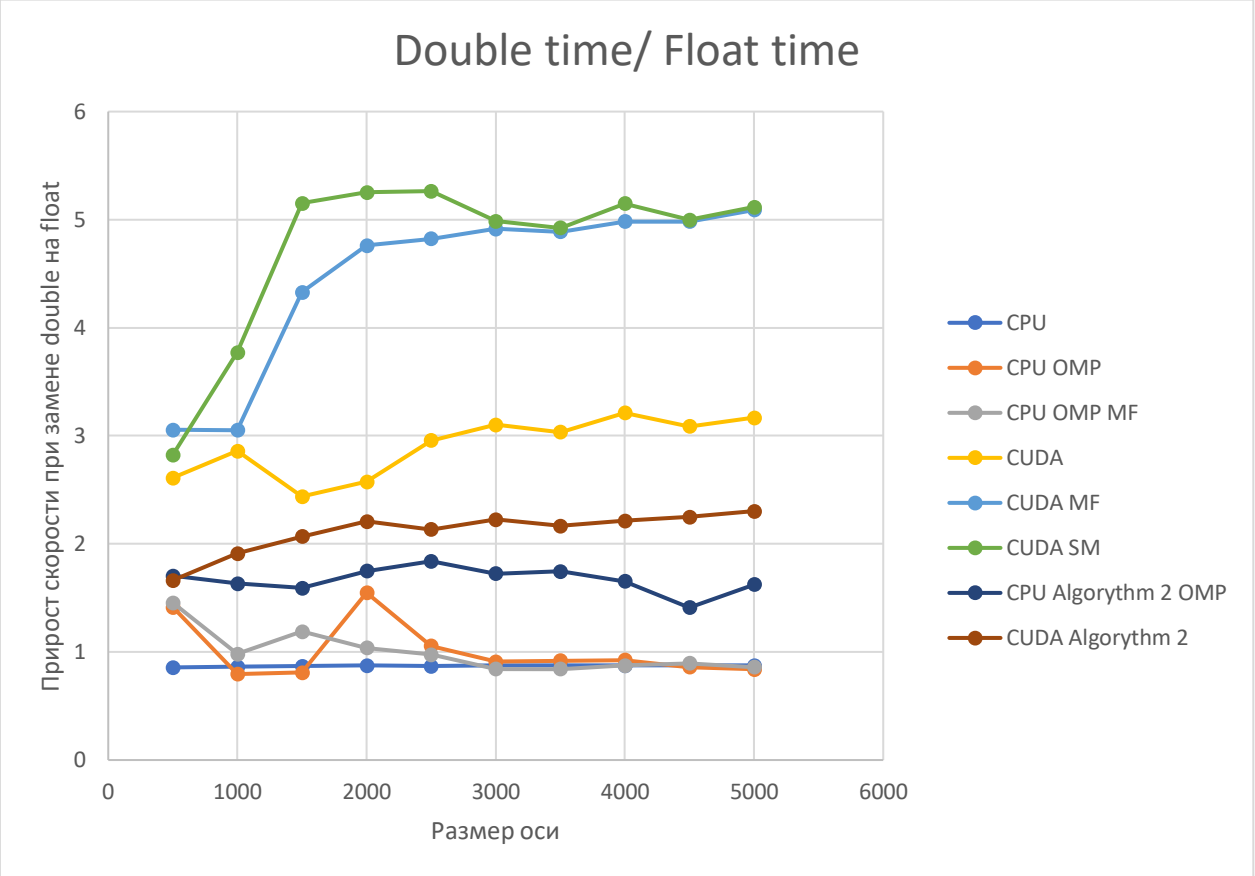

Рис. 30 Сравнения float и double версий алгоритмов

В приложении также рассмотрены результаты, полученные на вычислительной платформе от СПБГУ, на кластере GPUlab с характеристиками: центральным процессором AMD FX-8370 4.0 GHz, видеокартой GeForce GTX 1060. Количество OpenMP потоков для этой платформы выбрано равным 8.

По результатам тестирования получен стабильный прирост производительности при перенесении вычислений с процессора на видеокарту. Были применены некоторые техники по оптимизации кода, свойственные программам, написанным для платформы CUDA. Проведенный анализ данных показывает, что полученное ускорение для алгоритма №2 в сравнении с однопоточным алгоритмом не является максимально возможным, и требуется дальнейшая оптимизация. Во время работы над алгоритмами для GPU я применял профайлер Nsight Compute, который позволил повысить быстродействие кода, но также и показал, что при всей проделанной работе по оптимизации производительности, есть потенциал увеличить ее еще сильнее.

5. Проектирование нового алгоритма

В качестве нового алгоритма, потенциально работающего в реальном времени, мною предлагается улучшение существующих алгоритмов, основанных на трассировке лучей. Предлагаемые изменения:

- Данные о геометрии комнаты хранятся не в виде треугольных мешей, а в виде набора полей высот (heightfield). Одна комната может быть описана кубической картой высот с центром в центре комнаты. Так как часть геометрии комнаты может быть не видна из одной точки, и, вследствие, не будет описана полем высот, то предполагается использование многослойной карты высот. В качестве альтернативы можно использовать несколько кубических карт, расположенных в разных частях помещения, но первый вариант предпочтительнее ввиду программной поддержки многослойных текстур и поверхностей в CUDA.

- Вместе с данными о карте высот также хранятся данные о нормалях поверхности, и информацию об отражательной способности поверхности.
- Вместо технологии трассировки лучей используется технология Ray marching - шагания по лучу. Отличие от трассировки лучей состоит в том, что вместо поиска пересечения луча с геометрией сцены путем построения BVH (Bounding Volume Hierarchy, Иерархия ограничивающих объемов) [15] дерева и прохода по нему, алгоритм итеративно шагает по лучу на допустимое расстояние, которое вычисляется с помощью относительно простых операций. Вычисление этого допустимого расстояния можно проводить разными способами. Например, вычисление аналитически заданного расстояния со знаком (Signed Distance Function) до поверхности для набора примитивов [16]. Второй способ – кэширование значений SDF в трехмерной текстуре. Таким образом, для шага алгоритма Ray Marching надо вычислить координаты в пространстве объекта и прочитать трехмерную текстуру.
- В моем предложенном алгоритме используется не Signed Distance Function, а двумерная карта высот с использованием MIP-текстурирования. В каждый MIP уровень записывается максимальное из четырех значений текстелей на предыдущем уровне. При итерации алгоритма происходит чтение из иерархической карты высот, благодаря чему луч проходит большее расстояние, если он отдален от поверхности, и идет более точно, если он близок к ней. Подход в алгоритме аналогичен алгоритму из источника [17].

Применение алгоритма Ray Marching вместо Ray Tracing обосновано тем, что до недавнего времени в приложениях с интерактивной графикой алгоритм Ray Tracing практически не использовался, ввиду вычислительной сложности. В то же время алгоритмы ray marching успешно использовались

для рендеринга мягкого затенения [18], непрямого диффузного освещения, отражений в пространстве экрана [19], атмосферных явлений (освещения облаков, туманов) [20]. С выходом видеокарт Nvidia 2000 серии в 2018 году, началось применение аппаратного ускорения операций трассировки лучей, в связи с чем количество приложений, использующих этот алгоритм в интерактивном режиме, значительно повысилось. Но алгоритмы Ray marching остаются предпочтительнее в случае, если видеокарта не поддерживает аппаратного ускорения трассировки лучей.

Рассмотрим подробнее алгоритм прохода по лучу с использованием карт высоты:

Подготовка входных данных для алгоритма:

1. Загрузка карты высот с размерами $N \times N$, где N – является степенью двойки.
2. Заполнение уровней MIP таким образом, чтобы 4 значения редуцировались в одно – максимальное из четырех. Значения из текстуры возвращаются как числа типа float от 0.0 до 1.0.

Алгоритм итераций:

1. Начальный уровень MIP выбирается самый последний (тот, который размером 1×1 пикселей).
2. Проверка, не вышел ли луч за границы объема, в котором задана карта высот. Если вышел, то выход из алгоритма, возврат значения -1.0.
3. Выбор границ xlim и ylim (переменные типа float2) как границы пикселя на текущем уровне MIP, над которым находится луч.
4. Чтение значения Z из текстуры с текущего уровня MIP и текущего положения на луче.
5. Если полученное значение больше, чем высота от текущего положения до плоскости, то выбирается более низкий уровень MIP, возврат на шаг 2). Если луч под поверхностью даже на

нулевом уровне MIP, то значит, что пересечение луча с поверхностью найдено, переход к шагу 12.

