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 призводили с помощью библиотеки <chrono>.

Выполнение на CPU

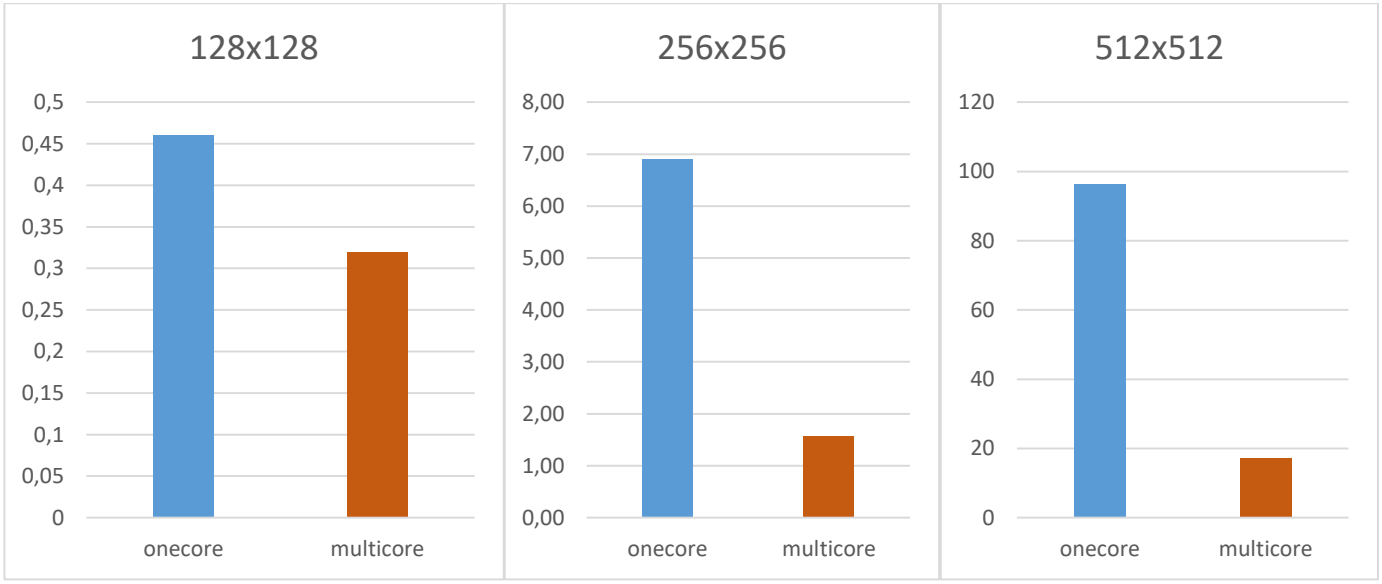
CPU- onecore

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.

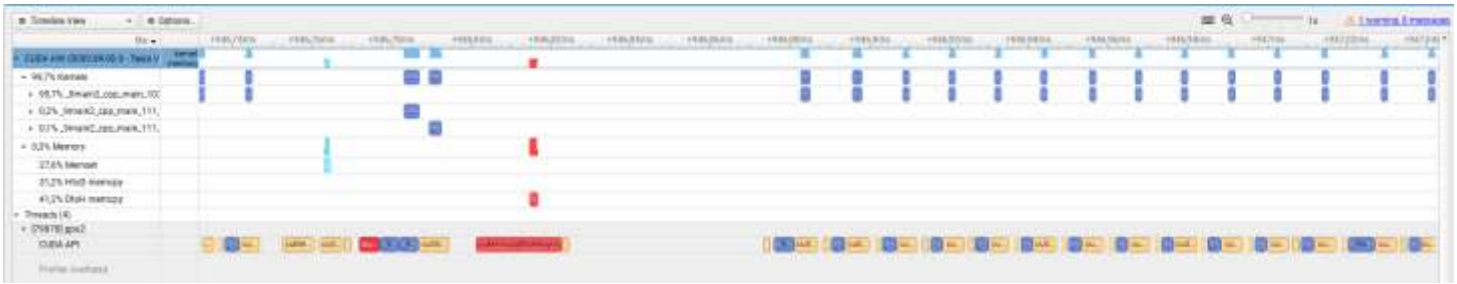
Зеленое – копирование с CPU на GPU, красное – наоборот, синее – операции.

