Параллелизм

Отчёт

Уравнение теплопроводности (Разностная схема)

Выполнил: Васильев Михаил Александрович

Группа: 22933

17.05.2024

**Цель работы:** реализовать решение уравнения теплопроводности в двумерной области с использованием разностной схемы и пятиточечного шаблона на равномерных сетках размером (128^2, 256^2, 512^2, 1024^2). Использовать компилятор pgcc/pgc++ с ключами "-acc" и "-Minfo=all" для сборки программы и перенести ее на GPU с помощью директив OpenACC. При замерах времени на CPU собрать данные об использовании нескольких ядер (acc=multicore).

Произвести профилирование программы с использованием "Nsight Systems" и провести оптимизацию кода. **Используемый компилятор:** pgc++.

**Используемый профилировщик:** “Nsight Systems”.

**Как производили замер времени работы:** для замера времени работы программы использовал библиотеку <chrono>.

Выполнение на CPU

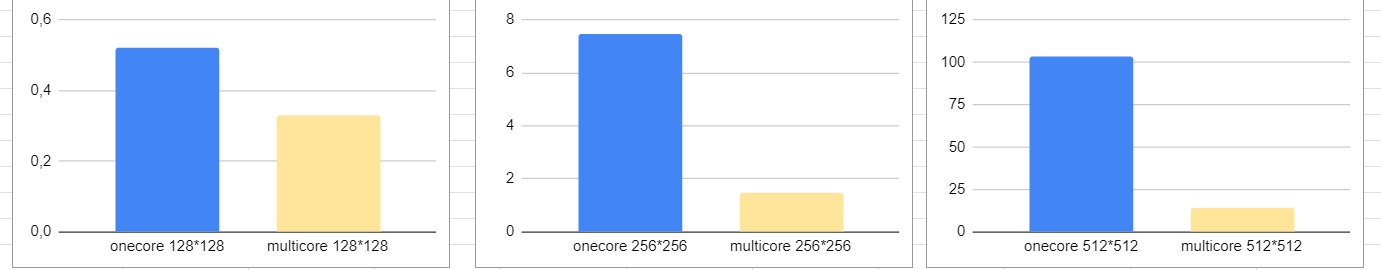
CPU – onecore

|  |  |  |  |
| --- | --- | --- | --- |
| Размер сетки | Время выполнения, в секундах | точность | Кол. - итераций |
| 128x128 | 0.520 | 1e-6 | 31000 |
| 256x256 | 7.470 | 1e-6 | 103000 |
| 512x512 | 103.671 | 1e-6 | 340000 |

CPU – multicore

|  |  |  |  |
| --- | --- | --- | --- |
| Размер сетки | Время выполнения, в секундах | точность | Кол. - итераций |
| 128x128 | 0.33 | 1e-6 | 31000 |
| 256x256 | 1.48 | 1e-6 | 103000 |
| 512x512 | 14.47 | 1e-6 | 340000 |
| 1024x1024 | 163.9 | 1e-6 | 1067000 |