6. После нахождения уровня MIP, на котором луч находится над поверхностью, и чтения значения Z из текстуры, строим пересечение луча с плоскостями $x = xlim(1),\ x = xlim(2),\ y = ylim(1),\ y = ylim(2),\ z = Z,\ z = 1.0$
7. Переместить луч в точку первого (ближайшего) пересечения с плоскостями.
8. Если луч пересекает плоскость $z = Z$, то текущий уровень MIP уменьшается (происходит шаг внутрь).
9. Иначе, если луч пересекает одну из граничных плоскостей, то текущий уровень MIP не уменьшается (шаг в сторону).
10. Если происходит два шага в сторону подряд, то уровень MIP увеличивается (шаг наружу).
11. Возврат к шагу 2, если не достигнуто максимальное количество итераций.
12. Алгоритм возвращает пройденное лучом расстояние.

Так как максимальное количество итераций ограничено, то оно может закончиться пока луч на полпути к своей реальной точки пересечения. Такое может случиться, если луч прошел близко к поверхности по касательной, но не пересек ее в этом месте. В таком случае есть смысл возвращать из алгоритма значение, соответствующее не концу луча, а наиболее близкой к поверхности точки на луче.

В качестве информации, описывающей акустические свойства материала, могут использоваться коэффициенты отражения для 4 диапазонов частот. Для каждого дошедшего до источника луча получаем четыре коэффициента отражения (полученные через перемножение сэмплов коэффициентов во всех точках отражений на пути луча) и длину луча. По значениям четырех коэффициентов для каждого направления строится ядро

свертки в частотном диапазоне, к которой применяется свертка функции HRTF для данного направления первичного луча. Последним этапом все полученные ядра свертки редуцируются в выходную стерео-свертку, применяющуюся к звуку источника. Операции по применению функций HRTF также можно ускорить с помощью GPU.

Рис. 31 Примеры рендера карты.

На Рис. 31 показаны примеры работы алгоритма шагания по лучу для текстуры кирпича размером 2048×2048 пикселей. Размеры окна рендеринга 1000×1000 пикселей. Неоптимизированная версия алгоритма работает за 300–400 микросекунд на кадр. Текущая версия находит только первое пересечение с поверхностью. Код доступен в репозитории GitHub[4].

Заключение

Были изучены современные подходы к решению задачи моделирования акустических свойств помещений, а также были реализованы некоторые из этих алгоритмов. Было проведено прямое сравнение производительности метода конечных разностей на CPU и GPU. Сделан вывод, что задача хорошо подходит для распараллеливания вычислений на видеокарте. Для звуковой системы на основе алгоритма Ray Marching был создан графический прототип, реализующий только основной алгоритм шагания по лучу.

[4] https://github.com/zarond/RayMarchCUDA

На сегодняшний день нет единого подхода к решению данной задачи, который бы был одновременно быстрым, интерактивным и высокоточным. Даже самые высокоточные методы не справляются с симуляцией на частоте 44100 Гц и требуют восстановления и увеличение частоты дискретизации вычисленной импульсной характеристики и их постобработку, иначе такая IR имеет свойства фильтровать частоты, которые выше половинной частоты симуляции. Поэтому существует подход комбинировать результаты волновых методов для низких частот и геометрических методов для высоких.

Перспективы развития

Мной планируется дальнейшее изучения методов и написание алгоритмов для платформы CUDA. Написанные на данный момент мной алгоритмы являются наивной имплементацией рассмотренных методов, и подлежат дальнейшей оптимизации. Планируется переход к задаче в трехмерном пространстве и учёт явлений дифракции в геометрических методах. Также в рамках данного исследования не были рассмотрены задачи описания объемного звука и зависимость импульсных характеристик от ориентации слушателя в пространстве.

1. Реализация задуманного алгоритма Ray Marching.
2. Интеграция с игровым движком Unity3D.
3. Использования технологии аппаратного ускорения трассировки лучей RTX.
4. Использование вычислительных шейдеров вместо технологии CUDA.

Список использованных источников:

[1] Мареев В. В., Станкова Е. Н. Основы методов конечных разностей. – 2012. // URL: http://www.apmath.spbu.ru/ru/staff/stankova/publ/publ1.pdf

[2] Pulkki V., Lokki T., Savioja L. Implementation and visualization of edge diffraction with image-source method //Audio Engineering Society Convention 112. – Audio Engineering Society, 2002.
// URL:
https://www.researchgate.net/publication/236169356_Implementation_and_visualization_of_edge_diffraction_with_image-source_method

[3] Cao C. et al. Interactive sound propagation with bidirectional path tracing //ACM Transactions on Graphics (TOG). – 2016. – Т. 35. – №. 6. – С. 1-11.
// URL: http://kunzhou.net/zjugaps/bst/bst.pdf

[4] Siltanen S., Lokki T., Savioja L. Room acoustics modeling with acoustic radiance transfer //Proc. ISRA Melbourne. – 2010.
// URL:
https://www.acoustics.asn.au/conference_proceedings/ICA2010/cdrom-ISRA2010/Papers/P5h.pdf

[5] https://odeon.dk/
Christensen C. L., Rindel J. H. A new scattering method that combines roughness and diffraction effects //Forum Acousticum, Budapest, Hungary. – 2005. – С. 344-352.
// URL: https://odeon.dk/pdf/CLC%20fa2005.pdf
[Электронный ресурс] URL: https://odeon.dk/

[6] Raghuvanshi N., Narain R., Lin M. C. Efficient and accurate sound propagation using adaptive rectangular decomposition //IEEE Transactions on Visualization and Computer Graphics. – 2009. – Т. 15. – №. 5. – С. 789-801.
// URL: http://gamma.cs.unc.edu/propagation/main.pdf

[7] Chern A. A Reflectionless discrete perfectly matched layer //Journal of Computational Physics. – 2019. – Т. 381. – С. 91-109.
// URL: https://arxiv.org/pdf/1804.01390.pdf

[8] Mehra R. et al. An efficient GPU-based time domain solver for the acoustic wave equation //Applied Acoustics. – 2012. – Т. 73. – №. 2. – С. 83-94.
// URL:
https://www.researchgate.net/publication/224035500_An_efficient_GPU-based_time_domain_solver_for_the_acoustic_wave_equation

[9] Raghuvanshi N., Snyder J. Parametric directional coding for precomputed sound propagation //ACM Transactions on Graphics (TOG). – 2018. – Т. 37. – №. 4. – С. 1-14.
// URL: https://www.microsoft.com/en-us/research/uploads/prod/2018/10/paramd.pdf

[10] Rosen M., Godin K. W., Raghuvanshi N. Interactive sound propagation for dynamic scenes using 2D wave simulation //Computer Graphics Forum. – 2020. – Т. 39. – №. 8. – С. 39-46.
// URL: https://www.microsoft.com/en-us/research/uploads/prod/2020/08/Planeverb_CameraReady_wFonts.pdf

[11] Tony Scudiero – Audio Tech Lead – NVIDIA
// URL:https://on-demand.gputechconf.com/gtc/2017/presentation/s7135-tony-scudiero-nvidia-vrworks-audio-improving-vr.pdf
[Электронный ресурс] // URL:https://developer.nvidia.com/vrworks/vrworks-audio

[12] CARL WAKELAND, ADVANCED MICRO DEVICES, INC. LAKULISH ANTANI, VALVE CORPORATION
// URL:https://gpuopen.com/gdc-presentations/2019/gdc-2019-s3-powering-spatial-audio.pdf
[Электронный ресурс] // URL: https://gpuopen.com/true-audio-next/

[13] CUDA C++ Programming Guide
[Электронный ресурс] // URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[14] Todd L. Veldhuizen. 1998. Arrays in Blitz++. In Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE '98). Springer-Verlag, Berlin, Heidelberg, 223–230.
// URL: https://dl.acm.org/doi/10.5555/646894.709708

[15] Bounding Volume Hierarchies
[Электронный ресурс] // URL: https://www.pbr-book.org/3ed-2018/Primitives_and_Intersection_Acceleration/Bounding_Volume_Hierarchies

[16] Signed Distance Functions Guide
[Электронный ресурс] // URL:
https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm

[17] Tevs A., Ihrke I., Seidel H. P. Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering //Proceedings of the 2008 symposium on Interactive 3D graphics and games. – 2008. – С. 183-190.
// URL: http://manao.inria.fr/perso/~ihrke/Publications/i3d08_lowres.pdf

[18] Daniel Wright, Dynamic Occlusion with Signed Distance Fields // SIGGRAPH 2015
[Электронный ресурс] // URL:
http://advances.realtimerendering.com/s2015/DynamicOcclusionWithSignedDistanceFields.pdf

[19] Erik Sintorn. Screen-Space Reflections, Seminar Course - DAT205/DIT226 Advanced Computer Graphics 2018
[Электронный ресурс] // URL:
http://www.cse.chalmers.se/edu/year/2018/course/TDA361/Advanced%20Computer%20Graphics/Screen-space%20reflections.pdf

[20] Andrew Schneider, Nathan Vos. The real-time volumetric cloudscapes of horizon zero-dawn. // SIGGRAPH 2015
[Электронный ресурс] // URL: https://www.guerrilla-games.com/read/the-real-time-volumetric-cloudscapes-of-horizon-zero-dawn

ПРИЛОЖЕНИЕ

Перечень использованного оборудования:

1. Персональный компьютер с характеристиками: процессор AMD Ryzen 5 4600H 3.00 GHz, видеокарта Nvidia GeForce GTX 1650Ti.
2. Вычислительный кластер GPUlab от СПБГУ с характеристиками: процессор AMD FX-8370 4.0 GHz, видеокарта Nvidia GeForce GTX 1060.

