

Теория параллелизма

## Отчет

Решение уравнения теплопроводности

Выполнил: Золотых Игорь, гр. 22933

16.05.2024

**Цель:** реализовать решение уравнение теплопроводности (разностная схема – пятиточечный шаблон) в двумерной области на равномерных сетках ( $128^2$ ,  $256^2$ ,  $512^2$ ,  $1024^2$ ).

**Используемый компилятор:** pgcc/pgc++ с ключами: “-acc”, “-Minfo=all”.

**Используемый профилировщик:** Nsight systems

**Замеры времени работы** призывали с помощью библиотеки <chrono>.

## Выполнение на CPU

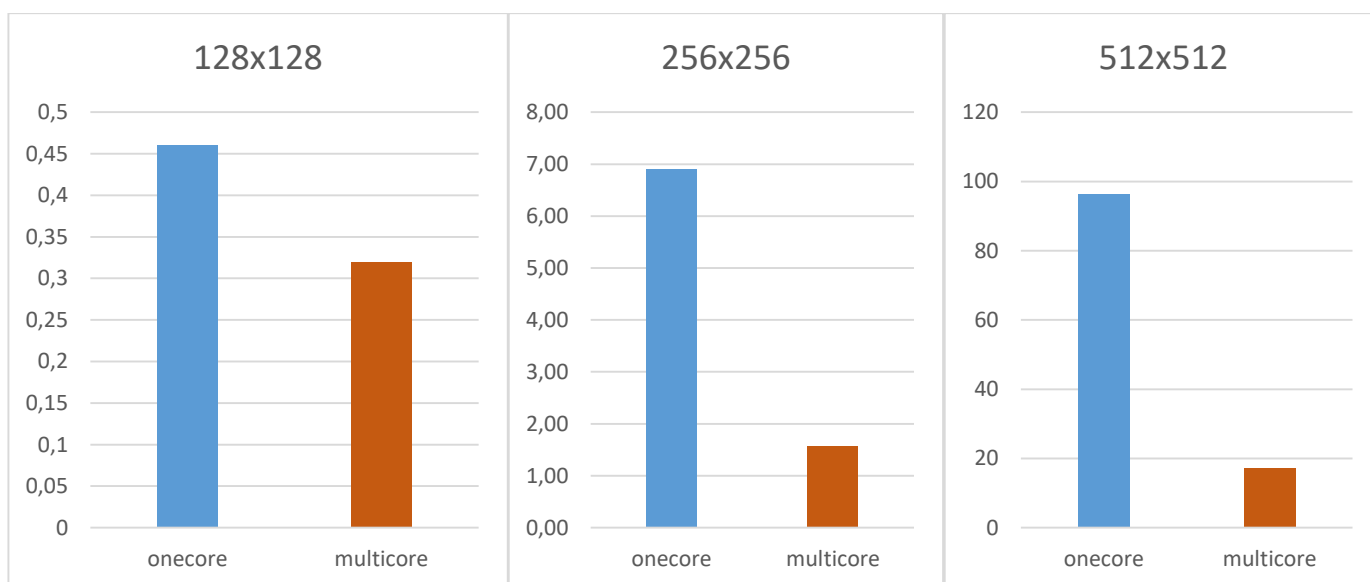
### CPU- onecore

Размер сетки	Время выполнения(с)	Точность	Кол-во итераций
128x128	0.46	1e-6	31000
256x256	6.9	1e-6	103000
512x512	96.27	1e-6	340000

### CPU-multicore

Размер сетки	Время выполнения(с)	Точность	Кол-во итераций
128x128	0.32	1e-6	31000
256x256	1.58	1e-6	103000
512x512	17.28	1e-6	340000
1024x1024	168.26	1e-6	1067000

## Диаграммы



## Выполнение на GPU

### Этапы оптимизации на сетке 512x512

Версия	Время выполнения(с)	Точность	Максимальное количество итераций	Комментарии
1	764	1e-6	1 000 000	Понадобилось 340тыс. итераций для достижения заданной точности, использовались, очень много времени тратится на обмен данными между устройствами.
2	4.03	1e-6	1 000 000	Оптимизированы циклы и копирование данных, количество итераций то же.