Диаграмма сравнения время работы CPU-one и CPU-multi

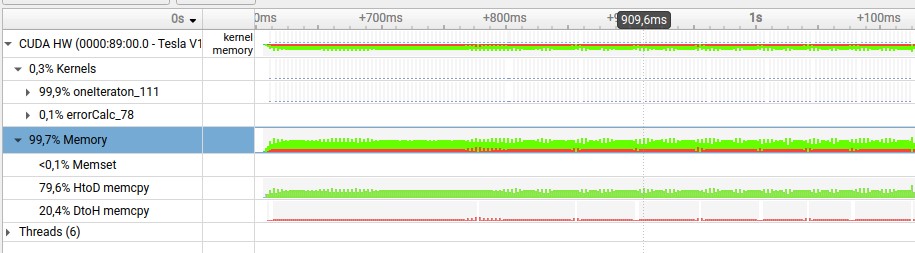


Выполнение на GPU

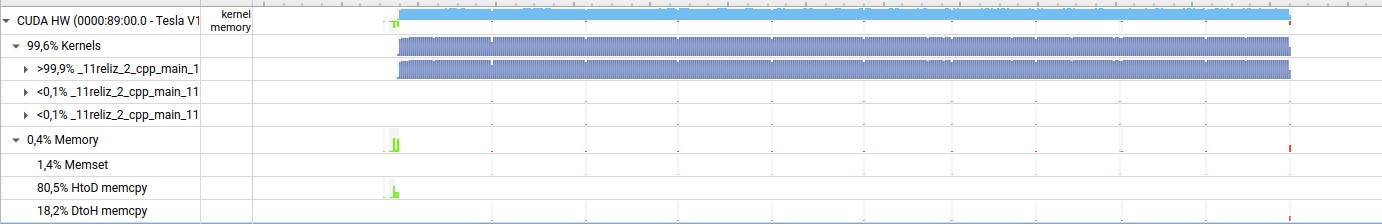
Этапы оптимизации на сетке 512х512

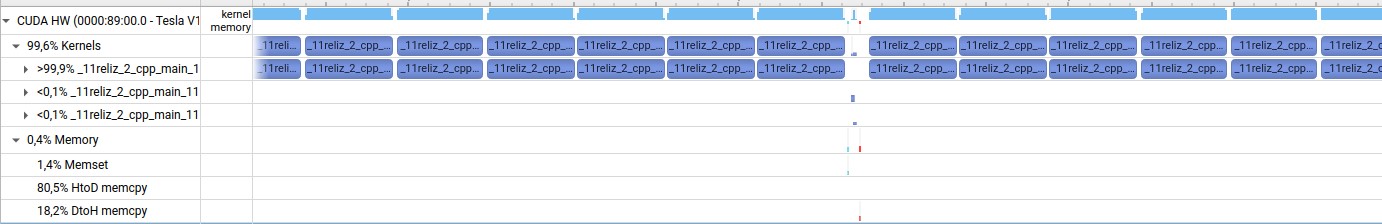
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Этап № | Время выполнения в секундах | Точность | Макс. - кол. итераций | Комментарии |
| 1 | 573 | 1e-6 | 1000 000 | Первая версия, точность 1e-6 достигнута на итерации - 345 000 |
| 2 | 85.2 | 1e-6 | 1000 000 | Вторая версия, после  оптимизации |
| 3 | 4.6 | 1e-6 | 1000 000 | Третья версия, после оптимизации |

**Первая рабочая версия:** можно увидеть, что 99,7% времени затрачено на операции с памятью (а именно 79,6% времени на обмен данными между хостом CPU и устройством GPU и 20,4% на возвращение данных на хост CPU) и 0.3% времени работы программы было затрачено на исполнение ядер. Это довольно плохой показатель для программы, для устранения этой проблемы я оптимизировал обмен данными между хостом CPU и устройством GPU.



**Вторая рабочая версия:** я оптимизировал синхронизацию данных между CPU и GPU, убрал лишнее копирование данных с хоста на девайс, также время работы ускорил другой подход к вычислению ошибки, в первой версии программы, ошибка вычислялась на каждой итерации из-за чего программа теряла много времени на максимизацию переменной среди потоков и синхронизацию между ними, теперь максимальная ошибка отправляется раз в 1000 итераций. Благодаря этим оптимизациям получилось добиться значений: 99,4% времени затрачено на вычисления и 0,4% на синхронизацию данных между CPU и GPU. Показатели стали гораздо лучше и прирост по времени в 5 раз в сравнении с первой версией.





**Третья рабочая версия:** в последней версии при использовании директивы openACC, я добавил такие атрибуты как **indepented** и **collapse(2)**, первый атрибут указывает, что итерации цикла независимы друг от друга, это позволяет компилятору эффективнее распределить итерации этого цикла по доступным потокам, второй атрибут схлопывает два цикла в один, создавая более крупное пространство итераций одним циклом, которое можно эффективнее распараллелить. Данная оптимизация позволила сократить время на вычисления.

