САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Направление: 02.03.02 «Фундаментальная информатика и информационные технологии
ООП: Программирование и информационные технологии

ОТЧЕТ ПО НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ ПРАКТИКЕ

Тема задания: Сравнение производительности методов моделирования распространения звука в замкнутом помещении с использованием вычислений на процессоре и на видеокарте

Выполнил: Макоев Артур Аланович, группа 17.Б11-пу

Руководитель практики от организации: Ганкевич Иван Геннадьевич, доцент

Руководитель практики от СПБГУ:

Погожев Сергей Владимирович, кандидат физ.-мат. наук, доцент

Санкт-Петербург
2021

Содержание

1. Введение

Цель работы – провести подробное 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[1].

2. 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 среды 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 среды 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 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.

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

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 OMP 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 долгое время работы. Однако, этот режим полезен тем, что он показывает время вычисления для конкретной задачи.

Для работы с данными использовались массивы из библиотеки blitz. 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[2]. 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

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

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 напрягали 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.

3. 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:

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

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

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

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

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 [1]. 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:

$$p^+ = B(p - C\nabla \cdot \bar{v}), \tag{3}$$

$$v_x^+ = BB_<(v_x - C\nabla_x p^+) + (B_< - B)(BY_< + B_<Y)(Bp^+ + B_<p_<^+), \tag{4}$$

$$v_y^+ = BB_\vee\left(v_y - C\nabla_y p^+\right) + (B_\vee - B)(BY_\vee + B_\vee Y)(Bp^+ + B_\vee p_\vee^+). \tag{5}$$

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:

$$\nabla_x p^+ = p^+ - p_<^+ \tag{6}$$

$$\nabla_y p^+ = p^+ - p_\vee^+ \tag{7}$$

$$\nabla \cdot \bar{v} = (v_x(x + \Delta x, y) - v_x(x, y)) + (v_y(x, y + \Delta y) - v_y(x, y)) \tag{8}$$

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. Результаты

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.

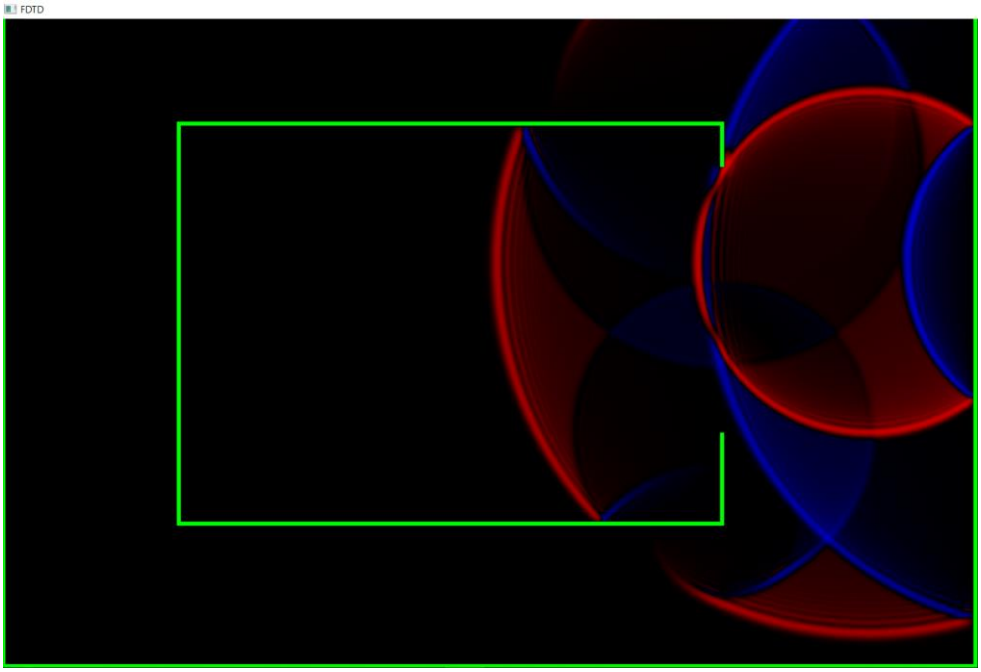

Рис. 1 Визуализация работы алгоритма №1. Клавиши + и − управляют яркостью, а клавиши w,a,s,d меняют положение источника.

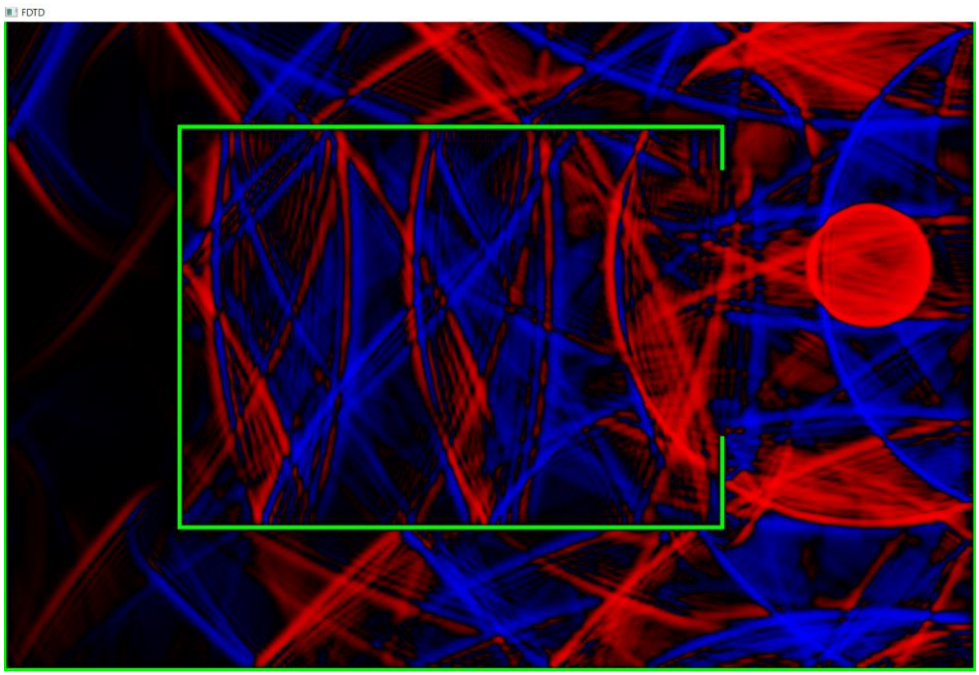

Рис. 2 Визуализация работы алгоритма №1

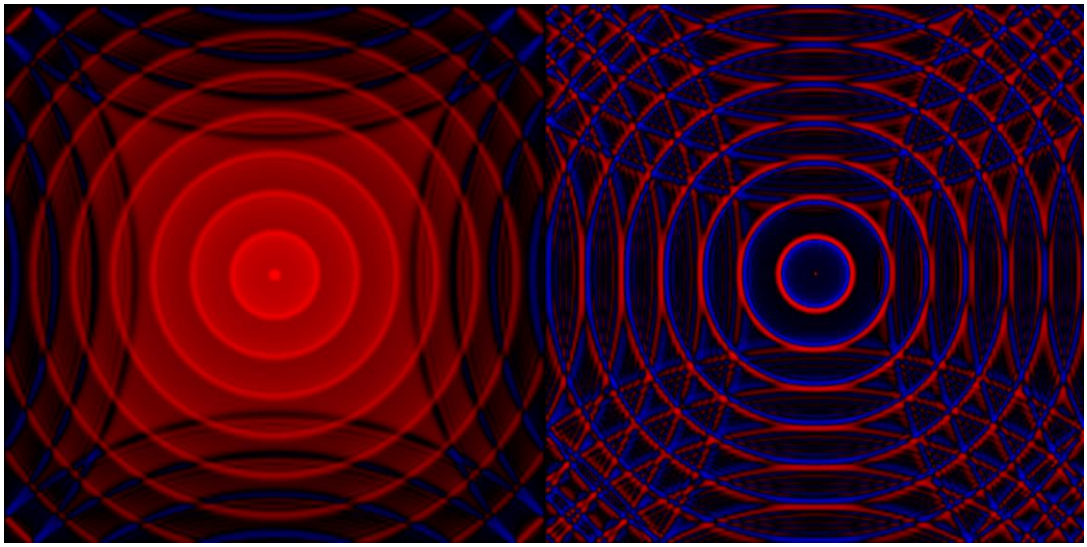

Рис. 3 Слева - алгоритм №1, справа - алгоритм №2

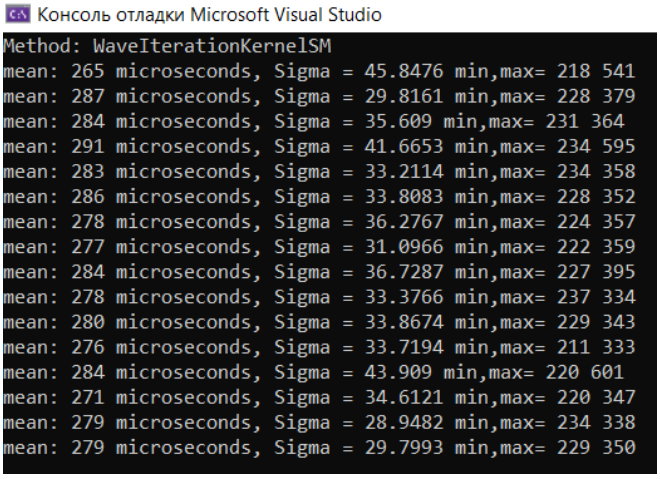

Рис. 4 Вывод программы в консоль в режиме визуализации

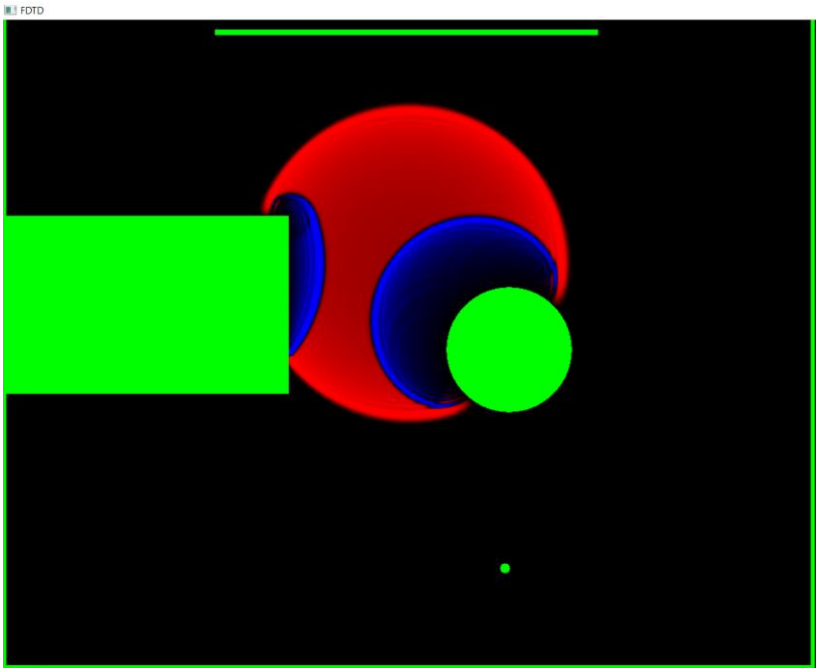

Рис. 5 Визуализация работы алгоритма №1

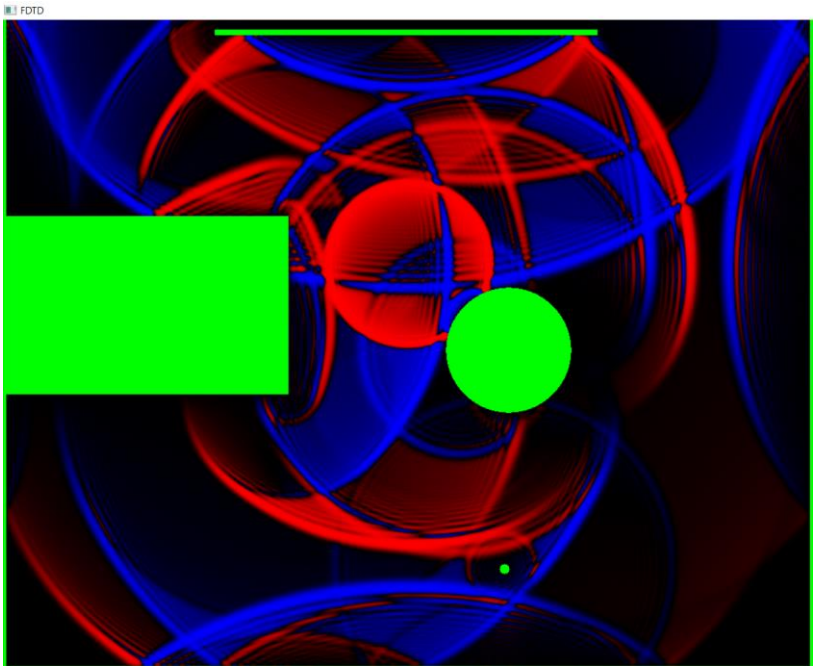

Рис. 6 Визуализация работы алгоритма №1

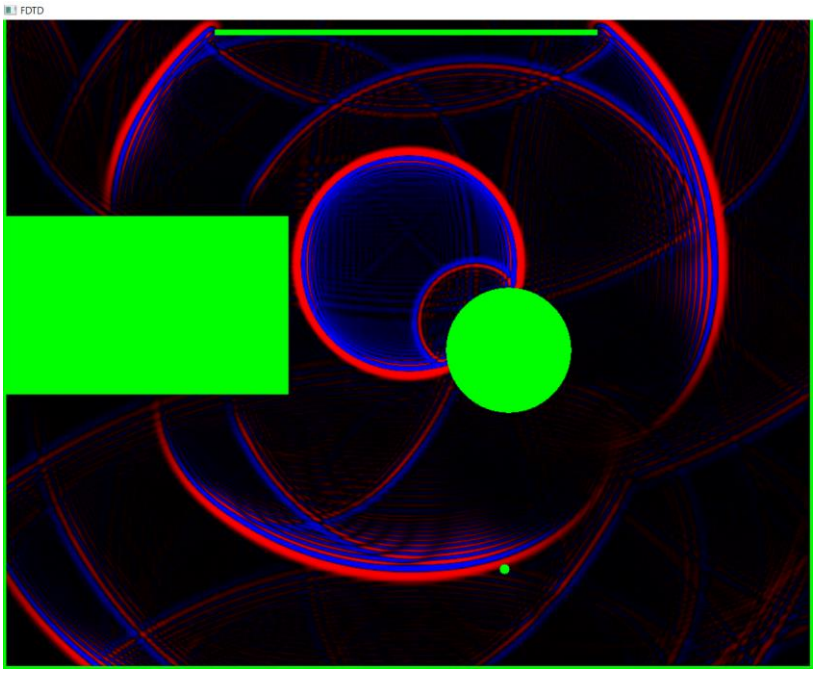

Рис. 7 Визуализация работы алгоритма №2. Малый коэффициент отражения поверхности.