Результаты бенчмарков на кластере GPUlab:

Результаты для типа данных float:

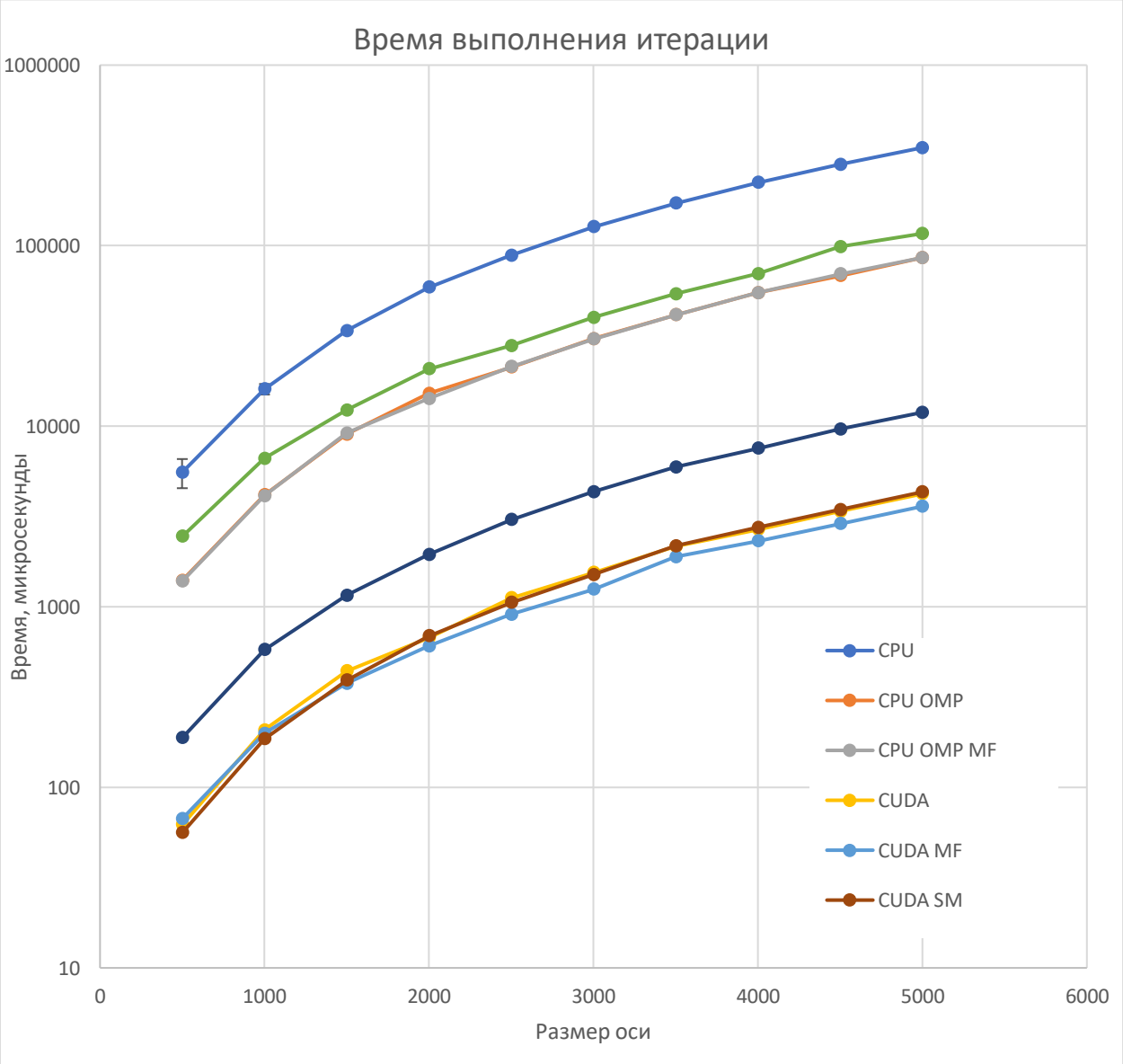

Рис. 32 Время выполнения одной итерации алгоритма в микросекундах в зависимости от размера сетки, логарифмический масштаб по вертикальной оси

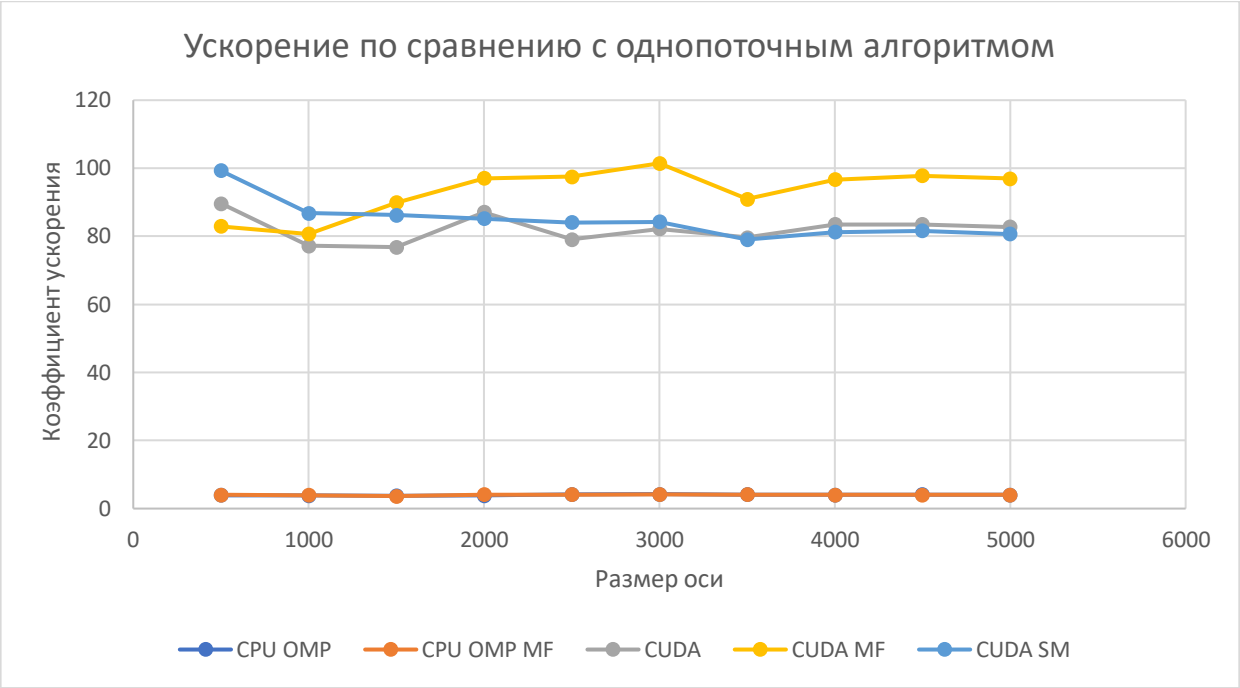

Рис. 33 Сравнение с однопоточным алгоритмом

По Рис. 33 можно наблюдать, что производительность лучшего метода на CPU более чем в 22 раз меньше, чем производительность лучшего метода на GPU. Видеокарта, установленная на кластере, дает больший прирост производительности, чем видеокарта, установленная на моем ПК. Однако, в этом случае использование общей памяти в методе CUDA SM не дает выигрыш по сравнению с обычным методом CUDA.