В **первой версии** явно видно, что очень много времени(почти все!!!, +-97%) тратится на копирование данных с CPU на GPU и обратно. На работу самой программы было затрачено от всего времени. Нужно оптимизировать обмен данными. Так же не были оптимизированы циклы, это стоит исправить.



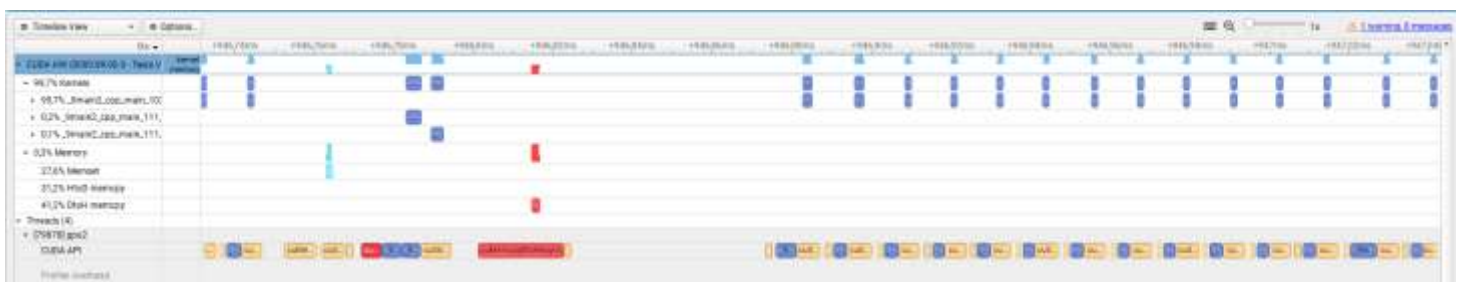
Зеленое – копирование с CPU на GPU, красное – наоборот, синее – операции.

Во **второй версии** были убраны лишние копирования данных, теперь это делается гораздо меньше раз, следовательно занимает в разы меньше времени. Так же были оптимизированы циклы. Были использованы атрибуты из **OpenACC** independent и collapse.

*Independent* нужен для того, чтобы сделать итерации цикла независимыми друг от друга.

*Collapse* в свою очередь нужен для того, чтобы «слить» два цикла в один. Это позволяет распределить итерации более эффективно.

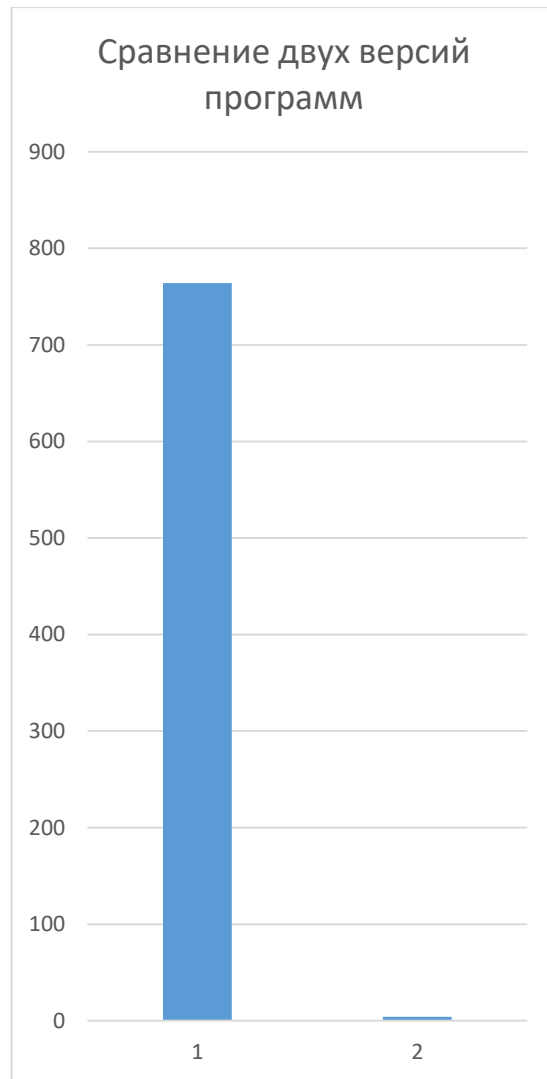
Все эти манипуляции дали очень значительный прирост в производительности.