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.

Результаты для программы, основанной на типе данных float:

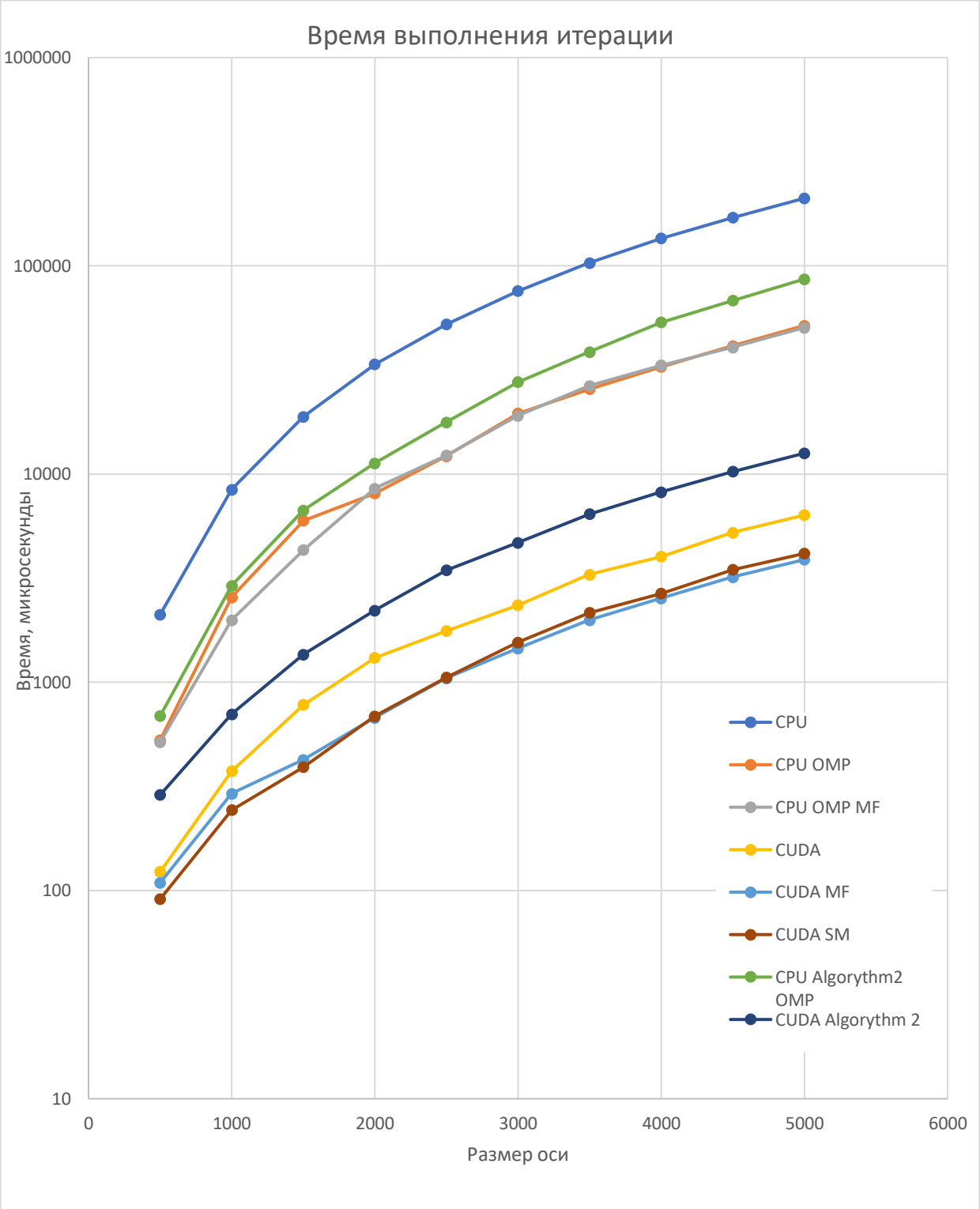

Рис. 8 Время выполнения одной итерации алгоритма в микросекундах в зависимости от размера сетки, логарифмический масштаб по вертикальной оси.

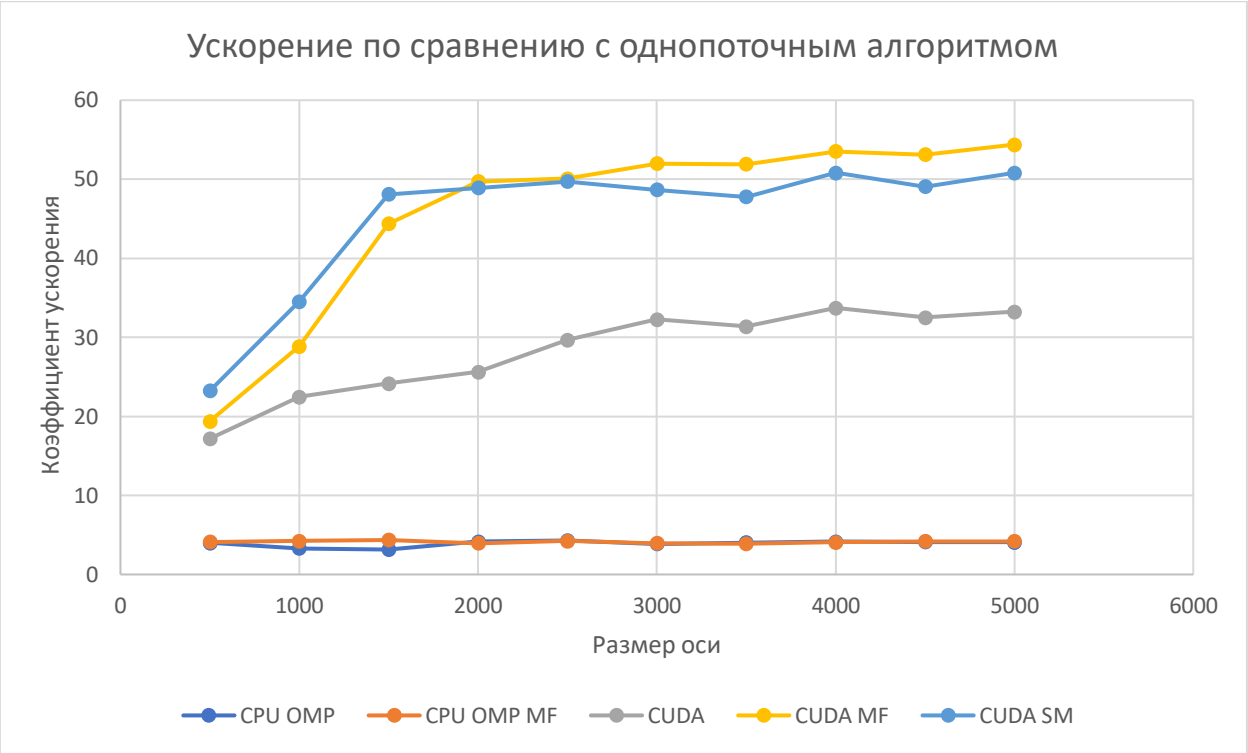

Рис. 9 Сравнение с однопоточным алгоритмом

По Рис. 9 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.

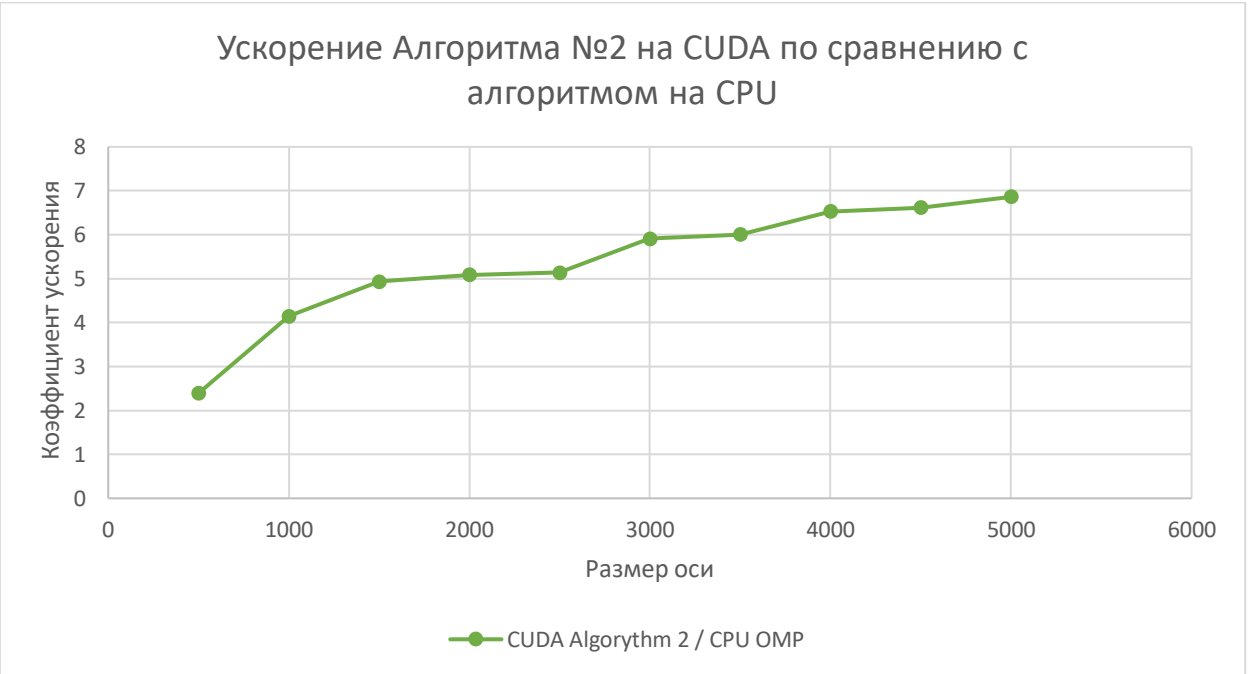

Рис. 10 Сравнение GPU алгоритма с CPU версией для алгоритма №2

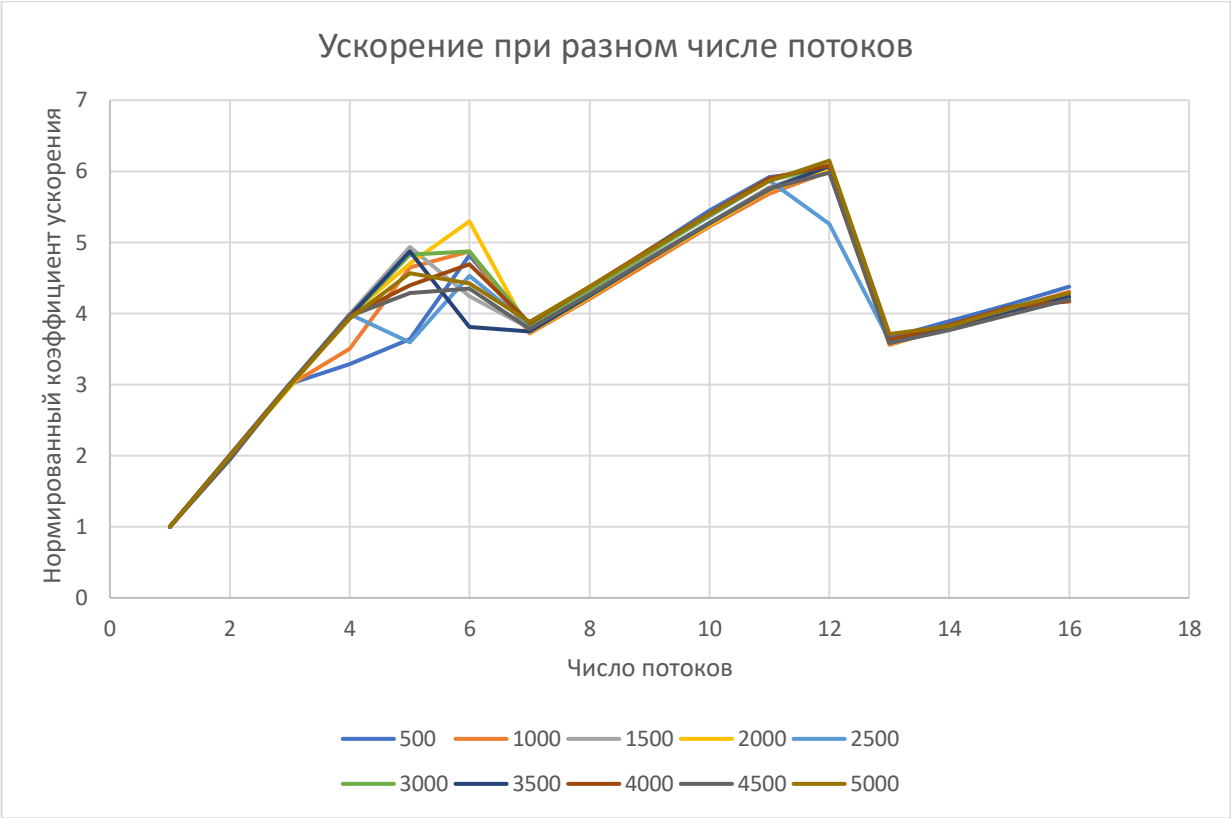

Рис. 11 Зависимость ускорения CPU алгоритма от числа потоков. На моем центральном процессоре 6 ядер и 12 логических процессоров.