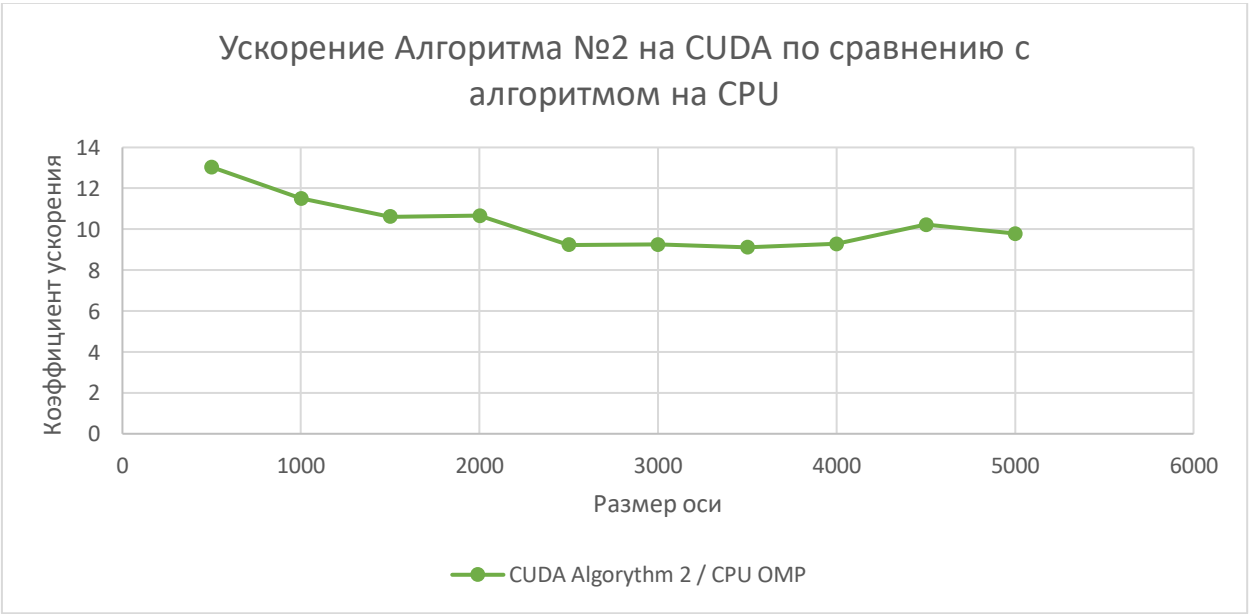

Рис. 34 Сравнение GPU алгоритма с CPU версией для алгоритма №2

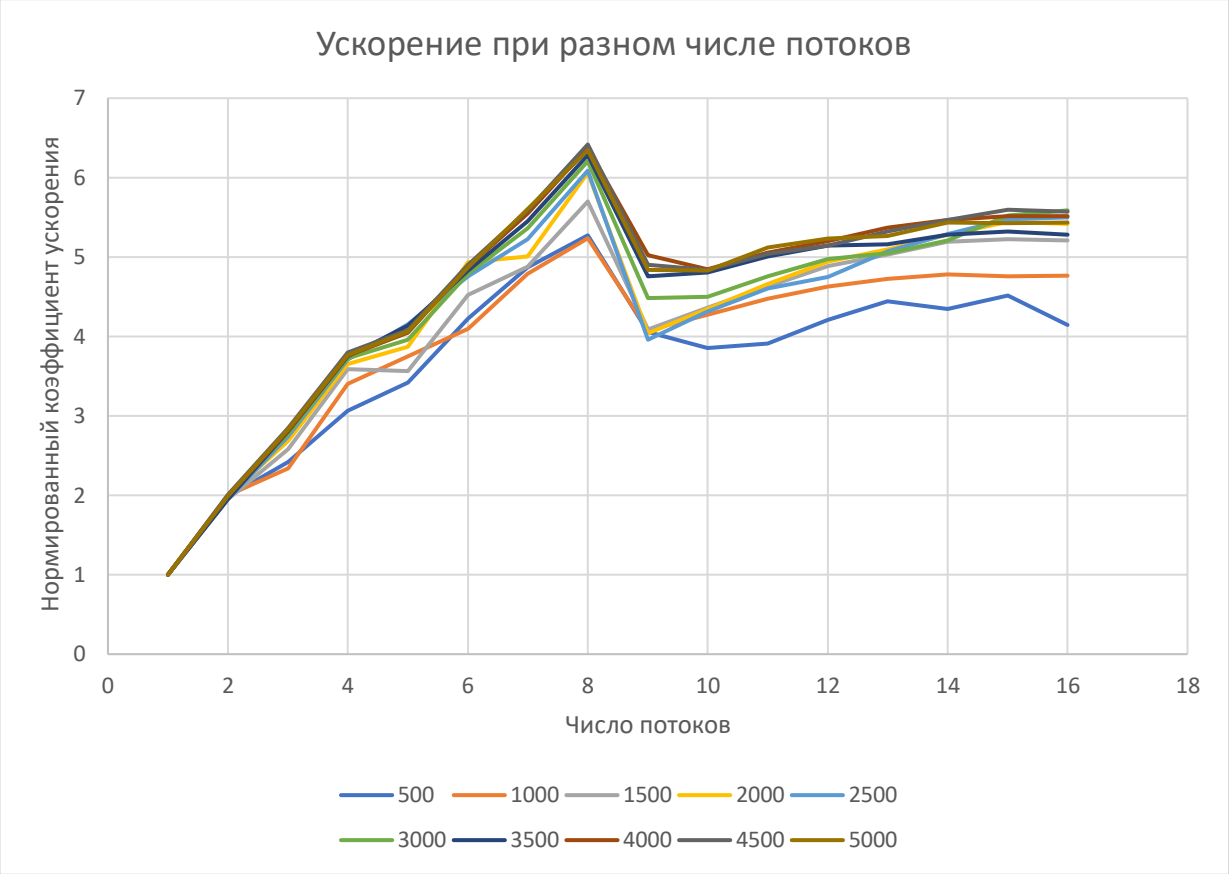

Рис. 35 Зависимость ускорения CPU алгоритма от числа потоков. По графику видно, что CPU обладает 8 логическими потоками.

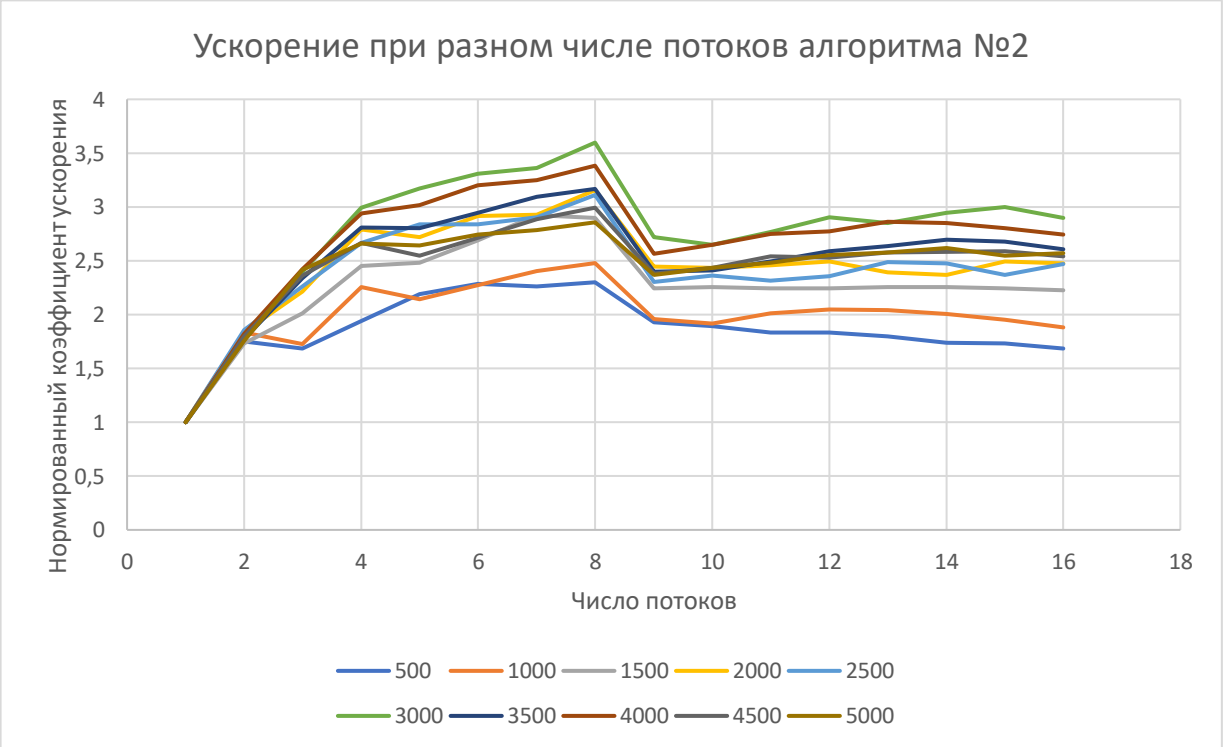

Рис. 36 Зависимость ускорения CPU версии метода от числа потоков для алгоритма №2