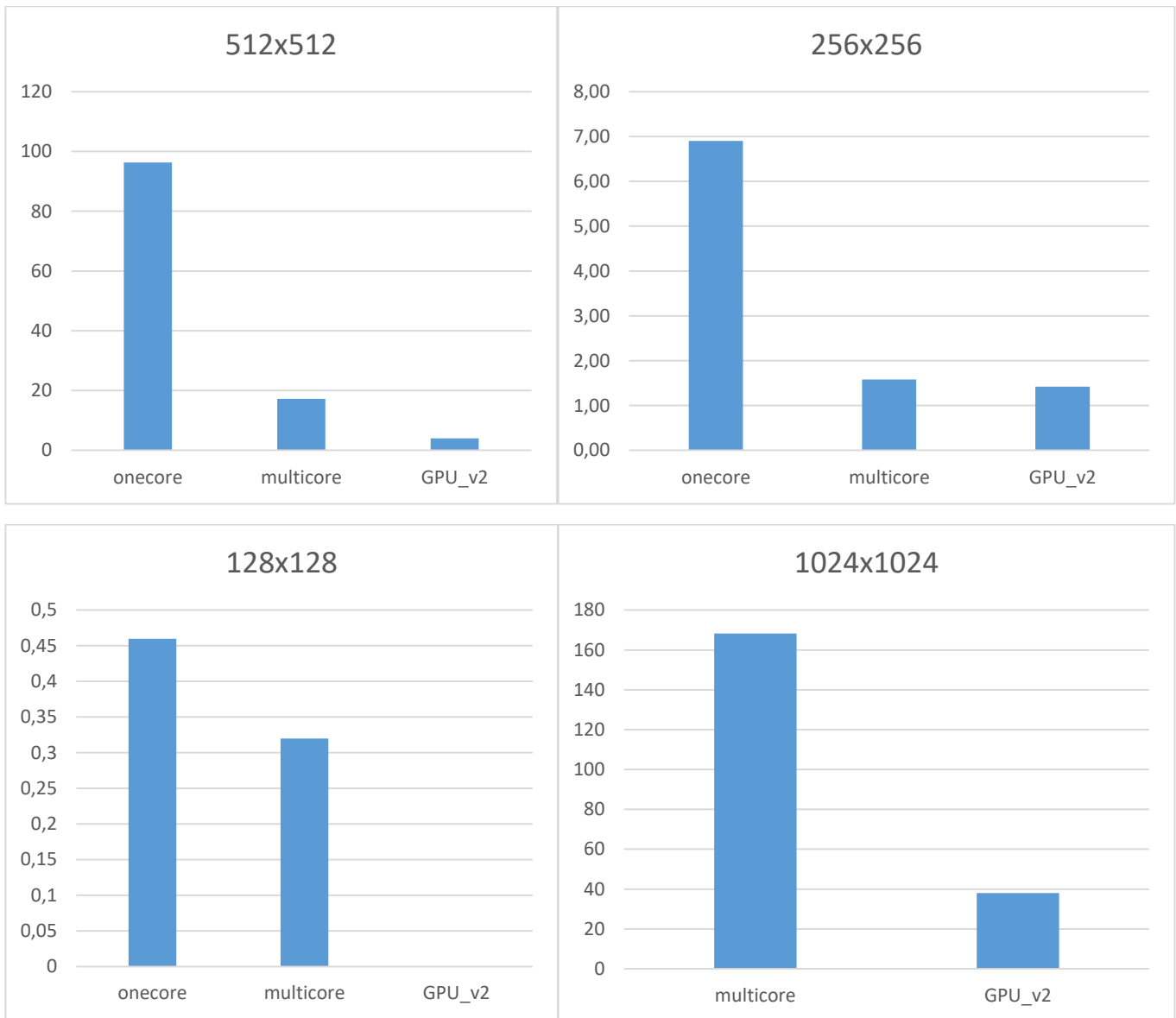
### Диаграмма оптимизации

(по горизонтали номер этапа; по вертикали время работы)



Размер матрицы	Время выполнения	Точность	Кол-во итераций
128x128	0.32	1e-6	31000
256x256	1.42	1e-6	103000
512x512	4.03	1e-6	340000
1024x1024	37.95	1e-6	1067000

Диаграммы сравнения времени работы CPU-one,  
CPU-multi, GPU(оптимизированный вариант) для разных размеров сеток



## Вывод

После написания нескольких кодов, которые работают по-разному, можно сказать, что на маленьких матрицах достаточно многопоточной реализации на CPU, но если матрицы большие, то гораздо производительнее оказывается GPU, особенно если правильно управлять операциями с памятью.