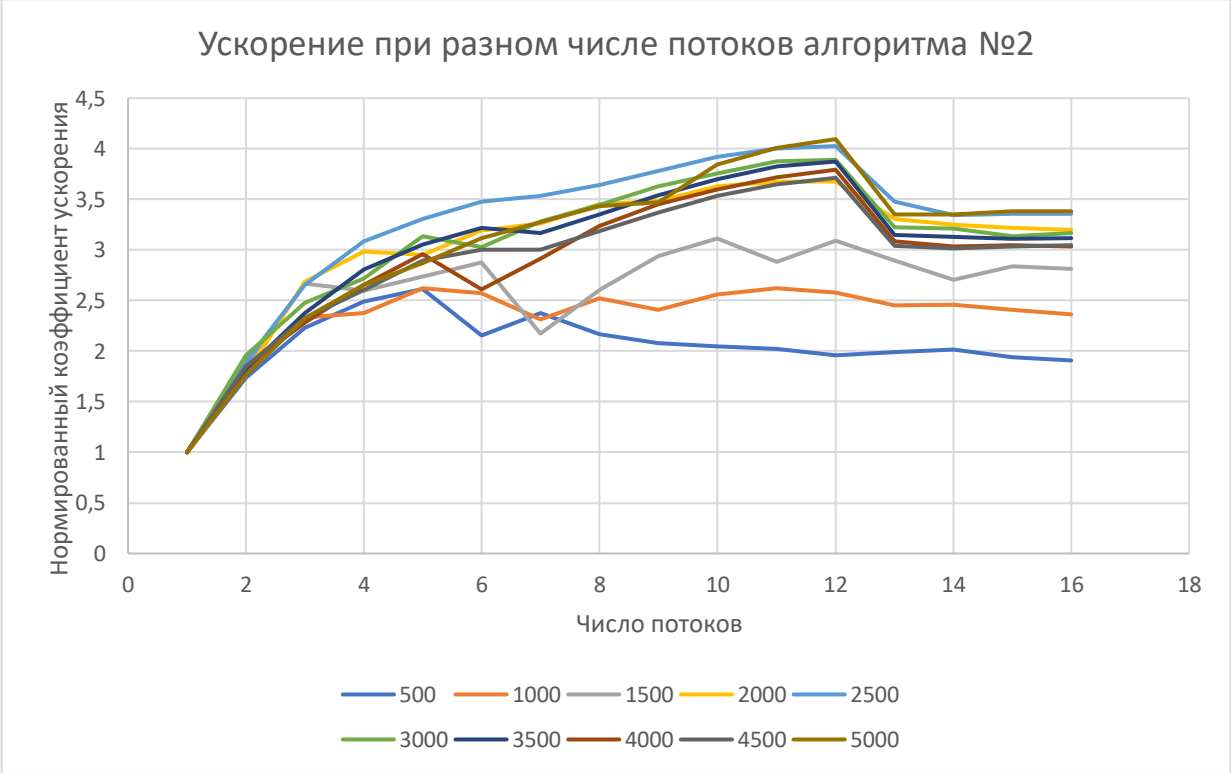

Рис. 12 Ускорение для алгоритма №2. Похожая зависимость сохраняется, но, вероятно, способ распараллеливания алгоритма не является оптимальным, из-за чего максимальное ускорение равно 4, а не 6.

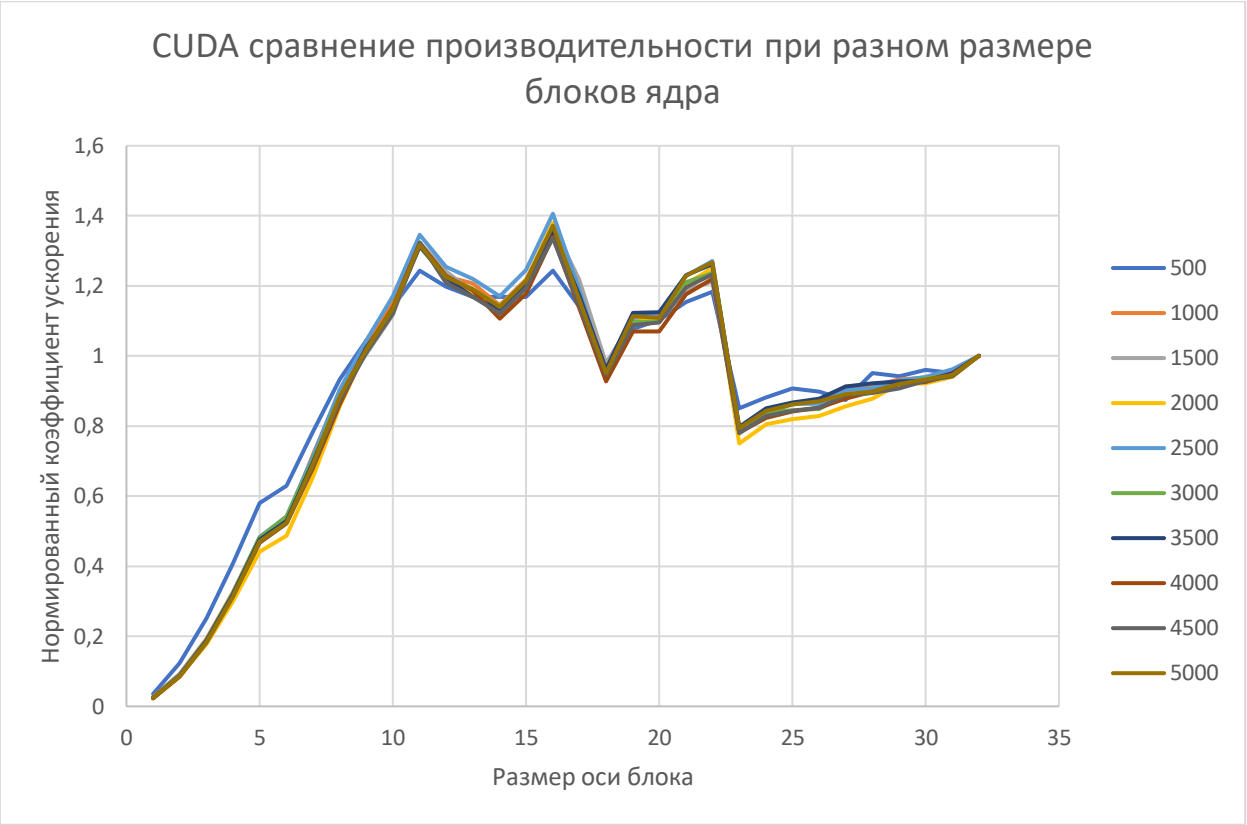

Рис. 13 Графики на отдельных размерах сетки нормированы относительно времени с размером блока, равного 32. Видно, что в случае выбора блока с размером в 16, производительность повышается в 1.36 раз.

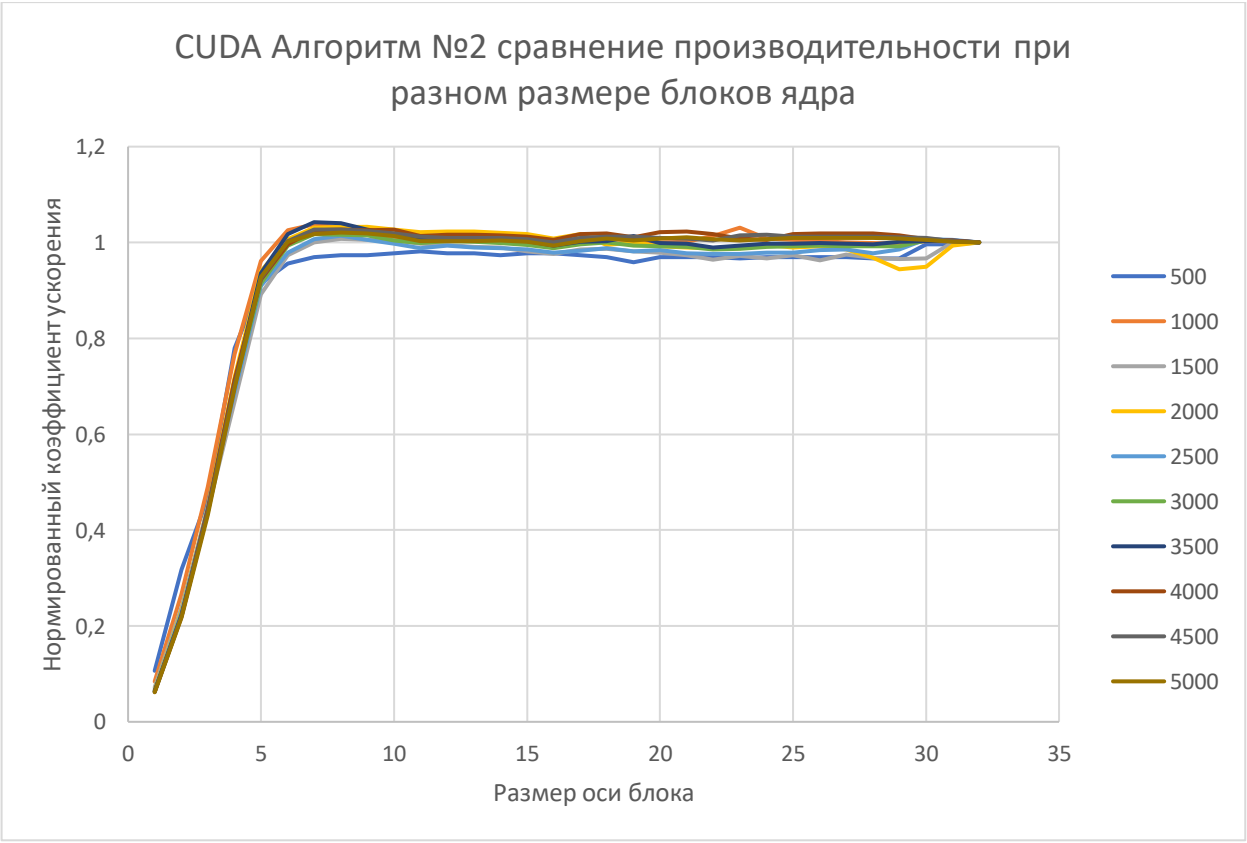

Рис. 14 Для алгоритма №2 размер блока имеет меньшее влияние на производительность

Результаты для типа данных double:

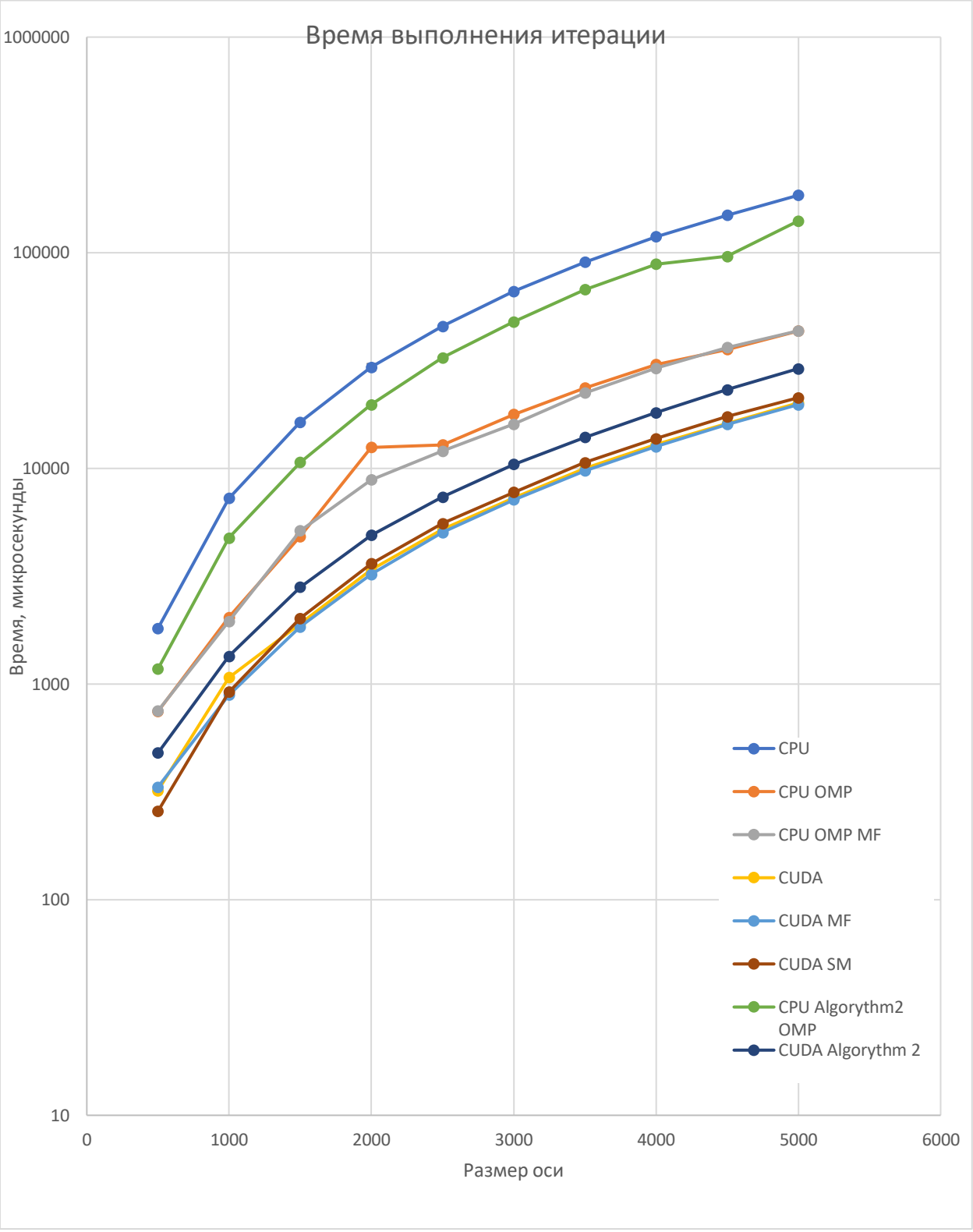

Рис. 15 Время выполнения одной итерации алгоритма в микросекундах в зависимости от размера сетки, логарифмический масштаб по вертикальной оси

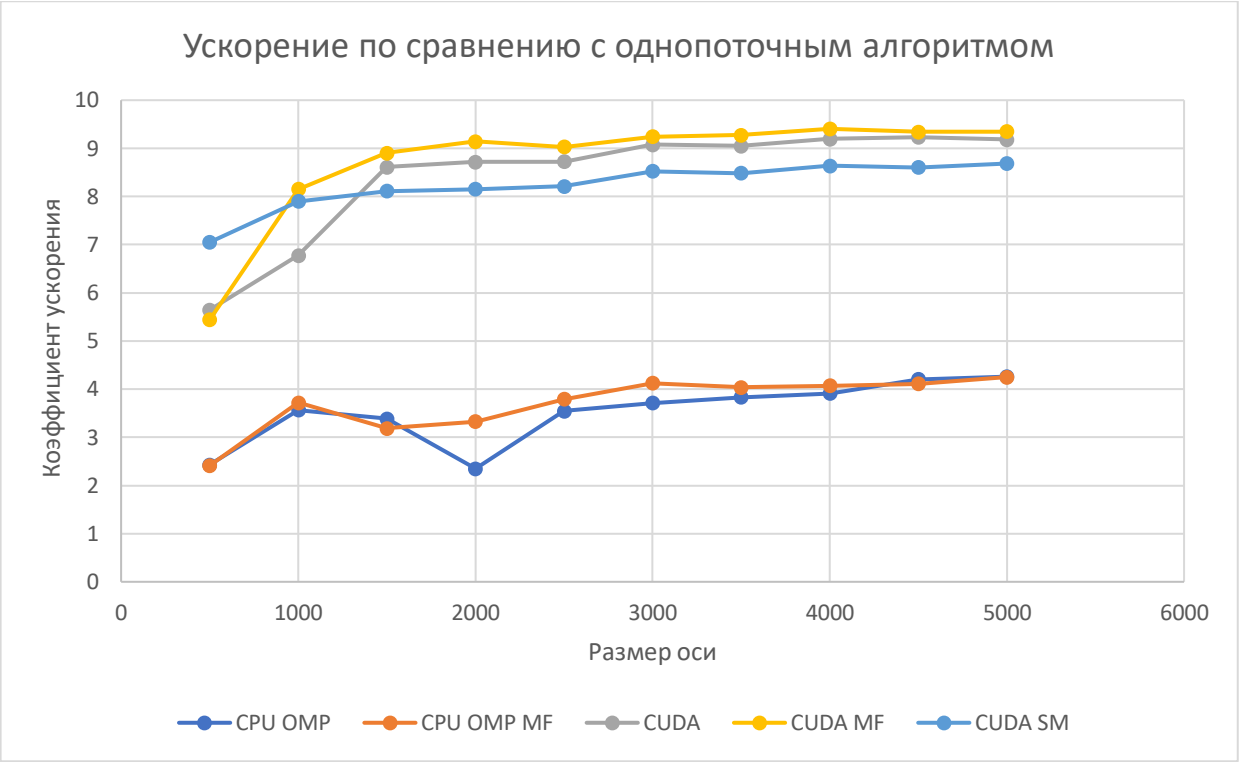

Рис. 16 Сравнение с однопоточным алгоритмом

Сравнивая график на Рис. 16 с графиком на Рис.9, 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.