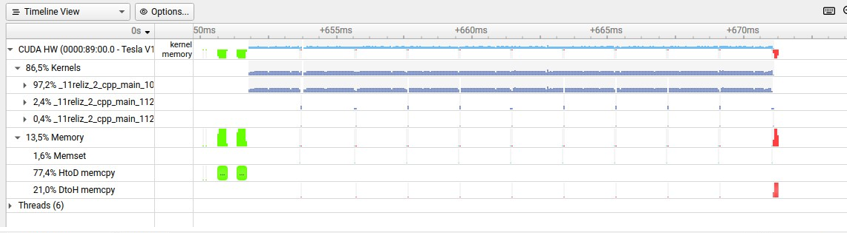
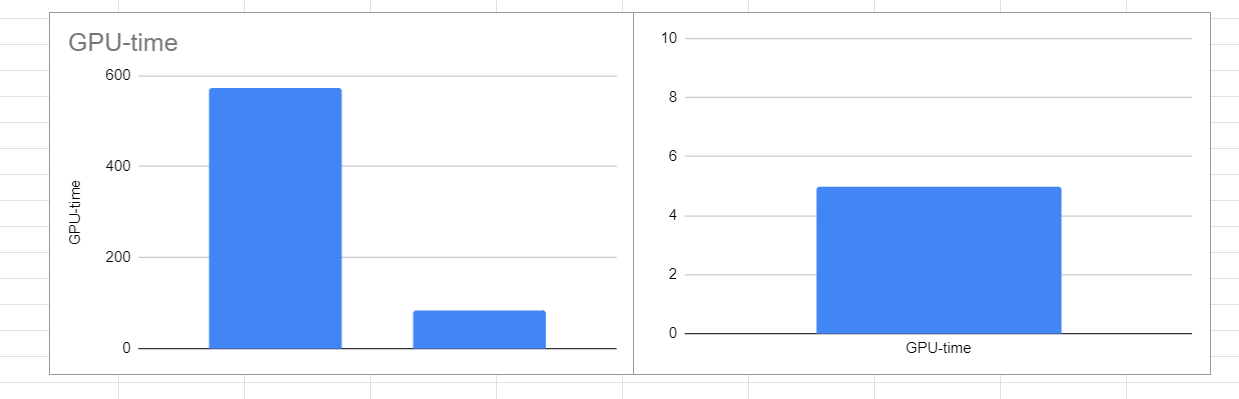
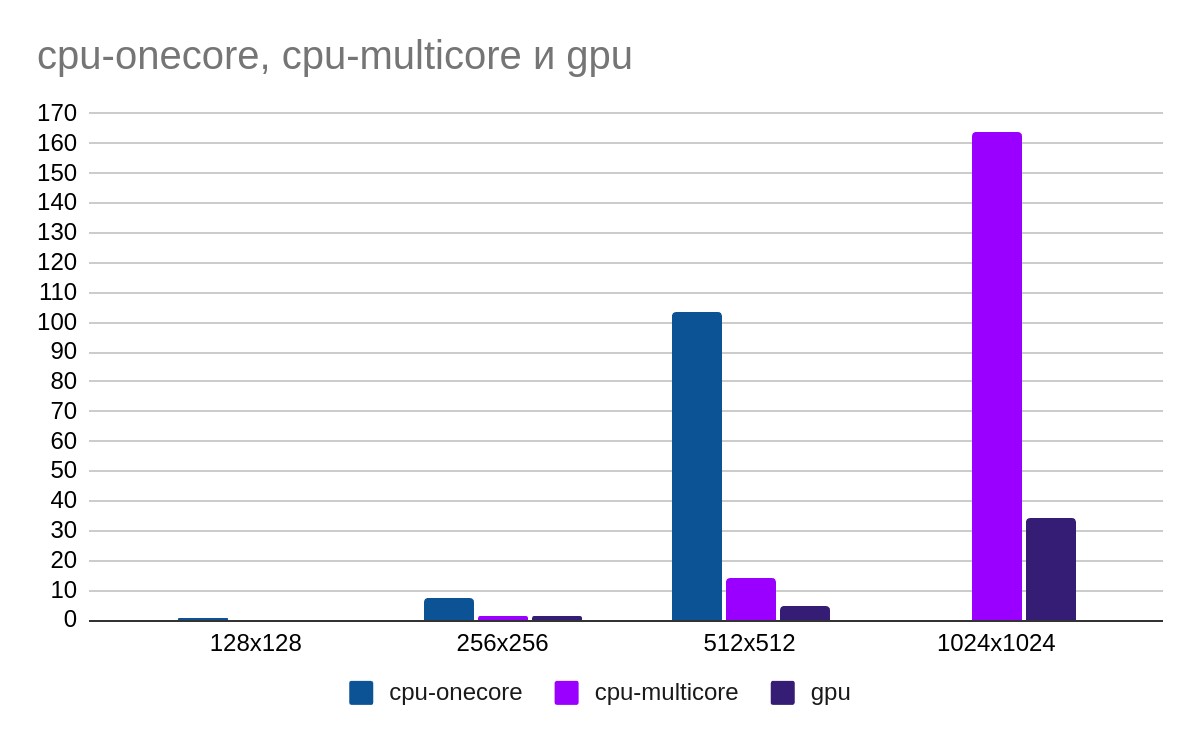