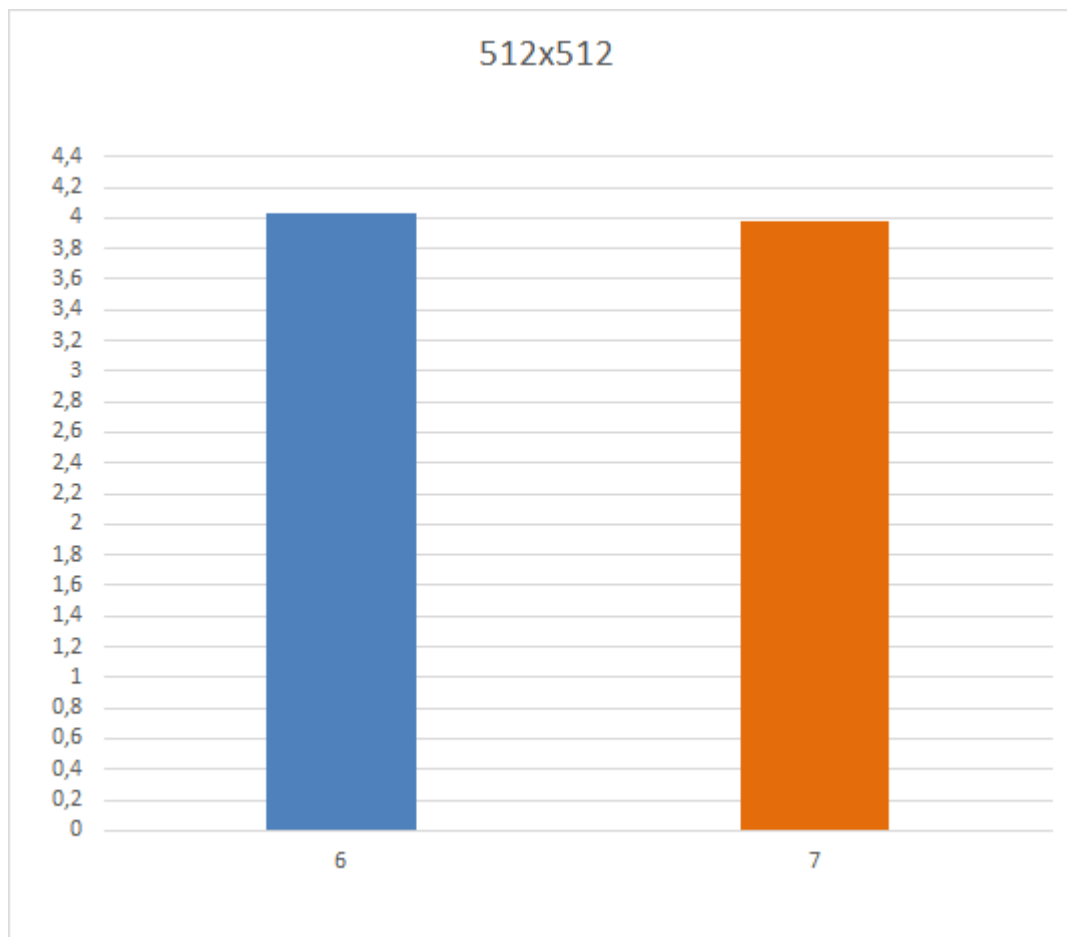


## 7 задание

Для расчета ошибки через *cublas* использовались функции *cublasDaxpy* для вычитания матриц и *cublasIdamax* для поиска максимального значения. Результат оказался положительным, но прирост в производительности был небольшой, так как тратится время на *cudaMemcpy*.