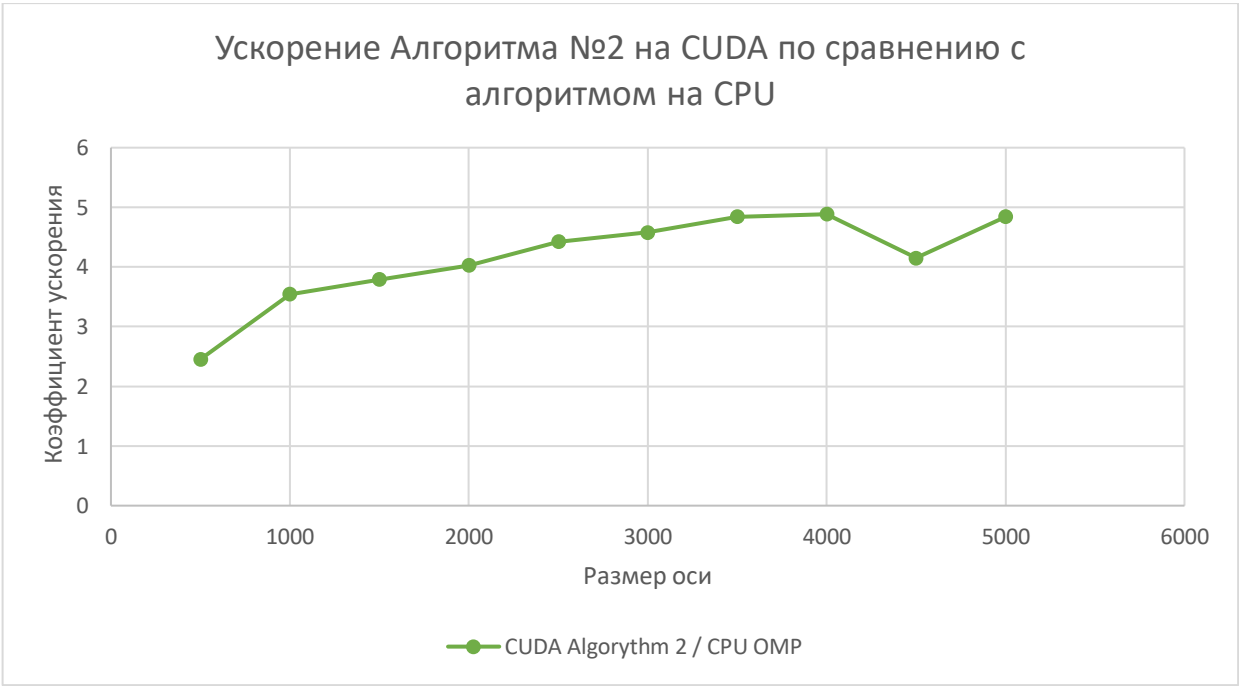

Рис. 17 Сравнение GPU алгоритма с CPU версией для алгоритма №2

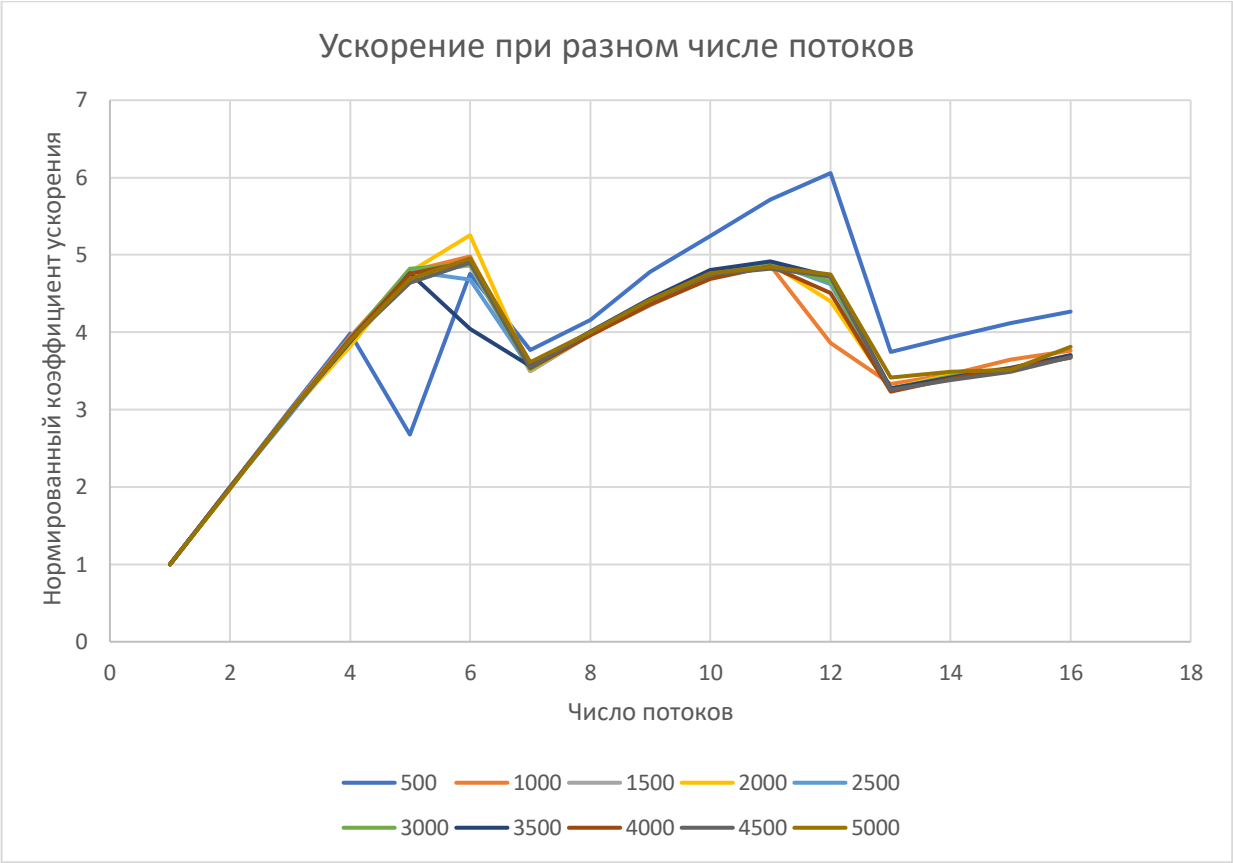

Рис. 18 Зависимость ускорения CPU алгоритма от числа потоков

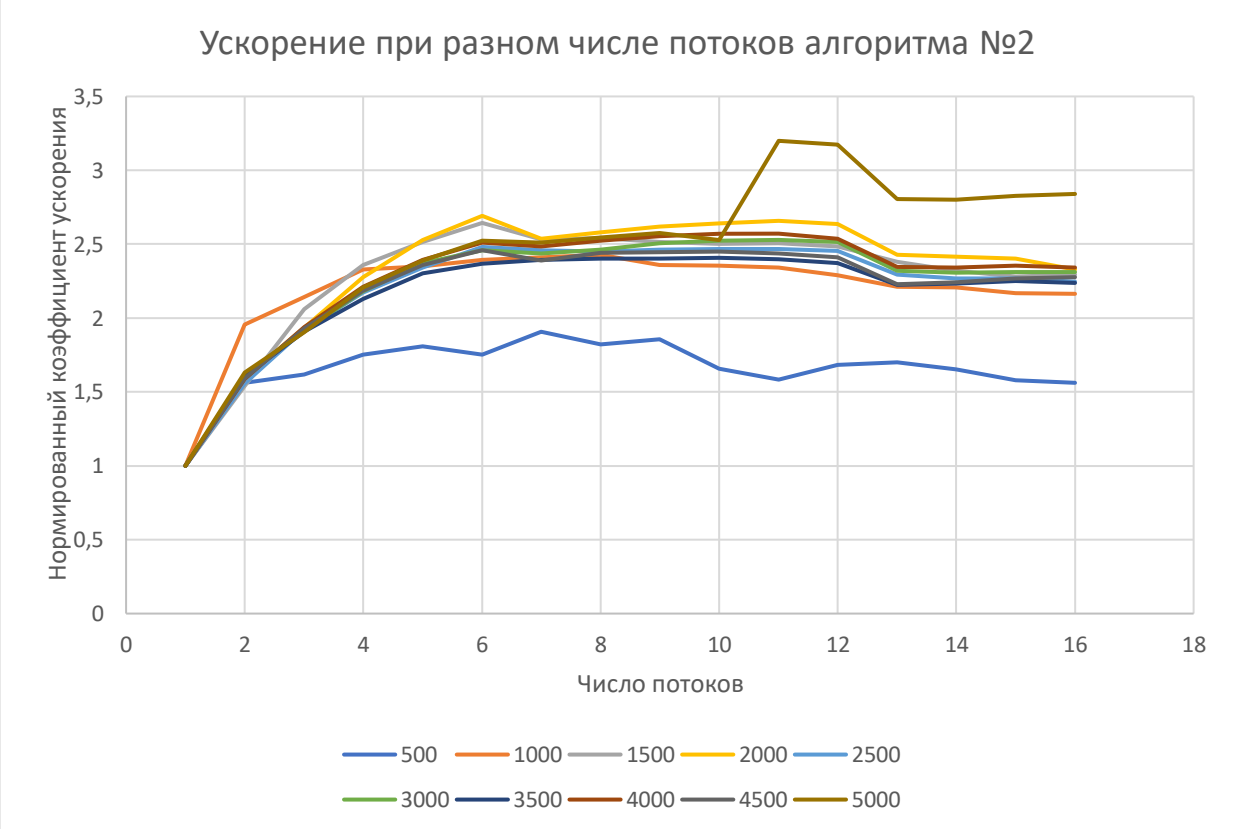

Рис. 19 Зависимость ускорения CPU алгоритма от числа потоков для алгоритма №2

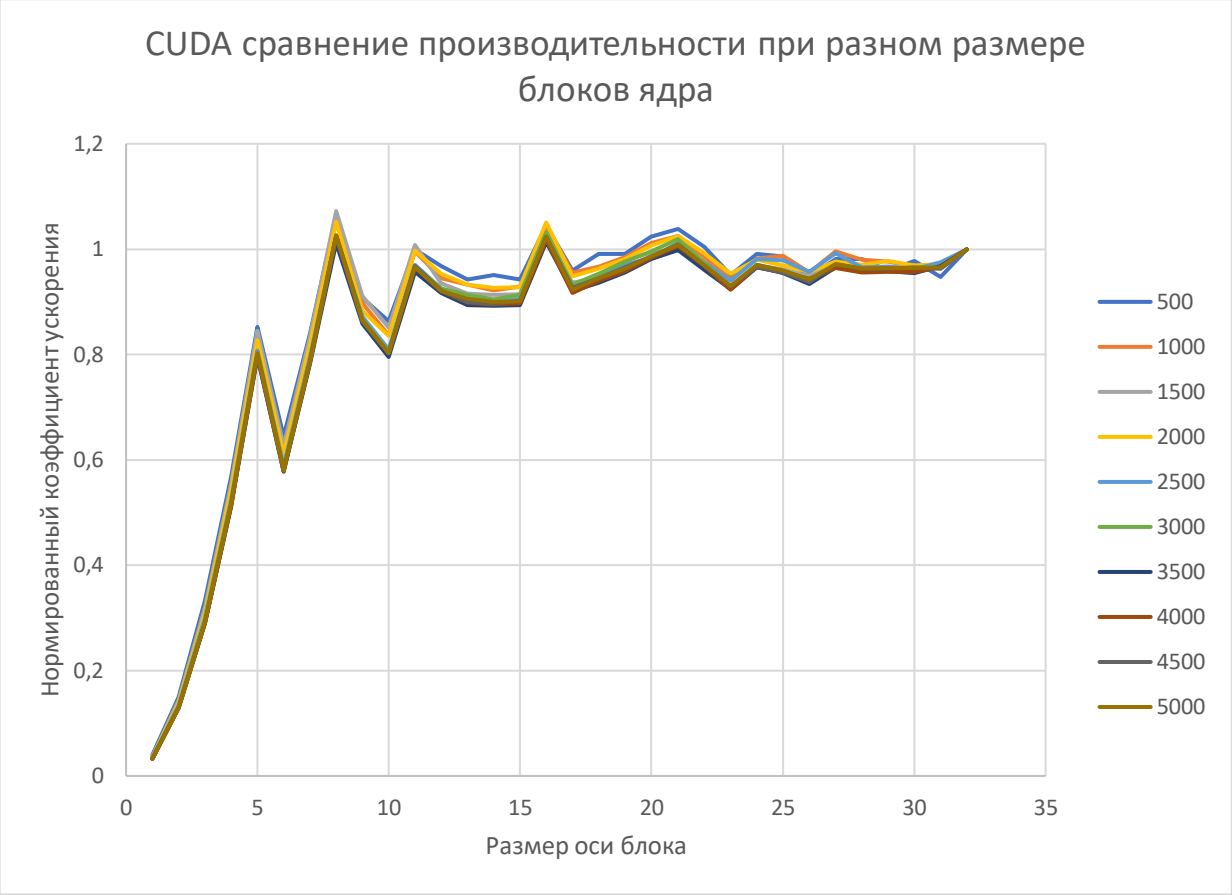

Рис. 20 Зависимость от размера блока ядра.