Во **второй версии** были убраны лишние копирования данных, теперь это делается гораздо меньше раз, седовательно 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.

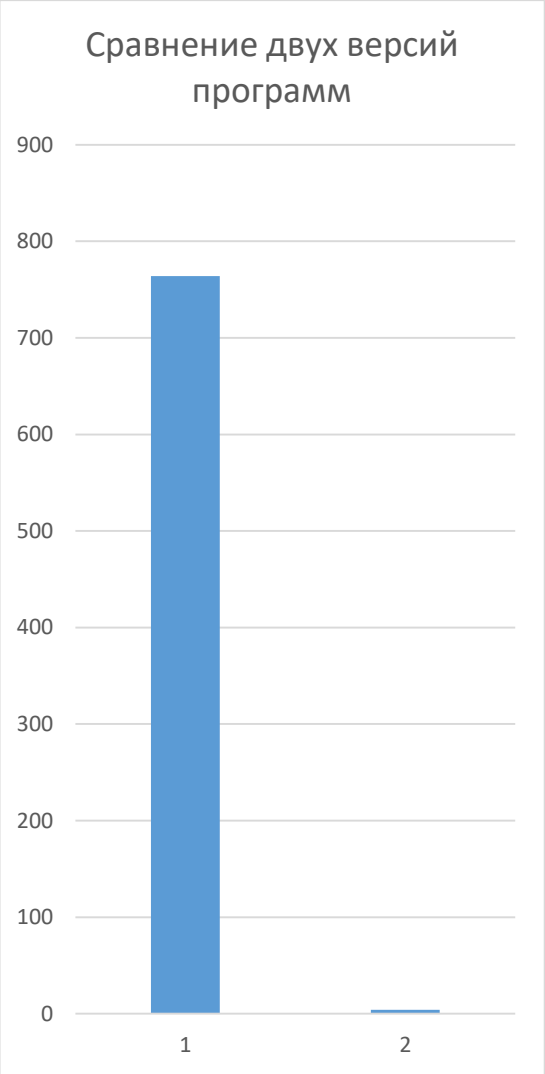
Все эти манипуляции дали очень значительный прирост в производительности.




0s
+933,7ms
+933,75ms
+933,8ms
+933,85ms
+933,9ms
+933,95ms
+934ms
+934,05ms
CUDA HW (0000:89:00.0 - Tesla V
kernel
memory
99,7% Kernels
99,7% _9main2_cpp_main_10
0,2% _9main2_cpp_main_111_
0,1% _9main2_cpp_main_111_
0,3% Memory
27,6% Memset
31,2% HtoD memcpy
41,2% DtoH memcpy
Threads (4)
[79878] gpu2
CUDA API
Profiler overhead
3 threads hidden

Диаграмма оптимизации

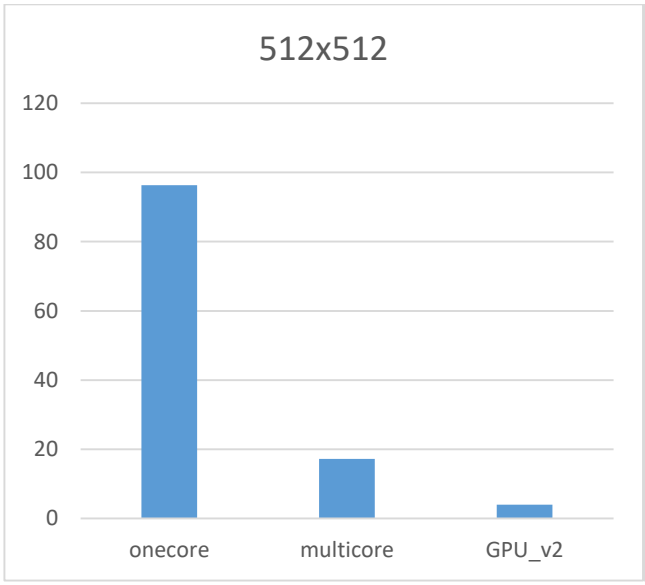
(по горизонтали номер этапа; по вертикали время работы)