Диаграмма оптимизации итогового варианта из 6 задания и из 7 на матрице 512x512  
(по горизонтали номер этапа; по вертикали время работы)



Этап	Время выполнения(сек)	Точность	Кол-во итераций
6	4.03	1e-6	340000
7	3.98	1e-6	340000

Диаграммы сравнения времени работы CPU-one,  
CPU-multi, GPU(оптимизированный вариант) и GPU(cublas) для разных размеров сеток

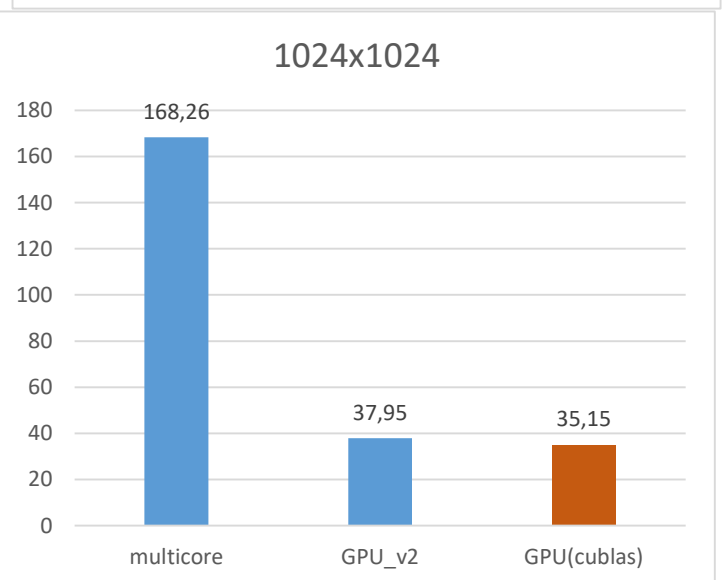
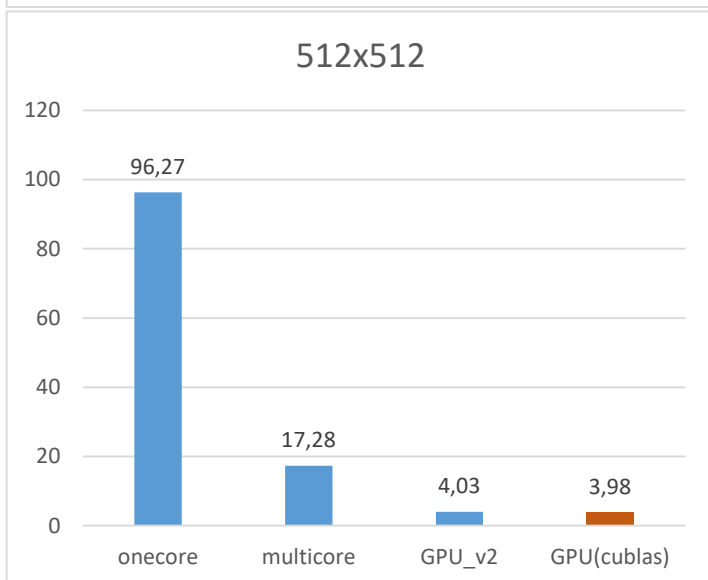
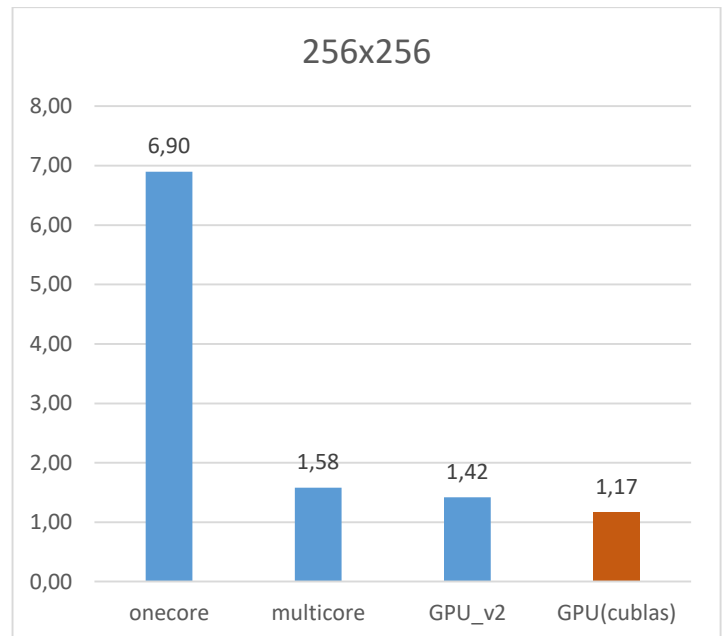
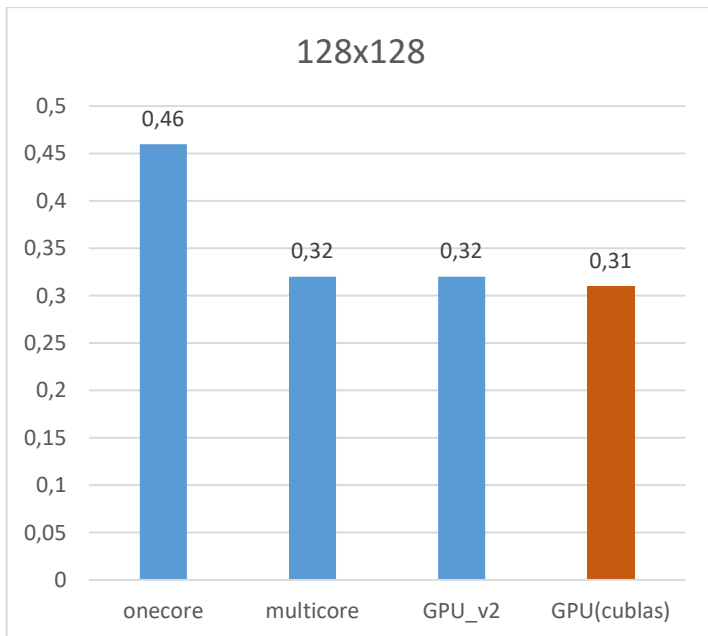