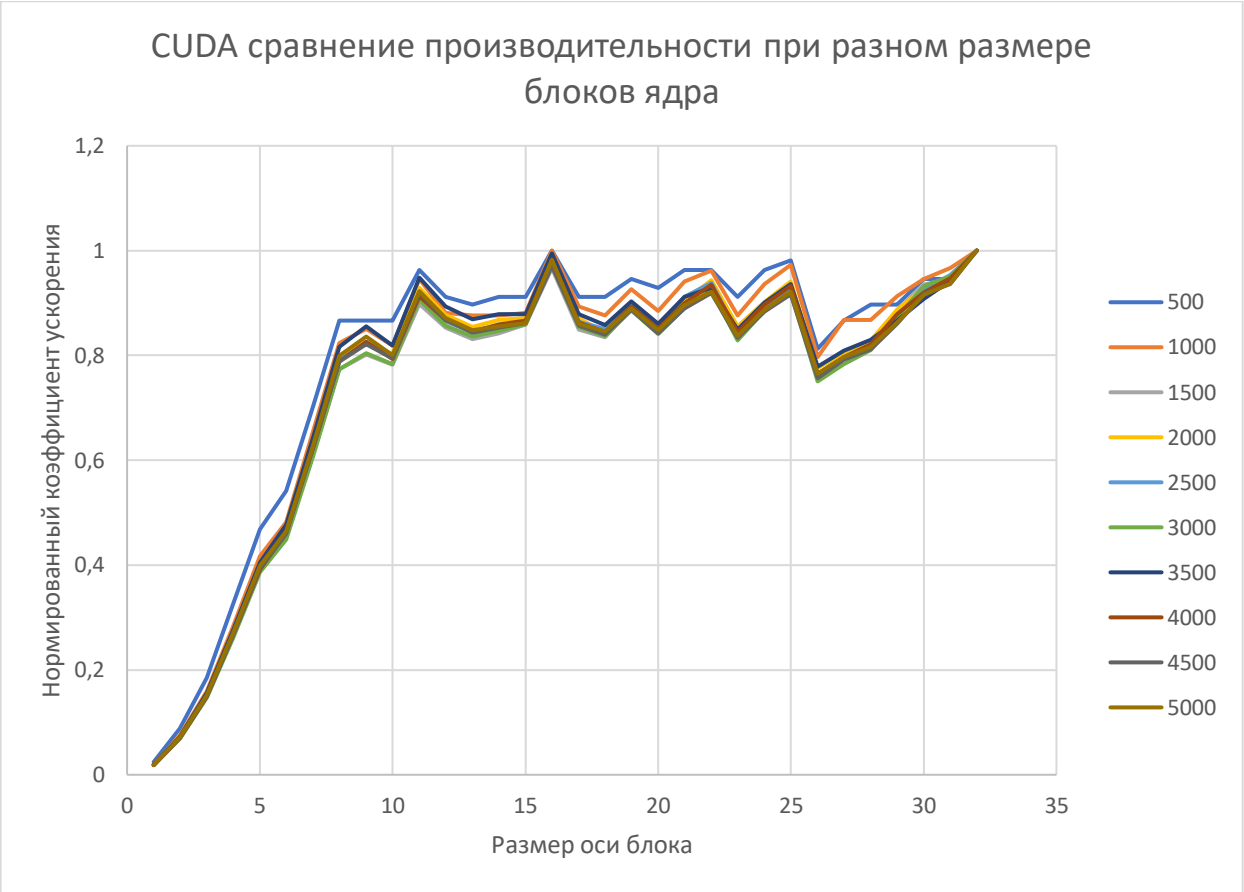

Рис. 37 Зависимость от размера блока ядра.

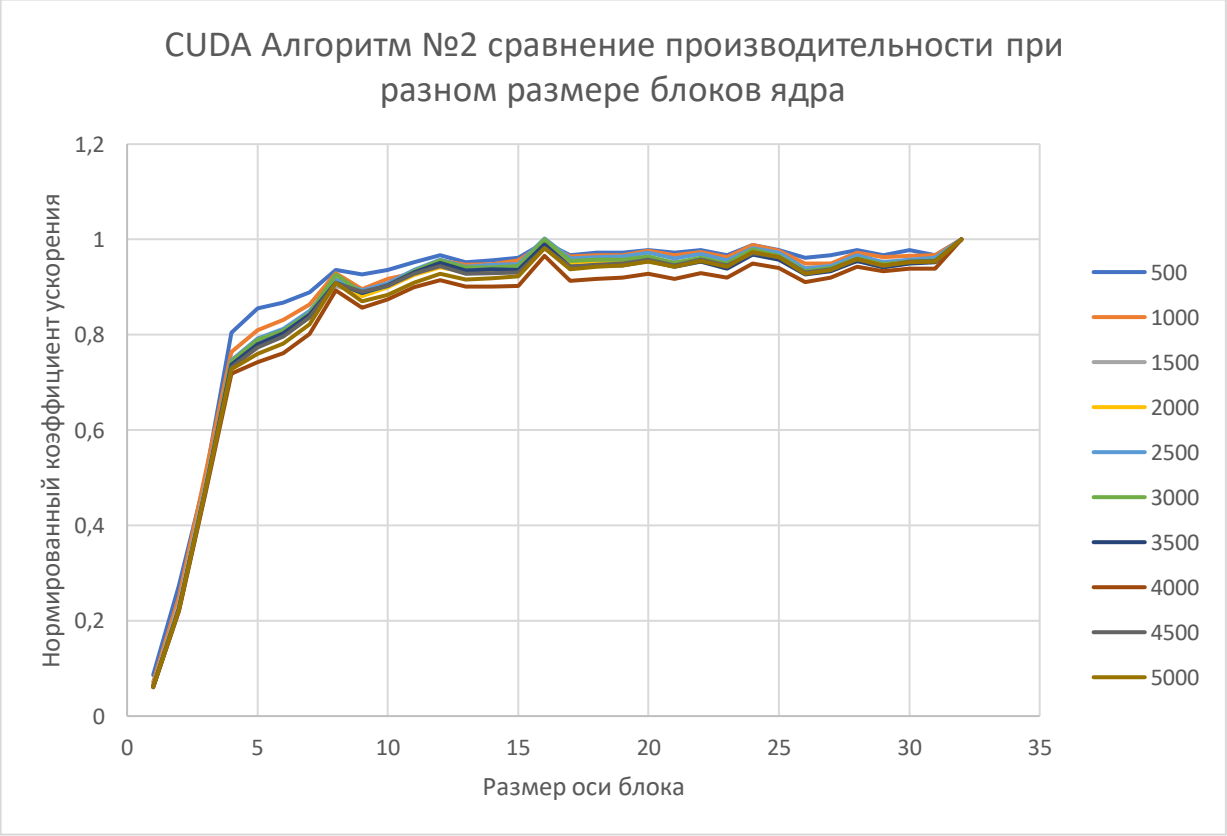

Рис. 38 Зависимость от размера блока ядра. Размер блока 32 дает наилучший результат.

Далее рассмотрим результаты бенчмарков для программы, работающей с типом данных double:

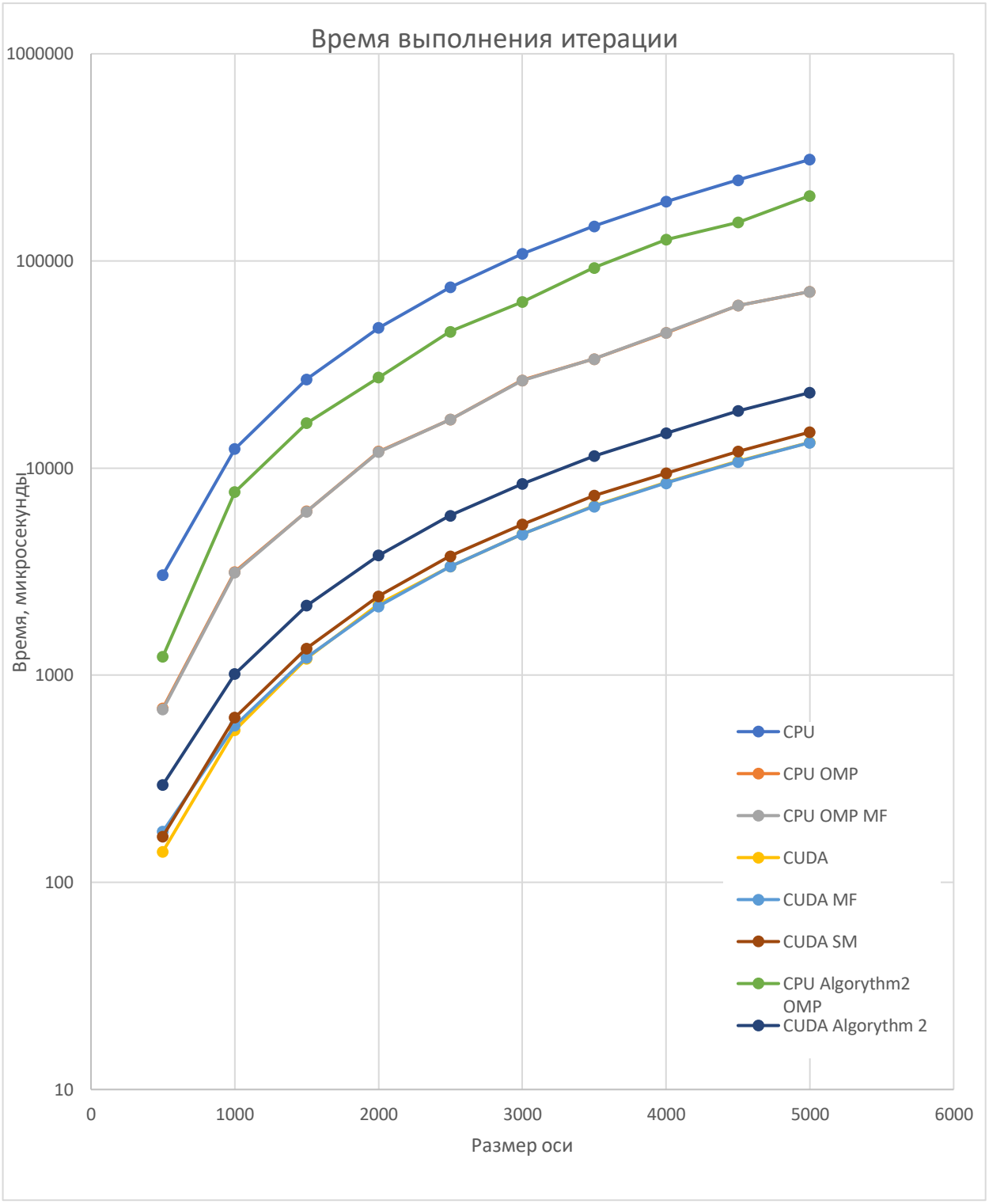

Рис. 39 Время выполнения одной итерации алгоритма в микросекундах в зависимости от размера сетки, логарифмический масштаб по вертикальной оси

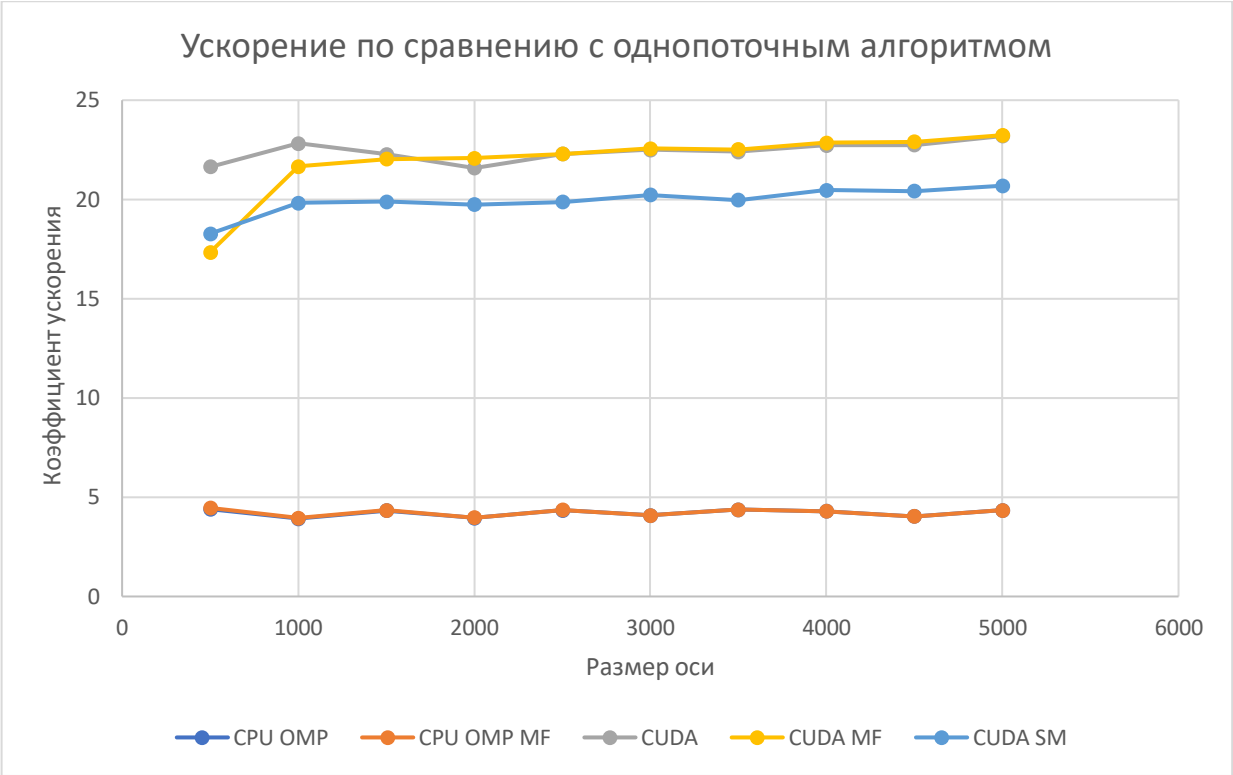

Рис. 40 Сравнение с однопоточным алгоритмом

Для типа данных double метод с общей памятью проигрывает остальным GPU методам.