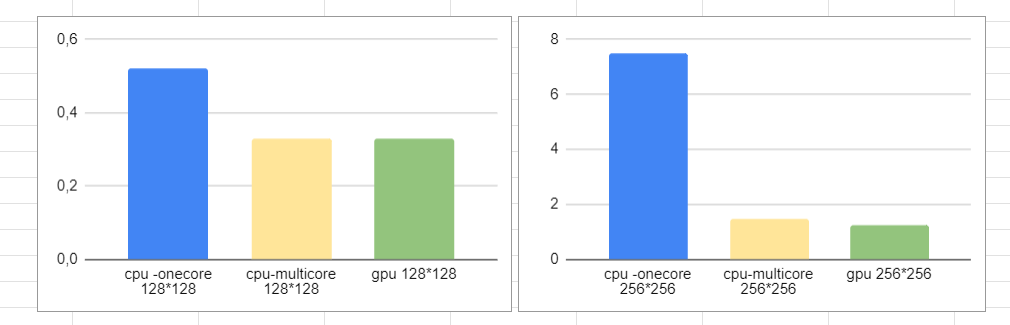
Диаграмма оптимизации



GPU - оптимизированный вариант

|  |  |  |  |
| --- | --- | --- | --- |
| Размер сетки | Время выполнения | Точность | Количество итераций |
| 128х128 | 0.33 | 1e-6 | 31000 |
| 256х256 | 1.2 | 1e-6 | 103000 |
| 512х512 | 4.44 | 1e-6 | 340000 |
| 1024х1024 | 38.6 | 1e-6 | 1067000 |





(\*Для наглядности)

Вывод: на больших матрицах лучше всего себя показывает GPU.

Дополнение к отчёту для 7 лабораторной:

В данной лабораторной вычисление максимального значения ошибки на

графическом процессоре реализовано через вызовы функций из библиотеки cuBLAS.

cuBLAS.

|  |  |  |  |
| --- | --- | --- | --- |
| Размер сетки | Время выполнения, в секундах | точность | Кол. - итераций |
| 128x128 | 0.33 | 1e-6 | 31000 |
| 256x256 | 1.2 | 1e-6 | 103000 |
| 512x512 | 4.4 | 1e-6 | 340000 |
| 1024x1024 | 38.4 | 1e-6 | 1067000 |