Таблица результатов для разных размеров матриц

Размер матрицы	Время выполнения(сек)	Точность	Кол-во итераций
128x128	0.31	1e-6	31000
256x256	1.17	1e-6	103000
512x512	3,98	1e-6	340000
1024x1024	35.15	1e-6	1067000

### Вывод

Использование функций из библиотеки `cublas` для оптимизации вычисления ошибки, не сильно дало прирост в производительности, так как теперь тратится время на дополнительные операции копирования данных.

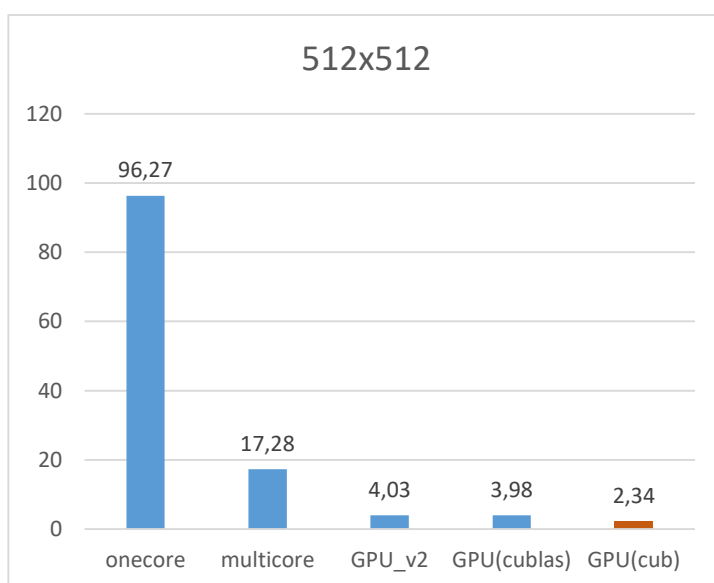
## 8 задание

Для вычисления ошибки используется функция `compute_error`, которая вычисляет локальные максимумы ошибки на каждом блоке, далее выбирается максимальное значение из этих ошибок.

Перед выполнением основного цикла создается CUDA граф с использованием функций `cudaStreamBeginCapture` и `cudaStreamEndCapture`. Эти функции захватывают последовательность операций CUDA, выполняемых в потоке CUDA. Далее граф запускается и выполняются итерации.