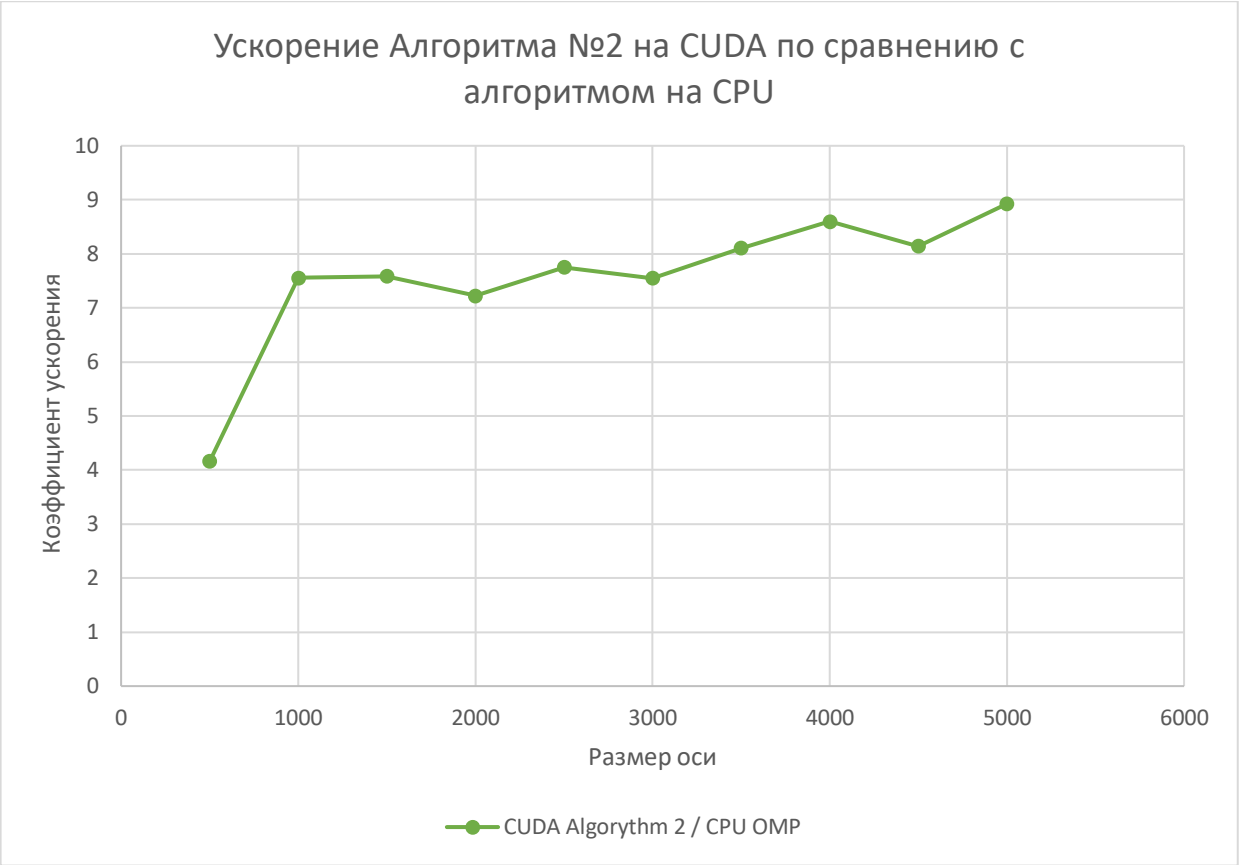

Рис. 41 Сравнение GPU алгоритма с CPU версией для алгоритма №2

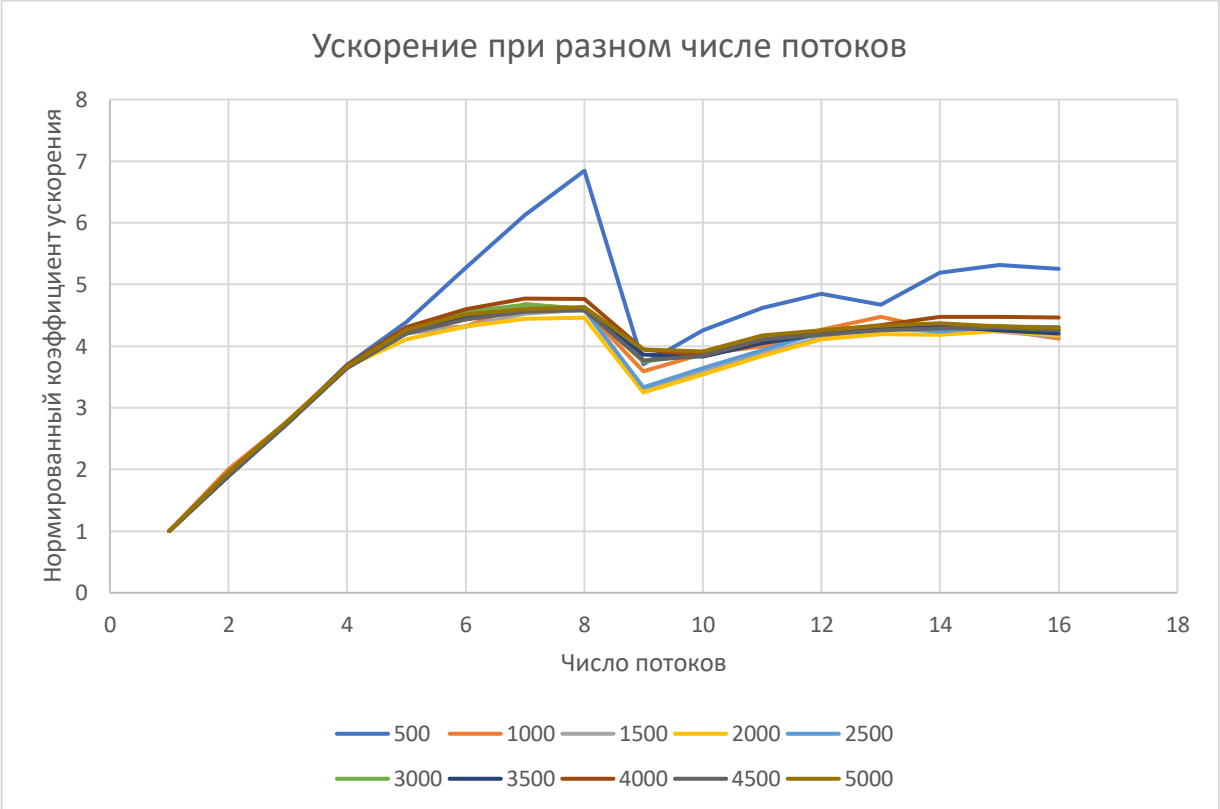

Рис. 42 Зависимость ускорения CPU алгоритма от числа потоков

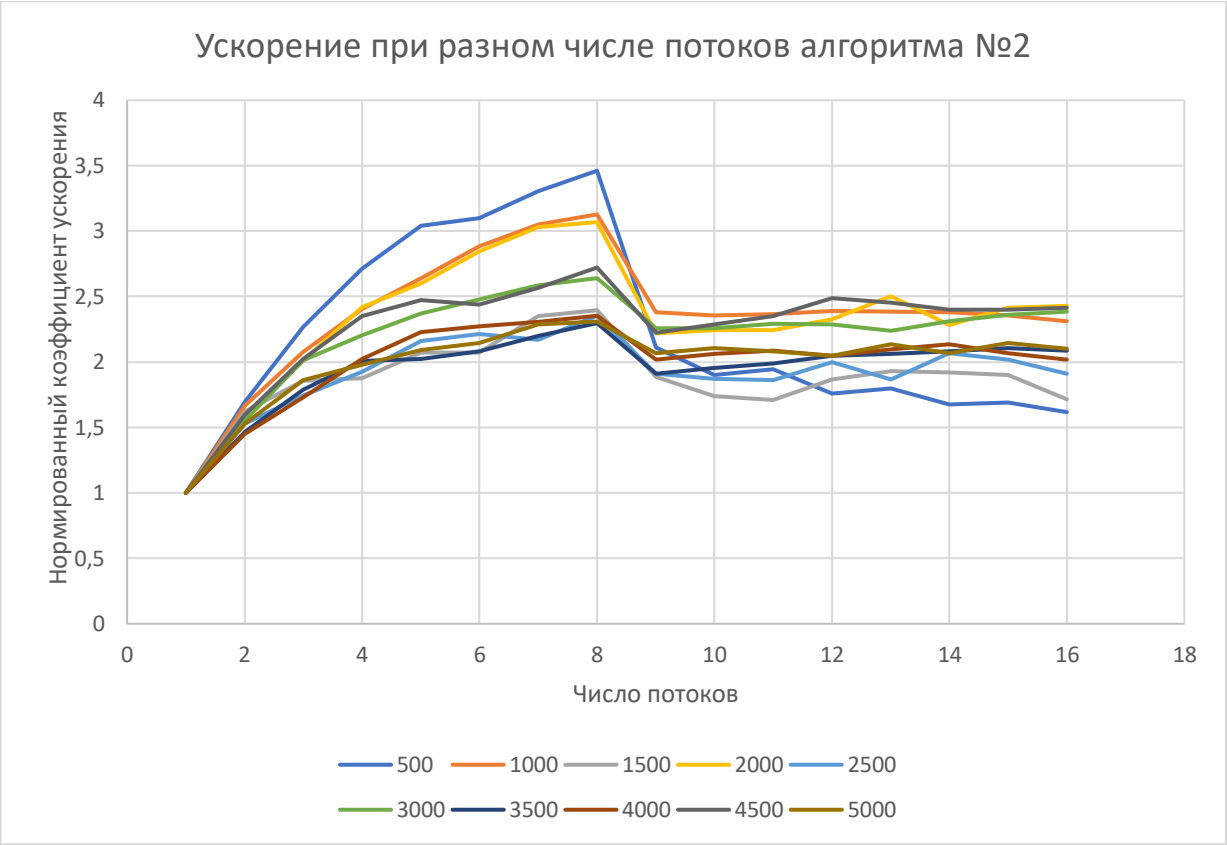

Рис. 43 Зависимость ускорения CPU алгоритма от числа потоков для алгоритма №2

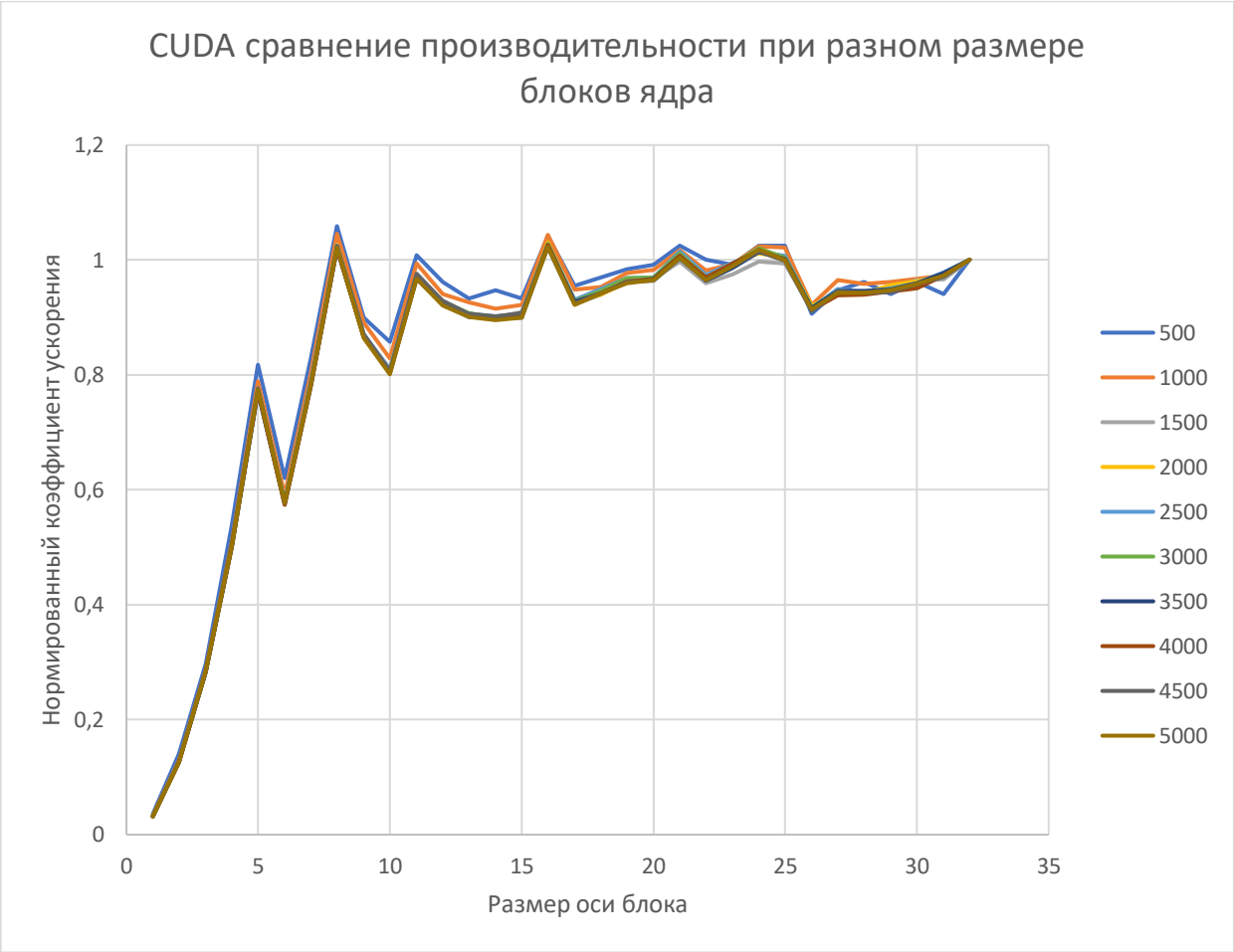

Рис. 44 Зависимость производительности GPU алгоритма от размеров блока