Диаграмма сравнения времени вычисления ошибки реализаций GPU и cuBLAS версий

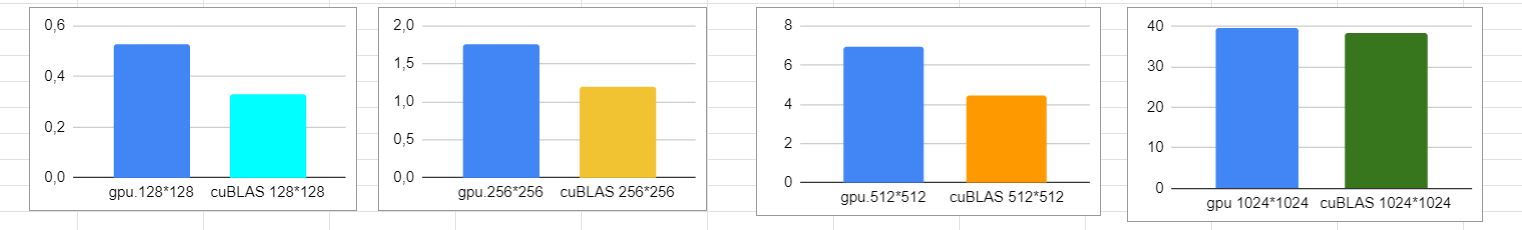
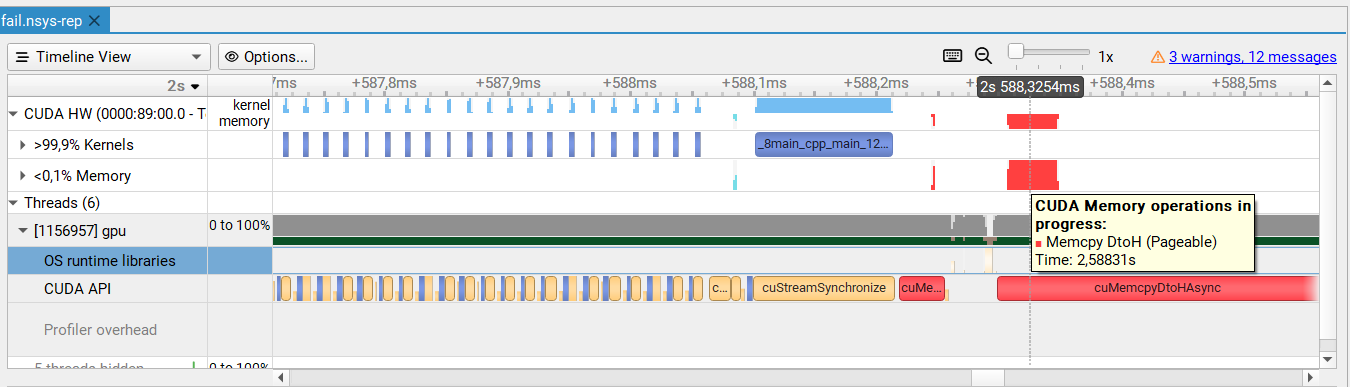