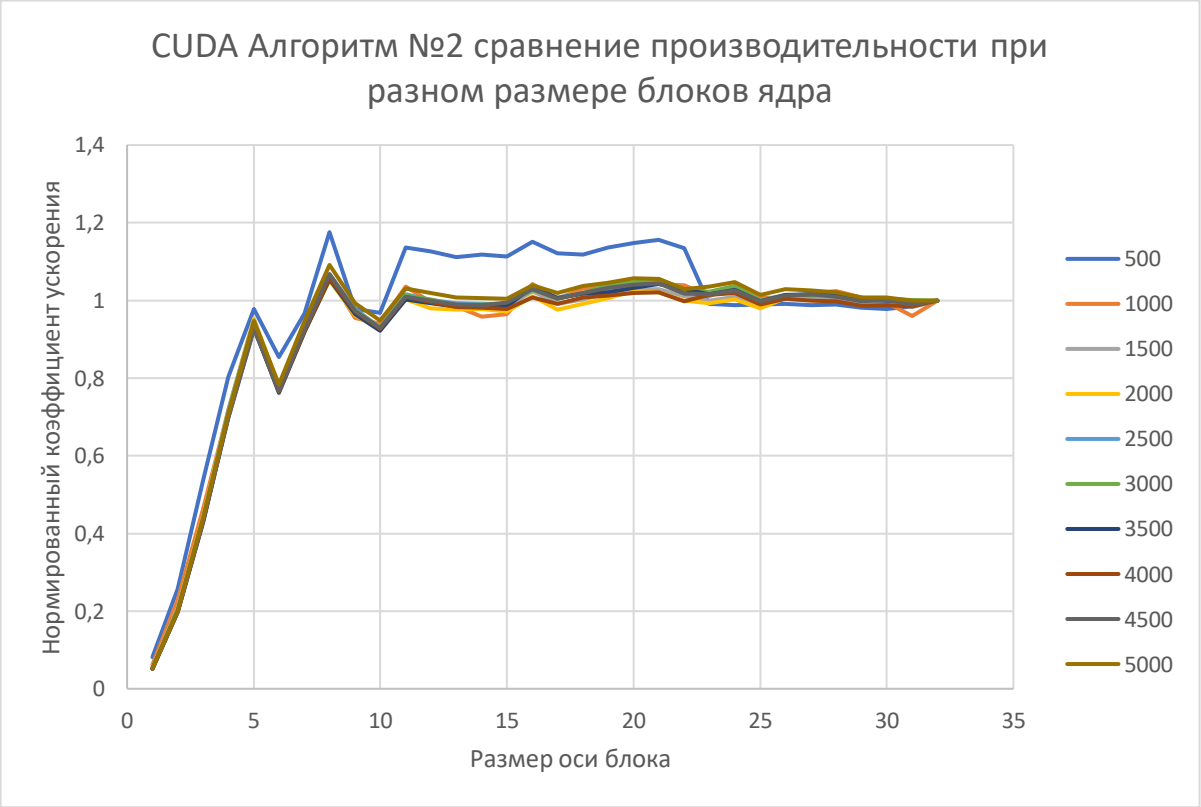

Рис. 21 Зависимость от размера блока ядра. Для типа данных double изменение размера с 32 не дает существенного прироста производительности

Рассмотрим во сколько раз производительность методов на типе данных float выше, чем на типе данных double. 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.

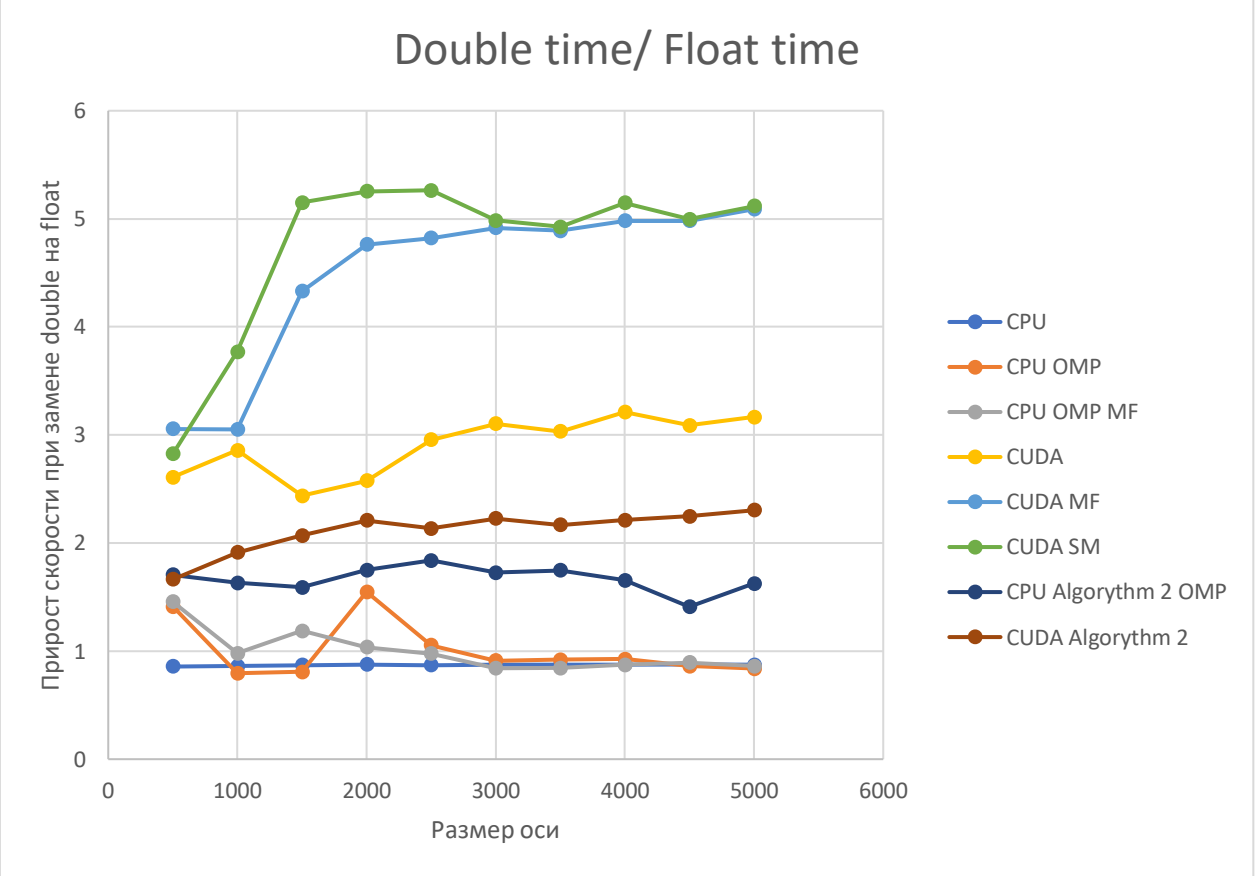

Рис. 22 Сравнения float и double версий алгоритмов

Далее рассмотрим результаты, полученные на вычислительном платформе от СПБГУ, на кластере GPUlab с характеристиками: центральным процессором AMD FX-8370 4.0 GHz, видеокартой GeForce GTX 1060. Количество OMP потоков выбрано равным 8.

Результаты для типа данных float:

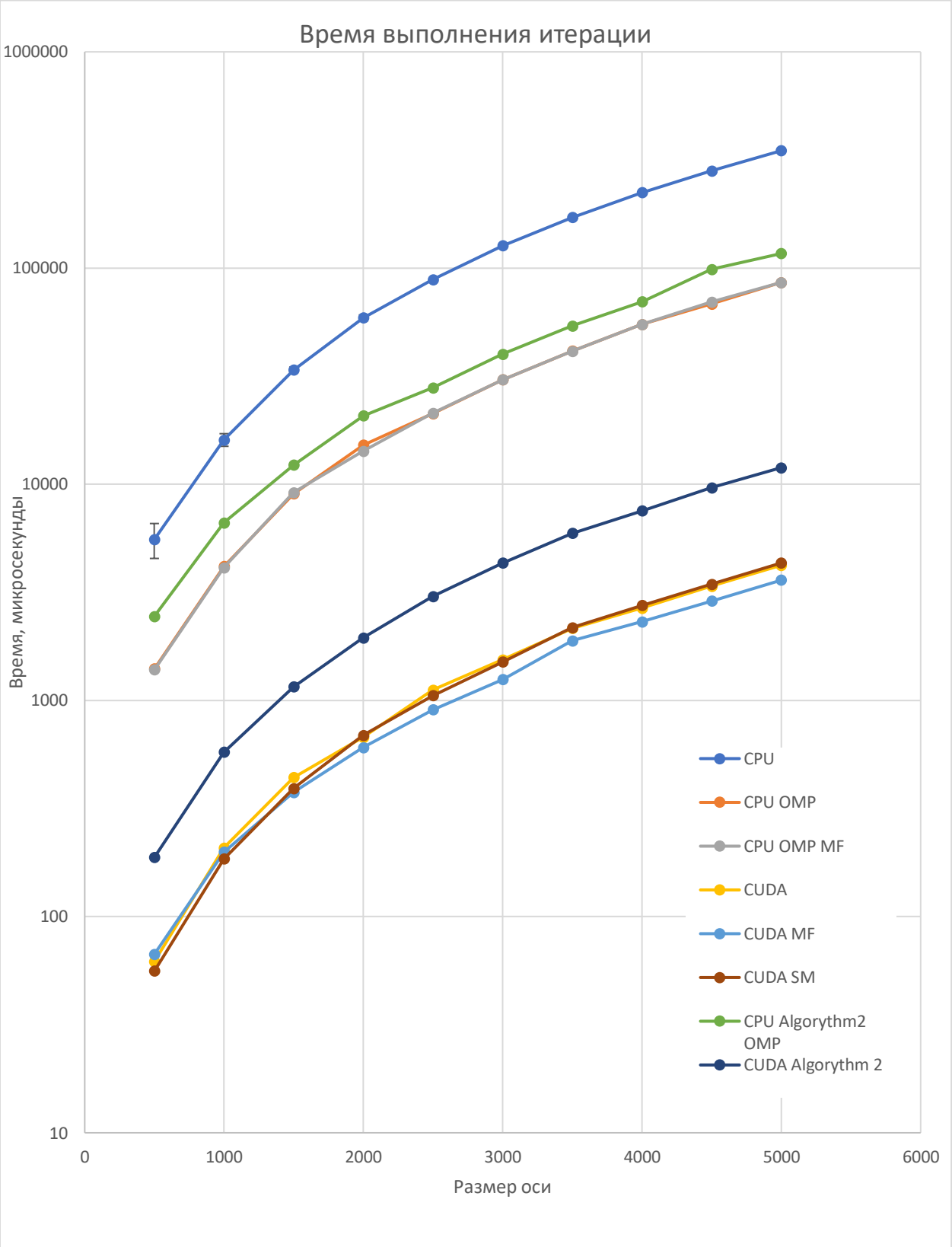

Рис. 23 Время выполнения одной итерации алгоритма в микросекундах в зависимости от размера сетки, логарифмический масштаб по вертикальной оси

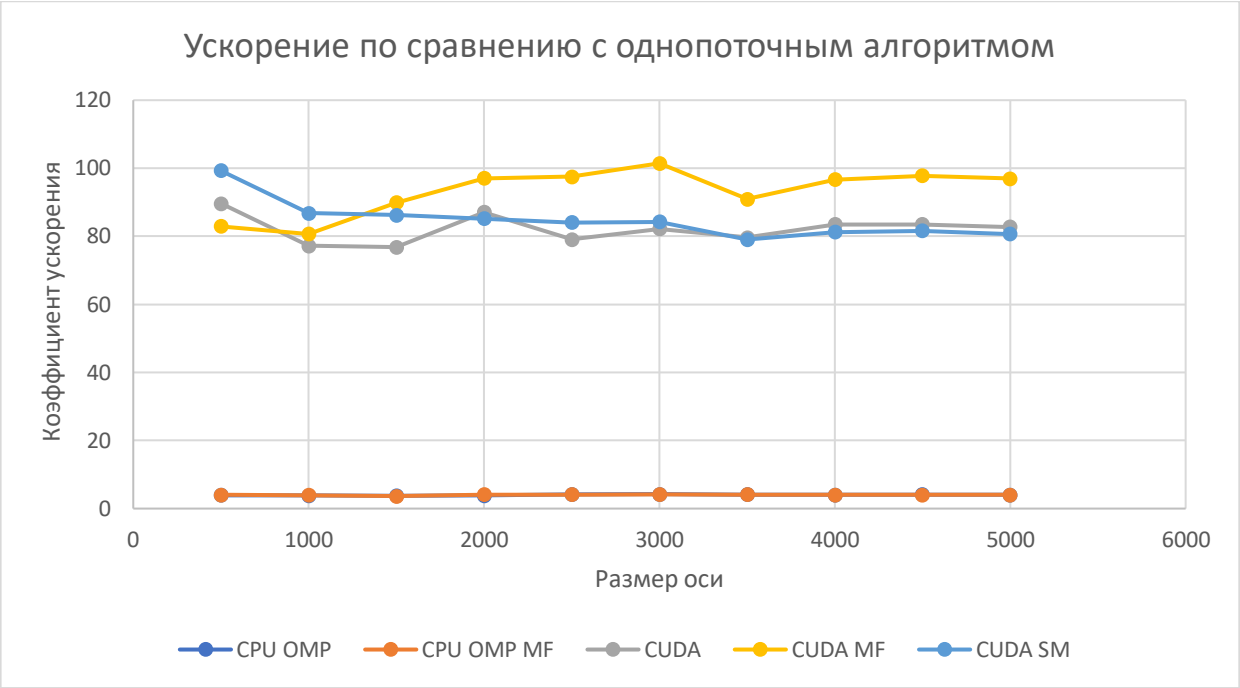

Рис. 24 Сравнение с однопоточным алгоритмом

По Рис. 24 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.