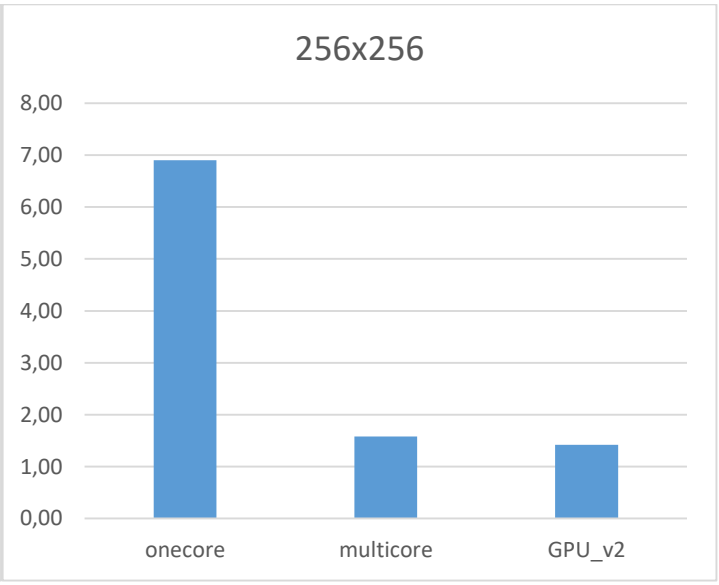


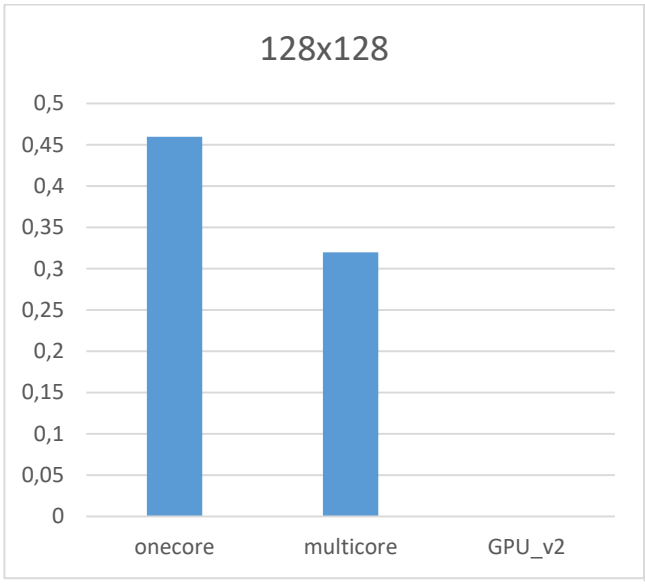
Размер матрицы	Время выполнения	Точность	Кол-во итераций
128x128	0.32	1e-6	31000
256x256	1.42	1e-6	103000
512x512	4.03	1e-6	340000
1024x1024	37.95	1e-6	1067000

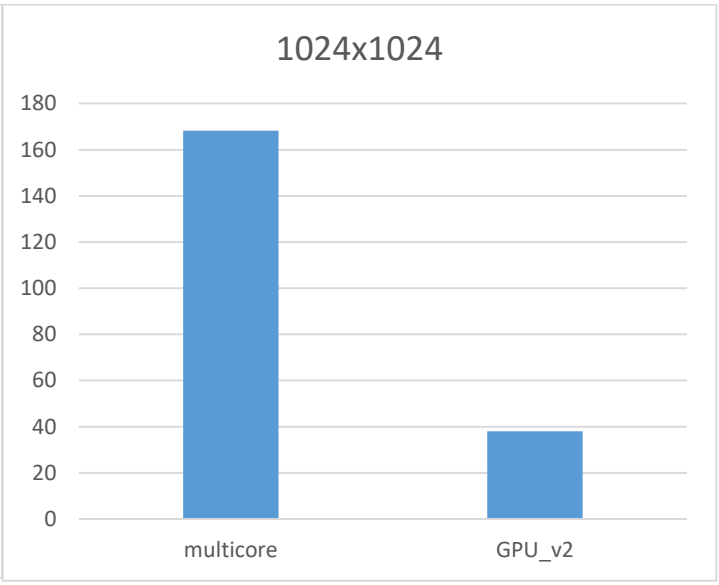
Диаграммы сравнения времени работы CPU-one,

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 паматью.