График профилирования на 1000 итераций



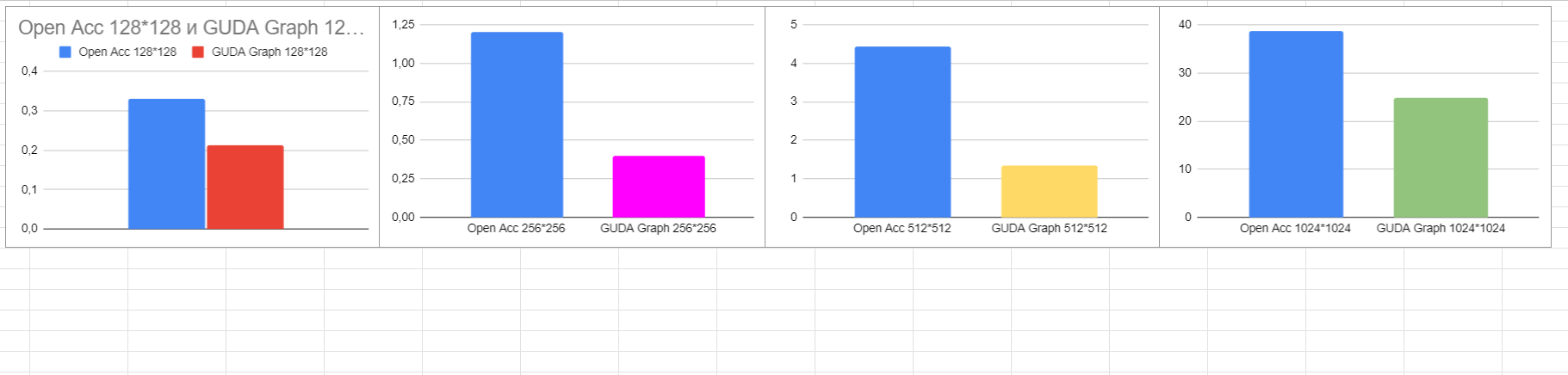
Получилось добиться значений: 99,9% времени затрачено на вычисления и 0,1% на синхронизацию данных между CPU и GPU.

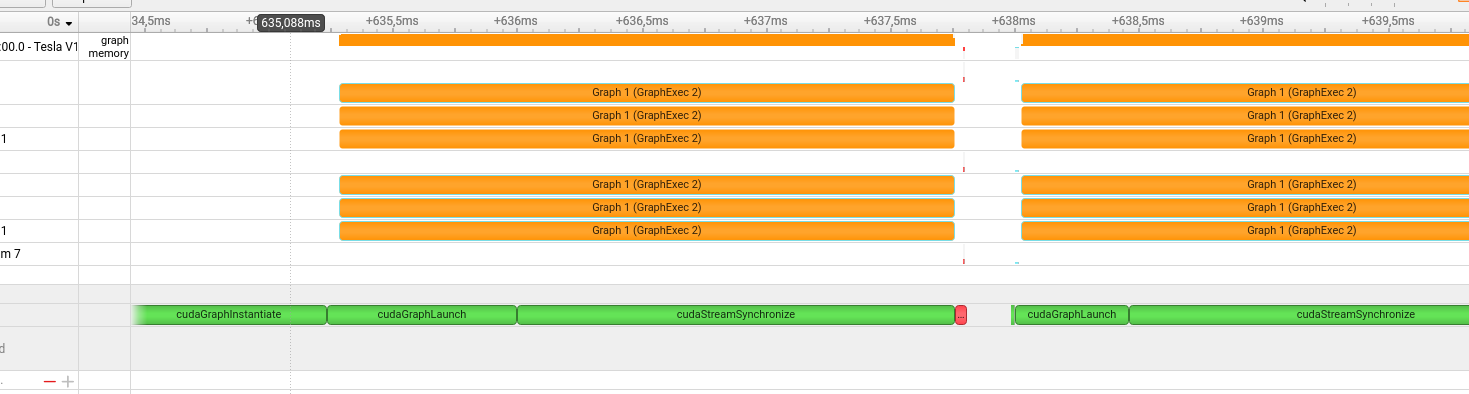
Вывод: При вычисление максимального значения ошибки на

графическом процессоре через вызовы функций из библиотеки cuBLAS я получил выигрыша в скорости в среднем на 5-10% больше нежели при вычислении на самой оптимизированной версии GPU.

CUDA Graph

|  |  |  |  |
| --- | --- | --- | --- |
| Размер сетки | Время выполнения | Точность | Количество итераций |
| 128\*128 | 0,214 | 1e-6 | 30100 |
| 256\*256 | 0,400 | 1e-6 | 100000 |
| 512\*512 | 1,357 | 1e-6 | 339600 |
| 1024\*1024 | 24,751 | 1e-6 | 1066700 |

****



Вывод: Результаты демонстрируют, что при различных размерах матриц CUDA Graph демонстрирует более высокую производительность по сравнению с OpenACC.