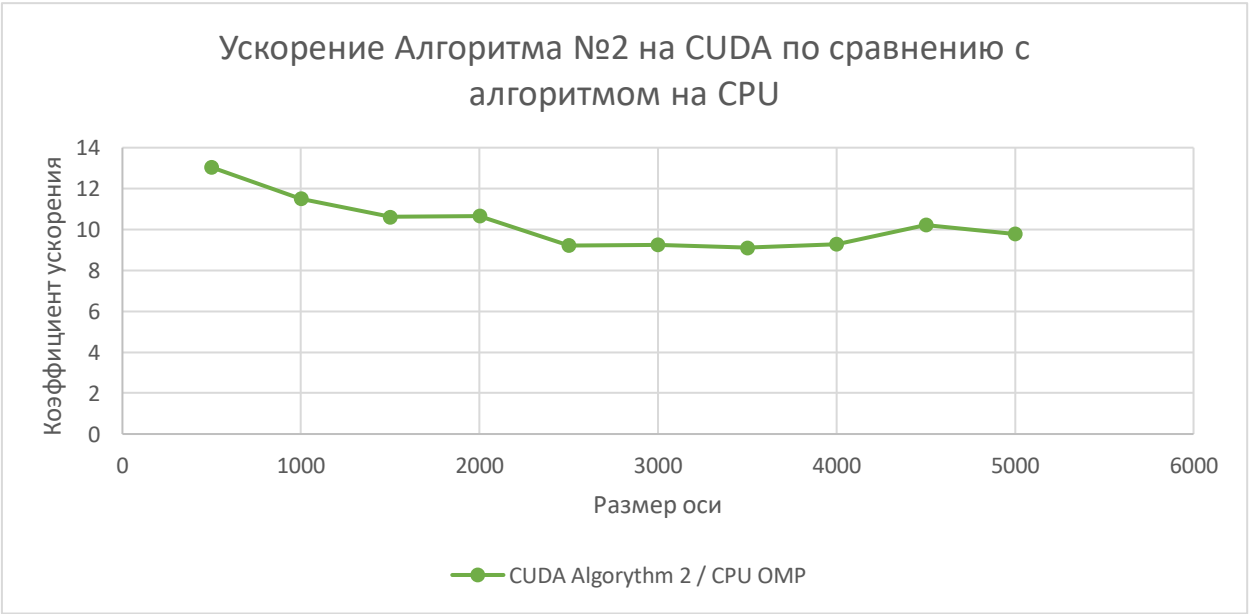

Рис. 25 Сравнение GPU алгоритма с CPU версией для алгоритма №2

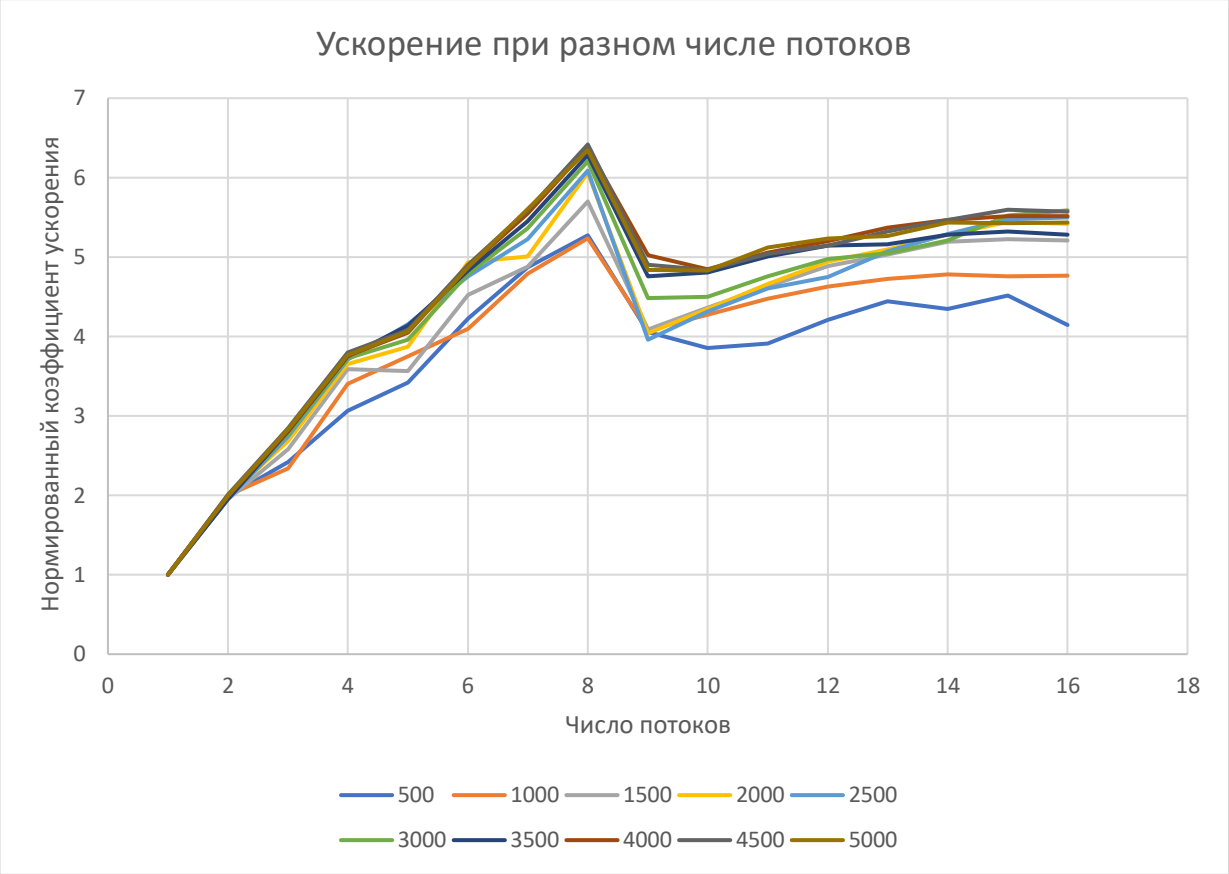

Рис. 26 Зависимость ускорения CPU алгоритма от числа потоков. По графику видно, что CPU обладает 8 логическими потоками.

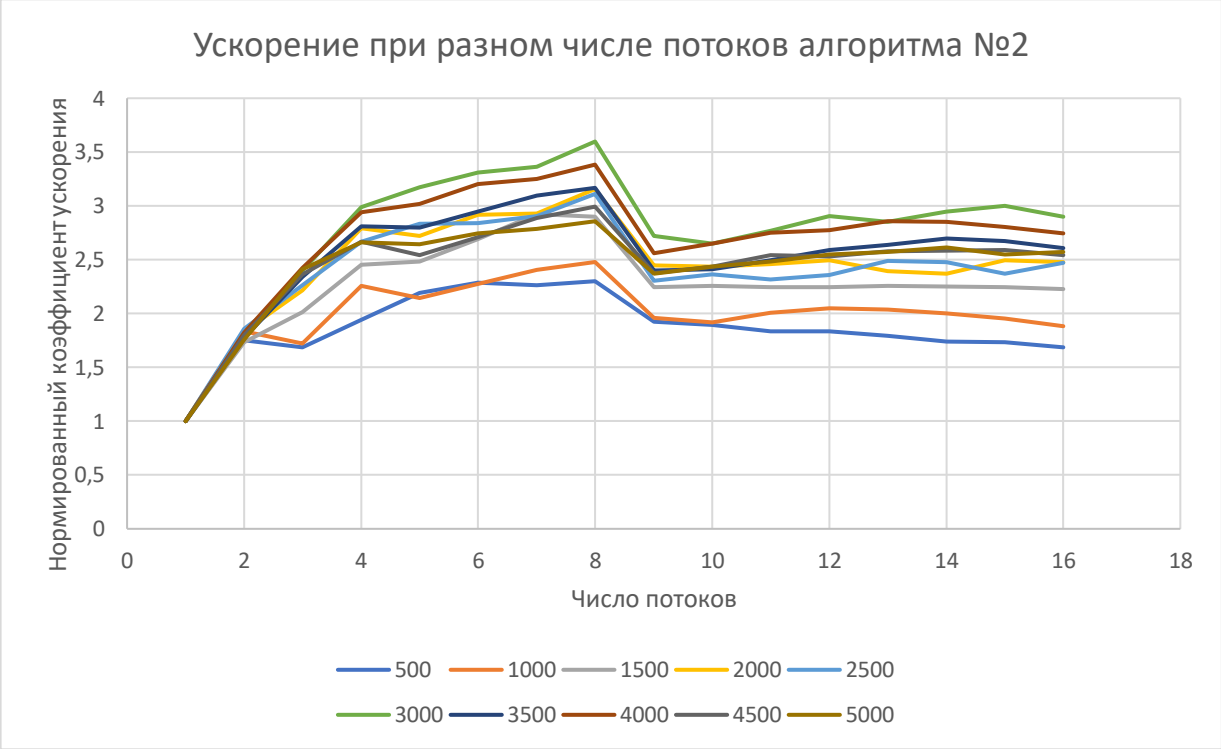

Рис. 27 Зависимость ускорения CPU версии метода от числа потоков для алгоритма №2

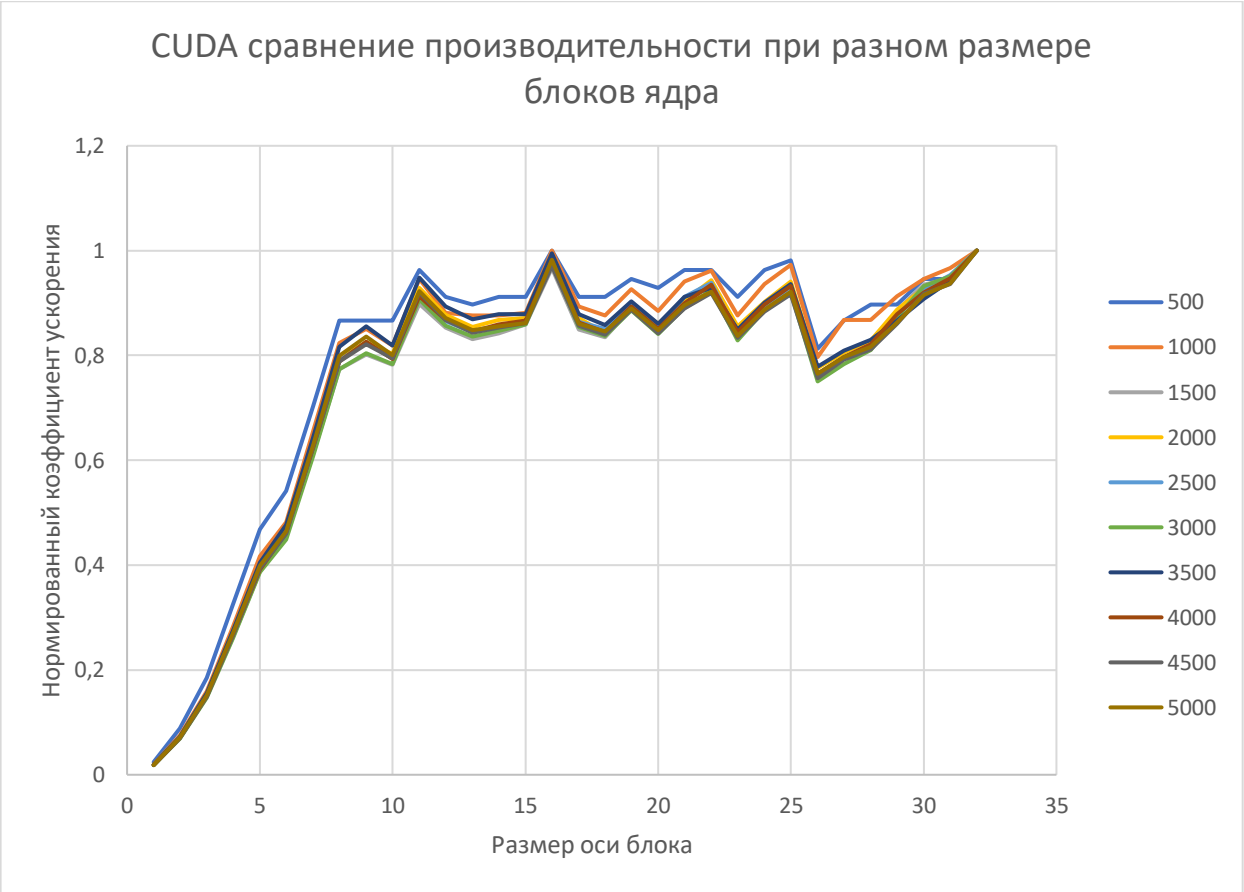

Рис. 28 Зависимость от размера блока ядра.

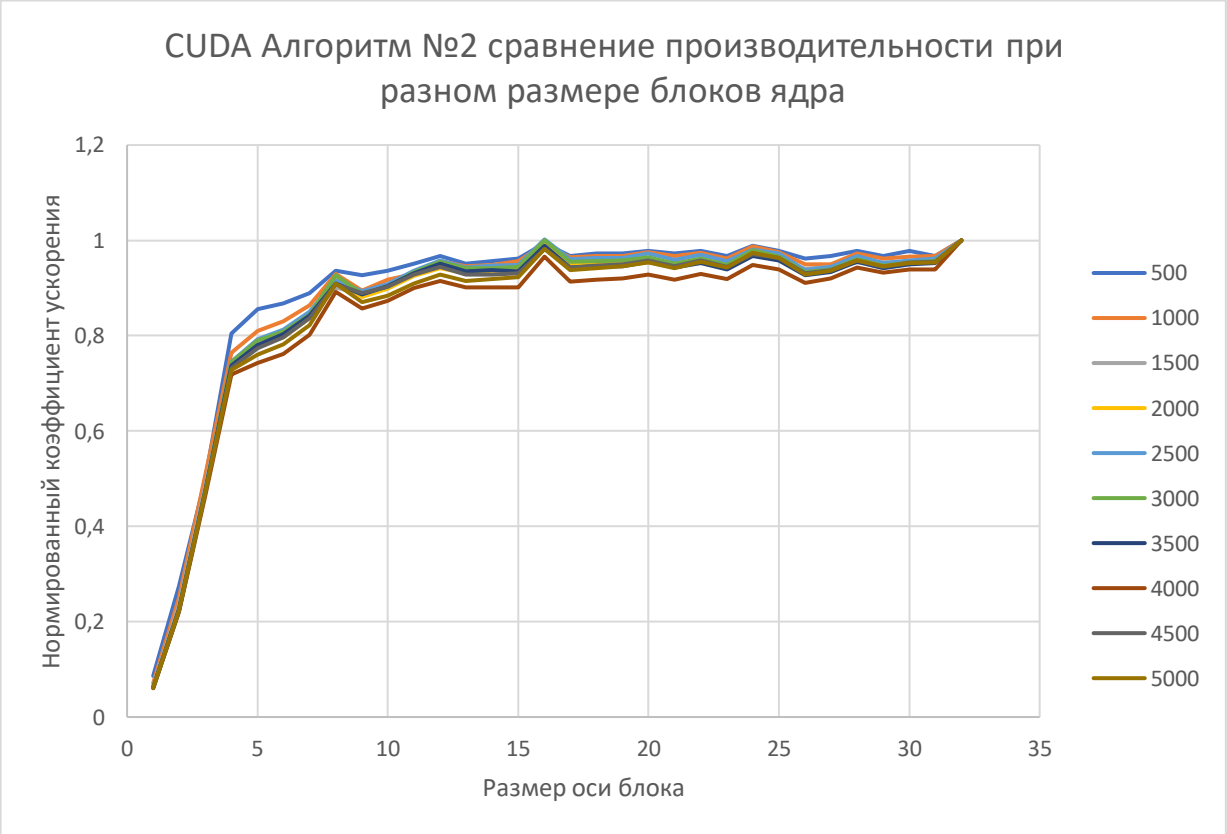

Рис. 29 Зависимость от размера блока ядра. Размер блока 32 дает наилучший результат.