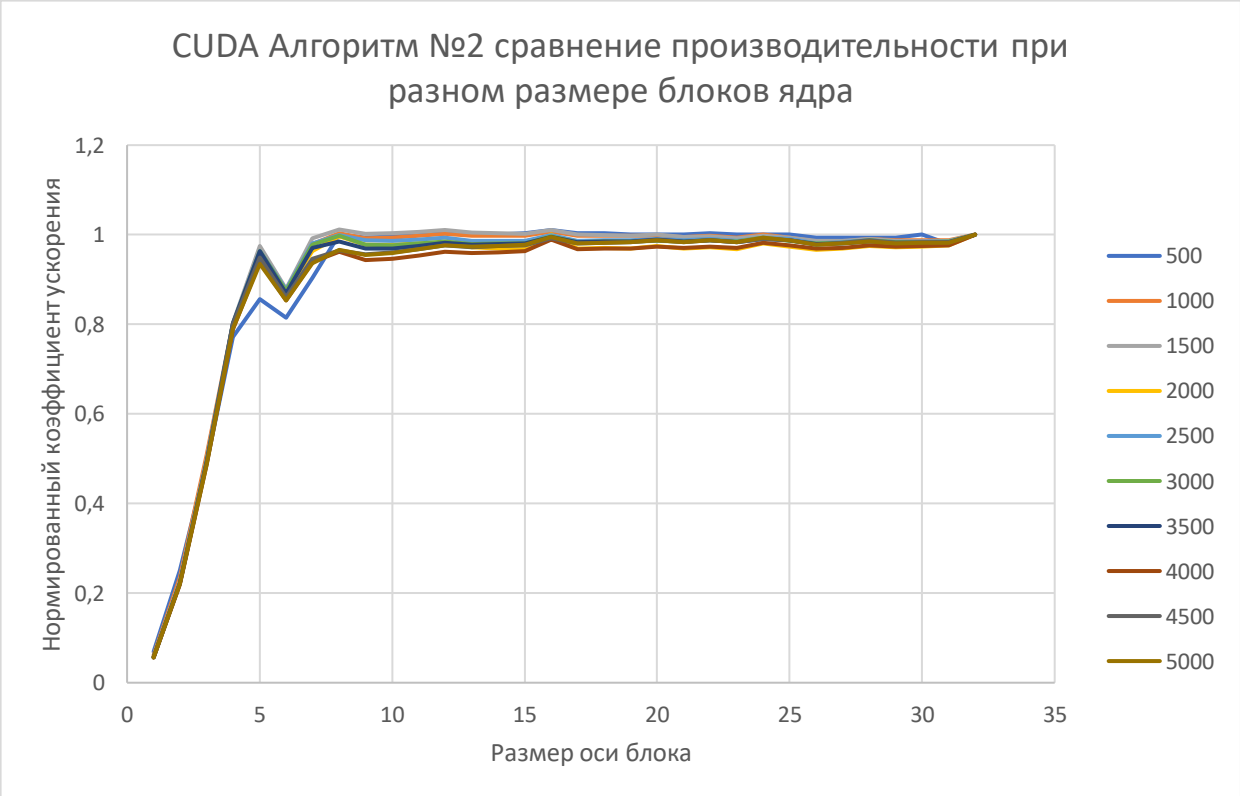

Рис. 45 Зависимость производительности GPU алгоритма от размеров блока

Сравнение производительности методов, работающих с разными типами данных на кластере GPUlab:

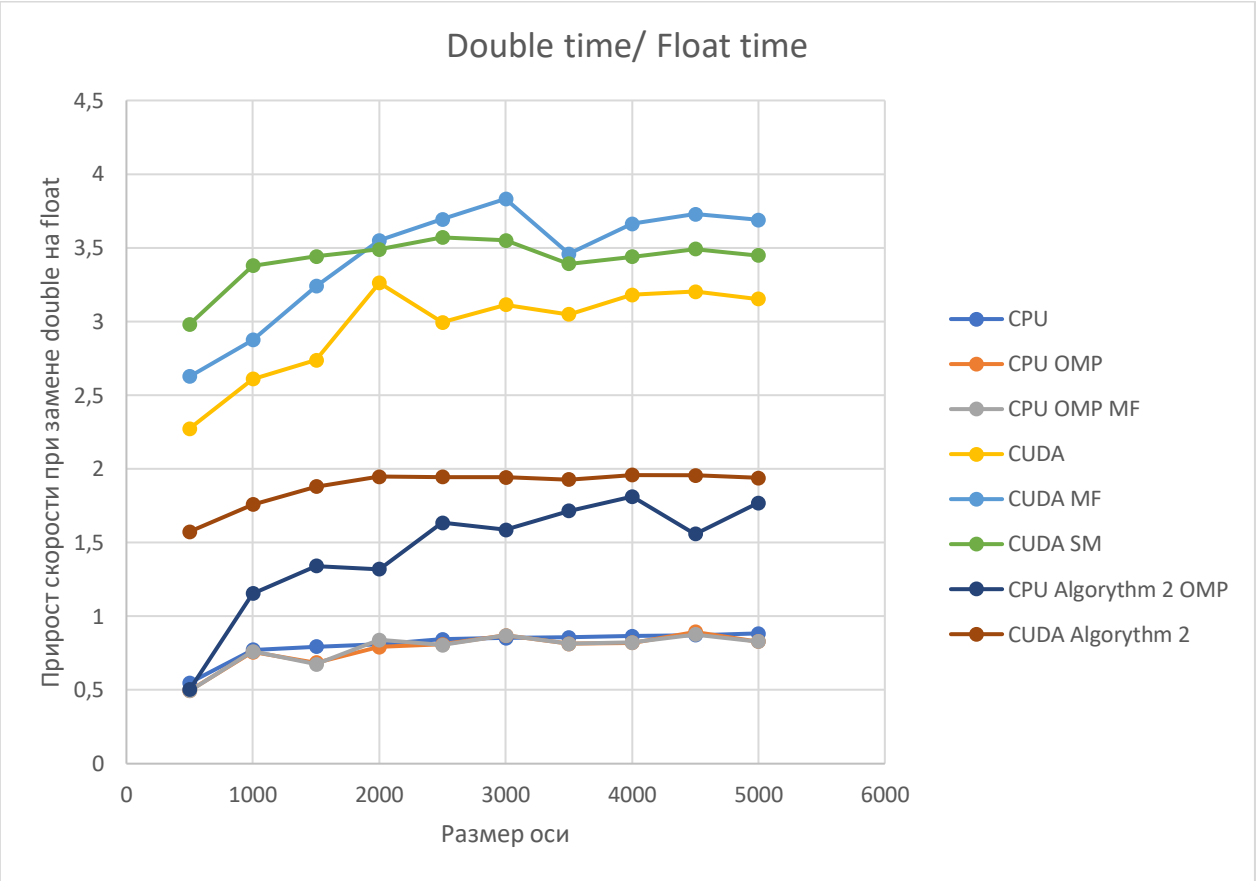

Рис. 46 Сравнения float и double версий алгоритмов. Наблюдается падение производительности, схожее с таковым на моем компьютере.