Диаграмма оптимизации итогового варианта из 7 задания и из 8 на матрице 512x512

(по горизонтали номер этапа; по вертикали время работы)



Этап	Время выполнения(сек)	Точность	Кол-во итераций
7	3,98	1e-6	340000
8	2,34	1e-6	340000

### Диаграммы сравнения времени работы CPU-one,

CPU-multi, GPU(оптимизированный вариант), GPU(cublas) и GPU(cub) для разных размеров сеток

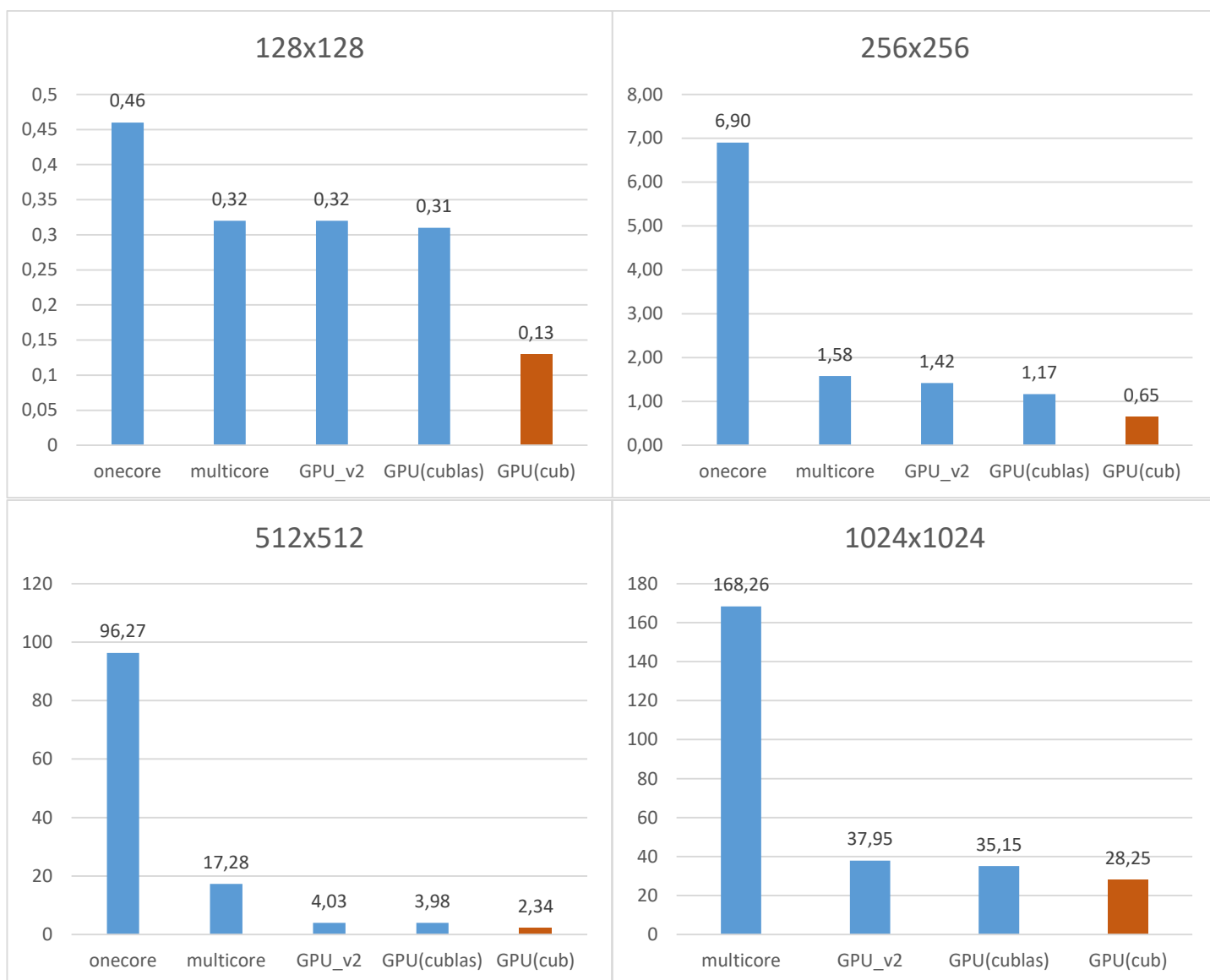


Таблица результатов для разных размеров матриц

Размер матрицы	Время выполнения(сек)	Точность	Кол-во итераций
128x128	0,13	1e-6	31000
256x256	0.65	1e-6	103000
512x512	2,34	1e-6	340000
1024x1024	28,25	1e-6	1067000

## Вывод

Использование библиотеки CUB, cudaGraph и блоков BlockReduce помогло ускорить процесс вычисления ошибки, а так же вычисления значений матрицы.