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

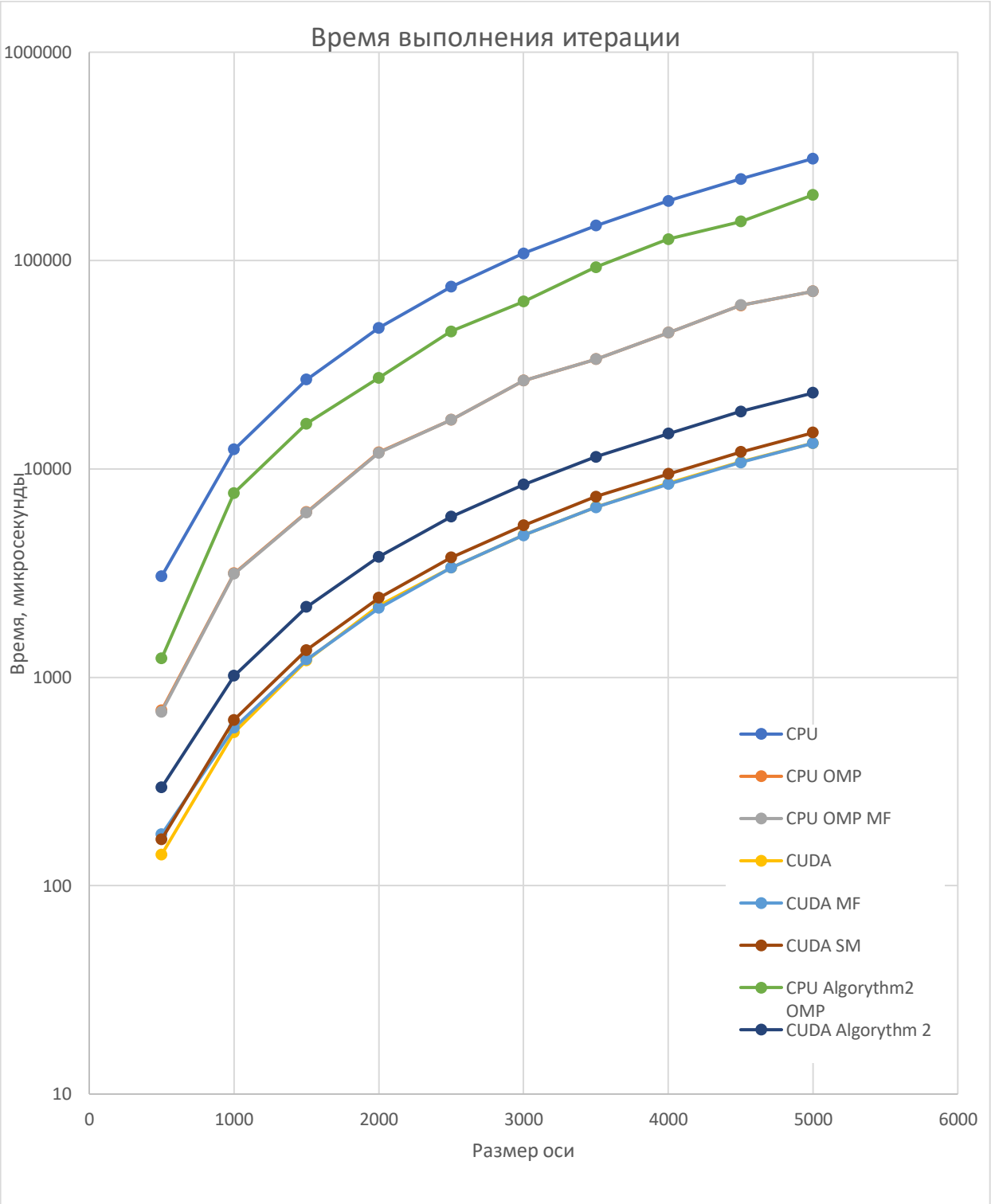

Рис. 30 Время выполнения одной итерации алгоритма в микросекундах в зависимости от размера сетки, логарифмический масштаб по вертикальной оси

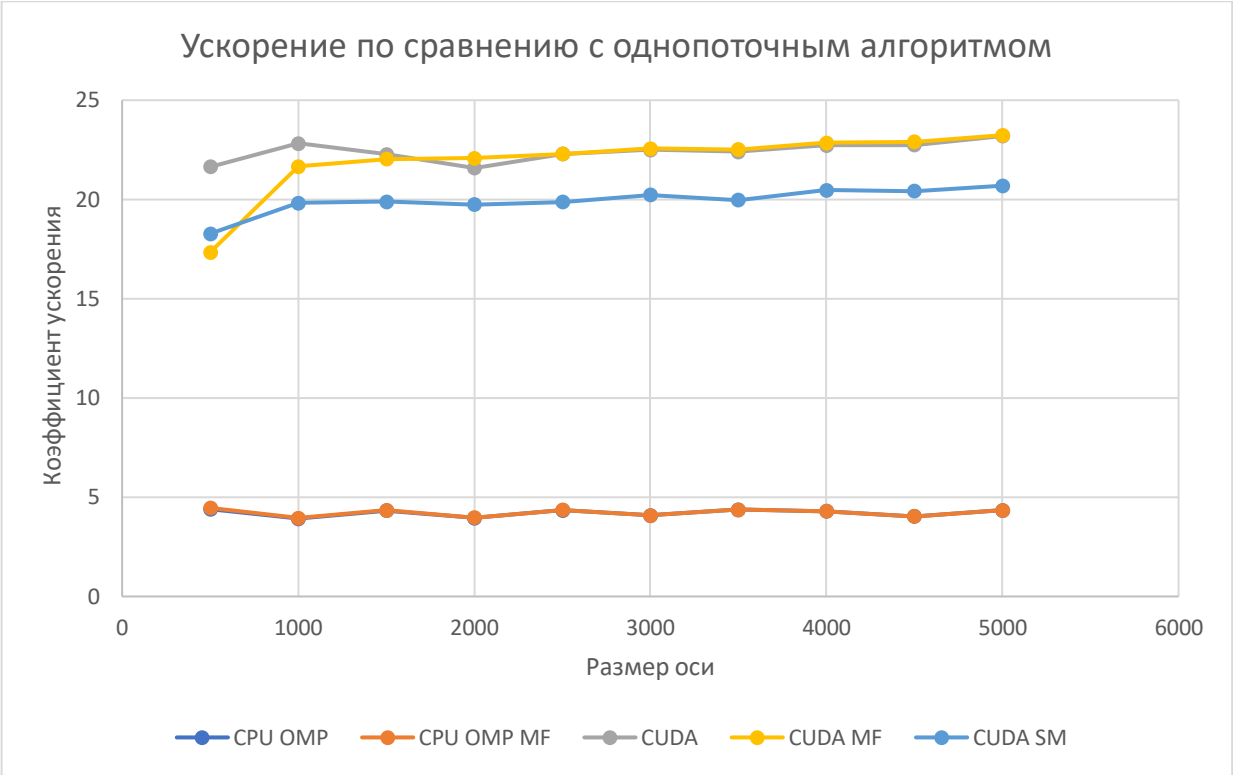

Рис. 31 Сравнение с однопоточным алгоритмом

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

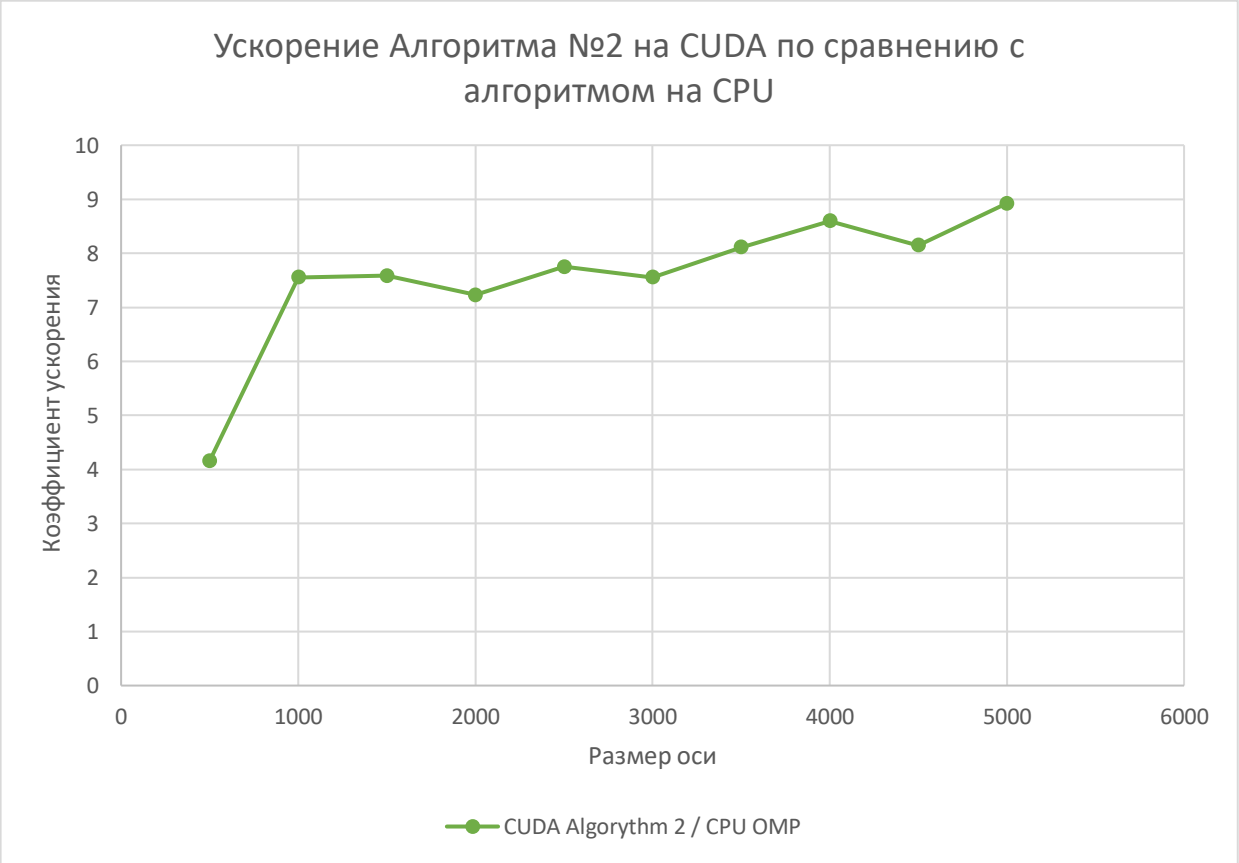

Рис. 32 Сравнение GPU алгоритма с CPU версией для алгоритма №2

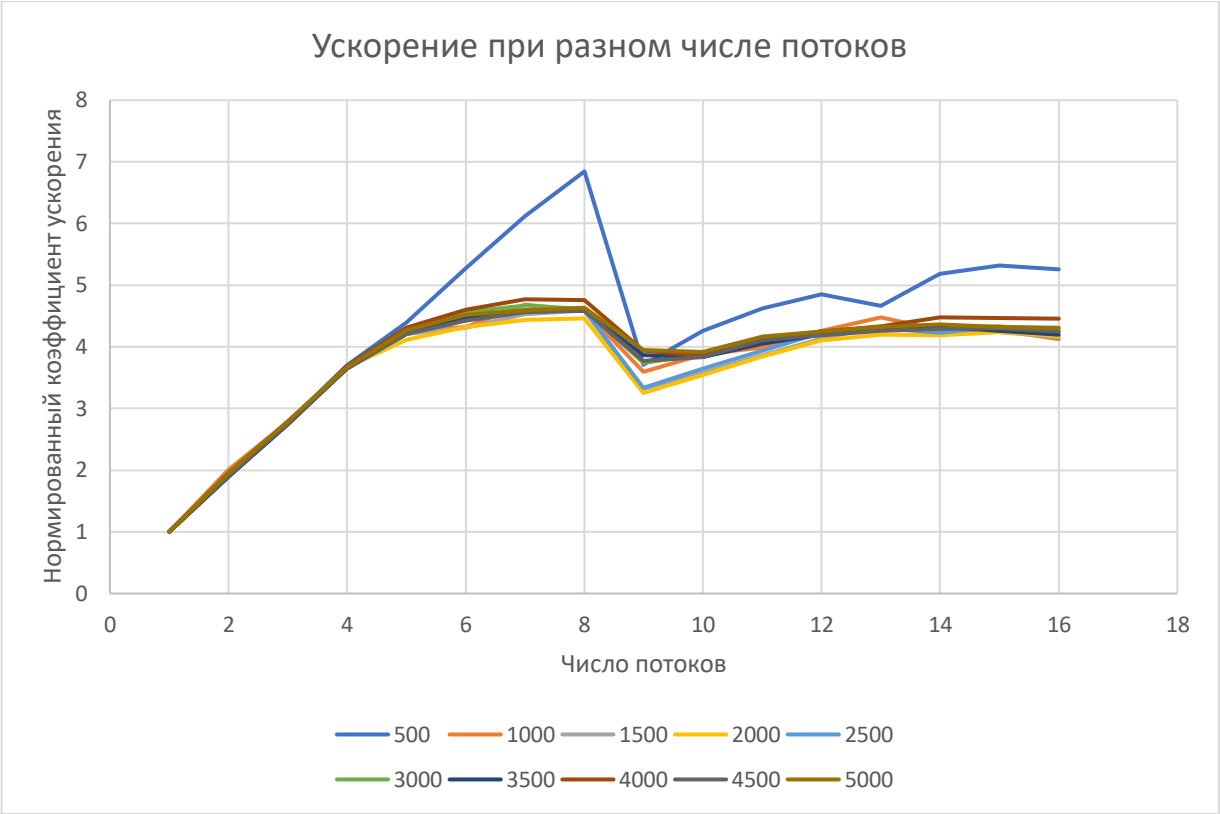

Рис. 33 Зависимость ускорения CPU алгоритма от числа потоков

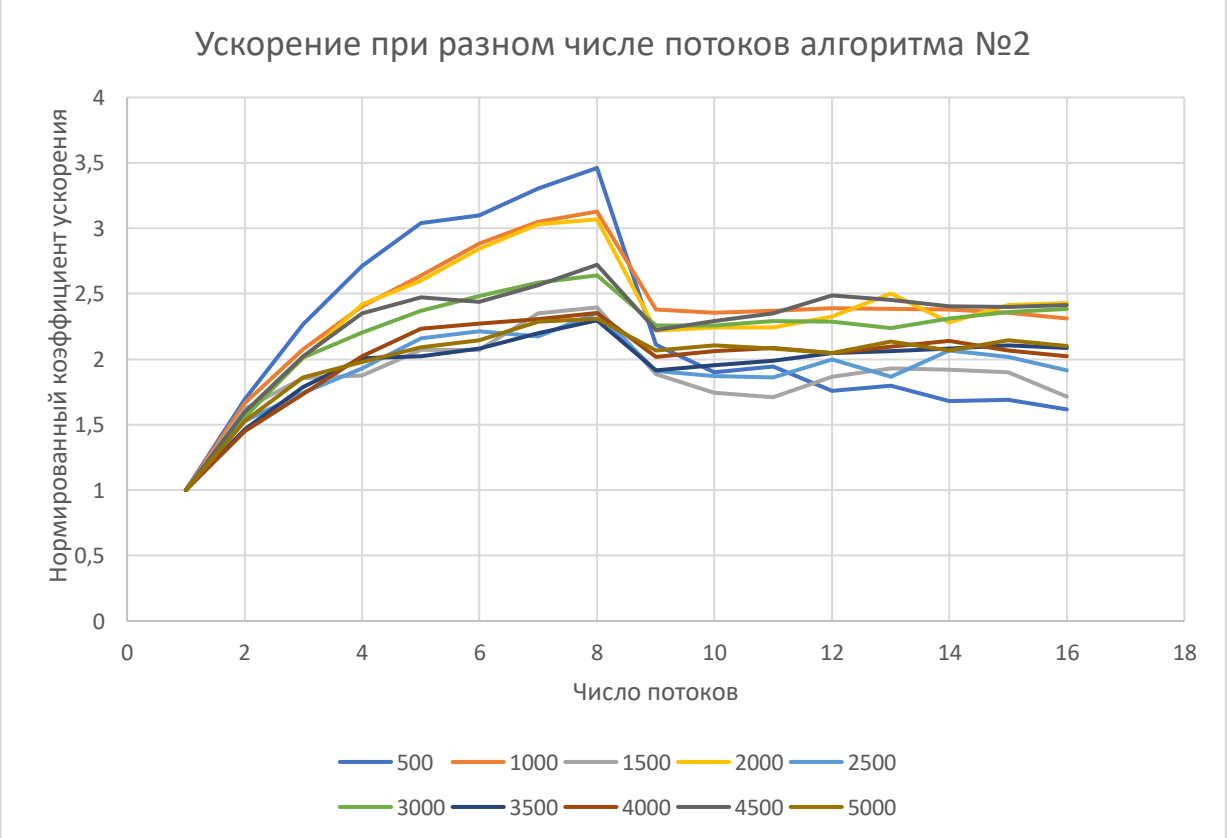

Рис. 34 Зависимость ускорения CPU алгоритма от числа потоков для алгоритма №2

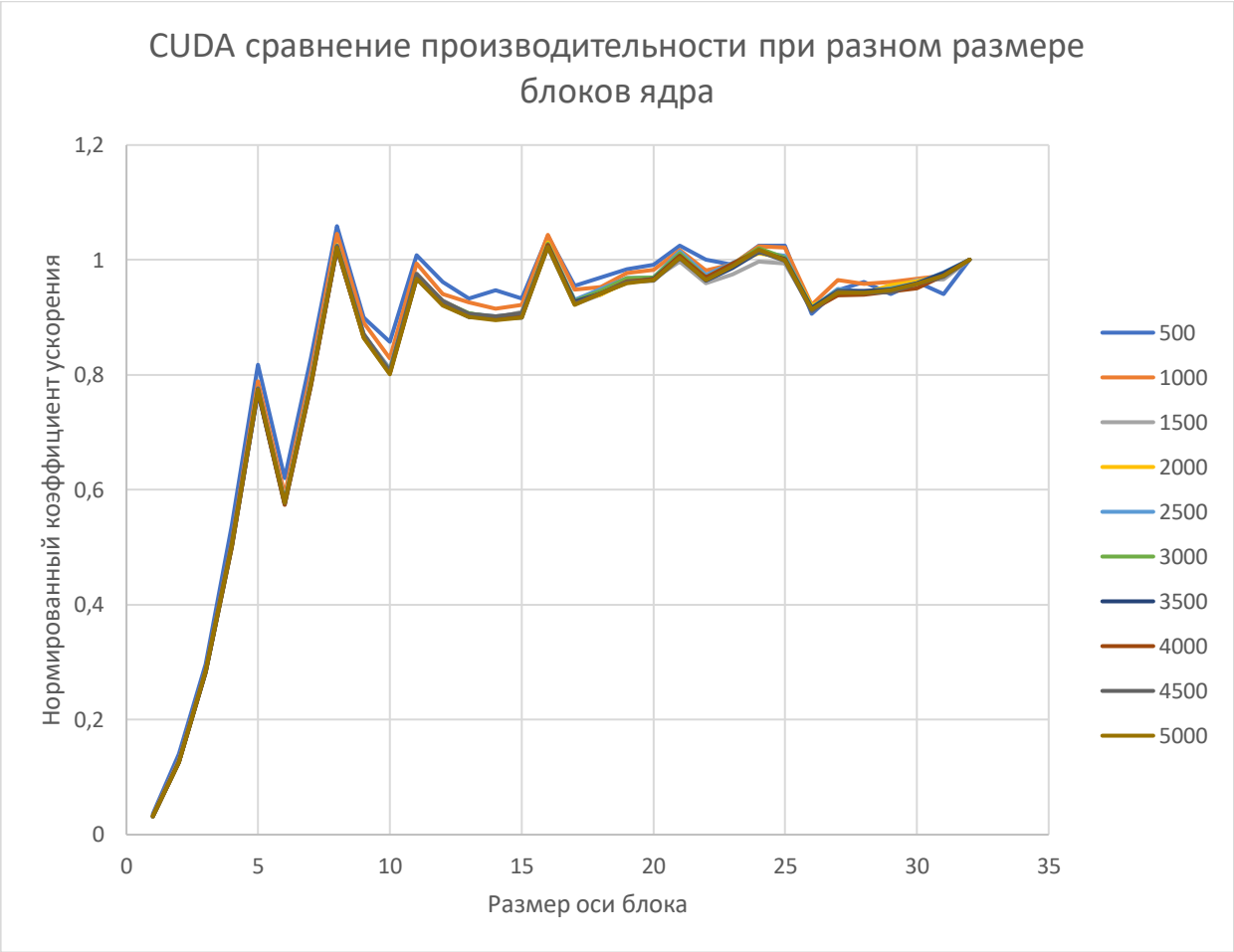

Рис. 35 Зависимость производительности GPU алгоритма от размеров блока

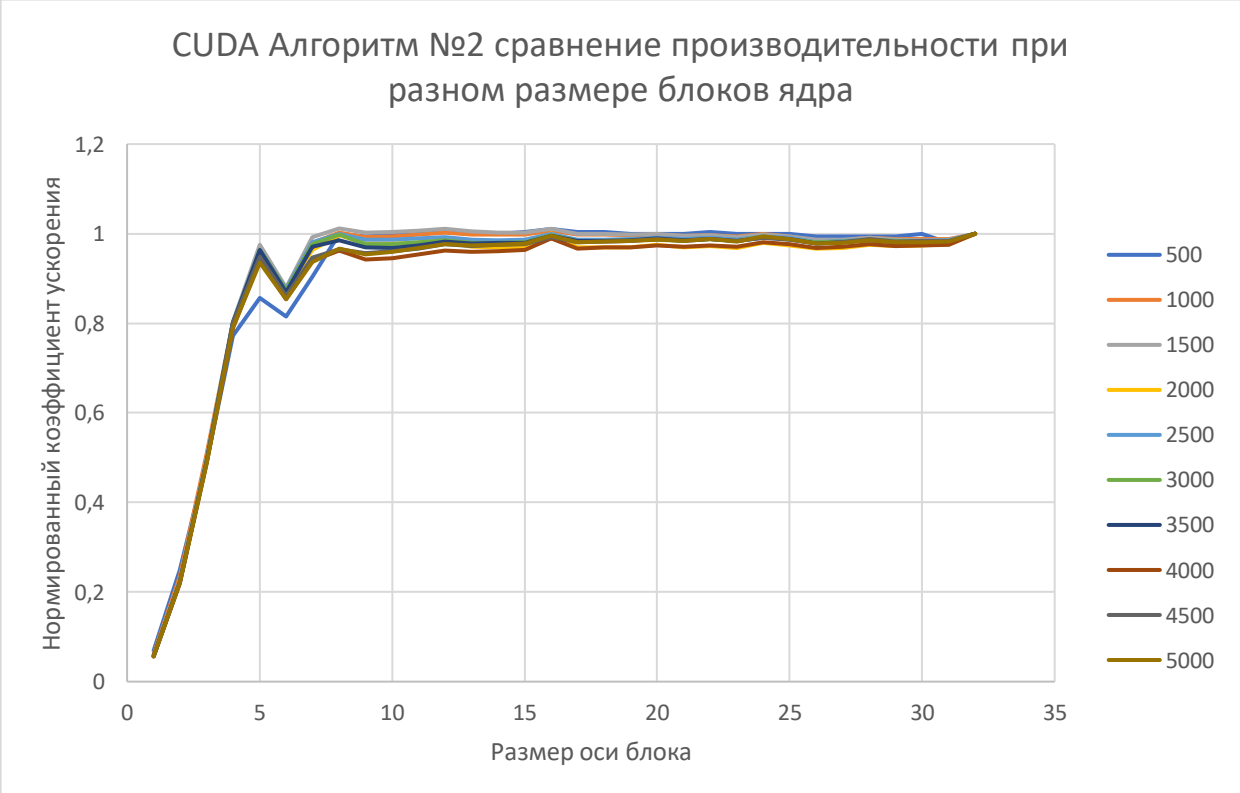

Рис. 36 Зависимость производительности GPU алгоритма от размеров блока

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

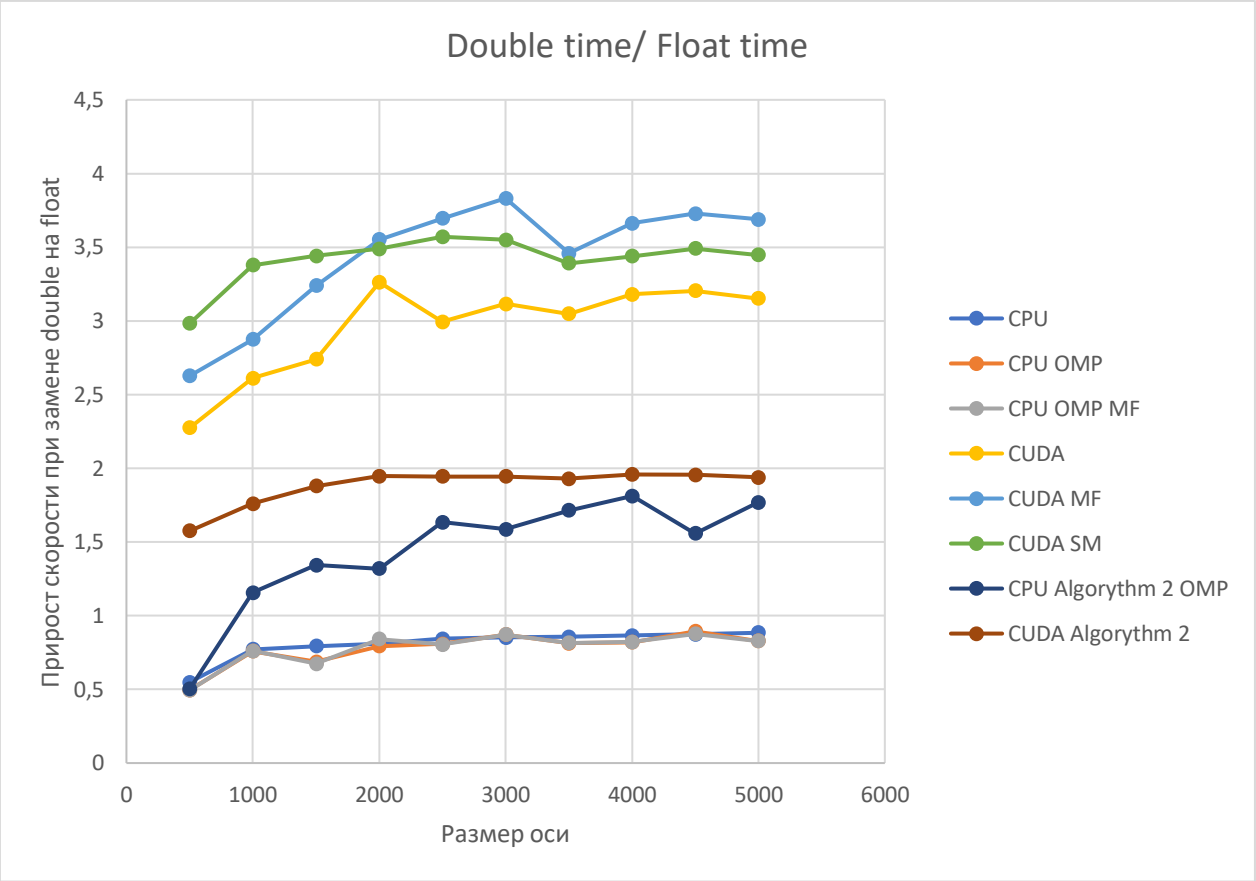

Рис. 37 Сравнения float и double версий алгоритмов. Наблюдается падение производительности, схожее с таковым на моем компьютере.

5. Заключение

В рамках научно-исследовательской практики была написана программа, реализующая различные варианты решения волнового уравнения методом конечных разностей на центральном процессоре и на видеокарте с поддержкой CUDA, проводящие замеры производительности методов. Были произведены замеры данных о времени вычисления методов на двух конфигурациях ПК: на моем персональном компьютер и на база вычислительного кластера GPUlab, находящимся на факультете ПМ-ПУ. 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.

Список использованных источников:

[1] 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

[2] ALLEN A., RAGHUVANSHI N.: Aerophones in Flatland: Interactive Wave Simulation of Wind Instruments. ACM Trans. Graph. 34, 4 (July 2015)

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.