Лабораторная работа №2

по курсу

«Программирование графических процессоров»

Выполнил:

Смирнов Александр

Группа 21225

Задание

Реализовать решение уравнение теплопроводности (пятиточечный шаблон) в двумерной области на  равномерных сетках. Перенести программу на GPU используя директивы OpenACC. Сравнить скорость работы для разных размеров сеток на центральном и графическом процессоре. Произвести профилирование программы с использованием NsightSystems. Произвести оптимизацию кода.

Решение

Исходный код расположен в репозитории <https://github.com/SmirAlex/gpu-programming> в папке lab2. Решение представлено в виде трех файлов: один хэдер файл с объявлением констант, функций и типов данных; один файл с основной логикой вычислений (в том числе на GPU); и один файл с логикой чтения аргументов командной строки и проведения измерений времени.

Описание решения

Перед началом решения инициализируется сетка, у которой значения на гранях (начальные и граничные условия) задаются с помощью линейной интерполяции значения в углах сетки (они предопределены заранее).

Для вычисления ошибки создаются 2 версии сетки (под них выделяется память): current\_grid и next\_grid, которые представляют собой соответственно текущее и следующее состояние сетки на каждую итерацию алгоритма. Значения в этих сетках предварительно устанавливаются как начальной сетке. Ошибка для каждой ячейки сетки определяется по пятиточечному шаблону. Цикл прекращается, когда достигнута желаемая точность (минимизирована ошибка) или превышено максимальное число итераций. В конце значения начальной сетки обновляются в соответствии со значениями последней полученной версией сетки (current\_grid).

Использованные директивы OpenACC:

#pragma acc data copy(grid [0:N\*N]) create(current\_grid [0:N\*N]) create(next\_grid [0:N\*N])

Копируется grid на GPU и обратно на момент выхода из последующего блока, выделяется память на GPU под current\_grid и next\_grid

#pragma acc kernels loop independent collapse(2)

Kernels – параллелизацией занимается компилятор

Independent – необходим для явного указания того, что у нас нет зависимости по данным. В противном случае компилятор считает, что она есть => параллелизма нет.

Collapse(2) – схлопывание двумерного цикла в одномерный.

#pragma acc data present(next\_grid [0:N\*N], current\_grid [0:N\*N])

Используется на каждой итерации. Без нее алгоритм по какой-то причине не сходится. Вероятно, идут обращения не к той памяти.

#pragma acc kernels loop independent collapse(2) reduction(max:error)

Используется на каждой итерации. Все то же самое, только добавляем редукцию по максимизации ошибки для корректности.

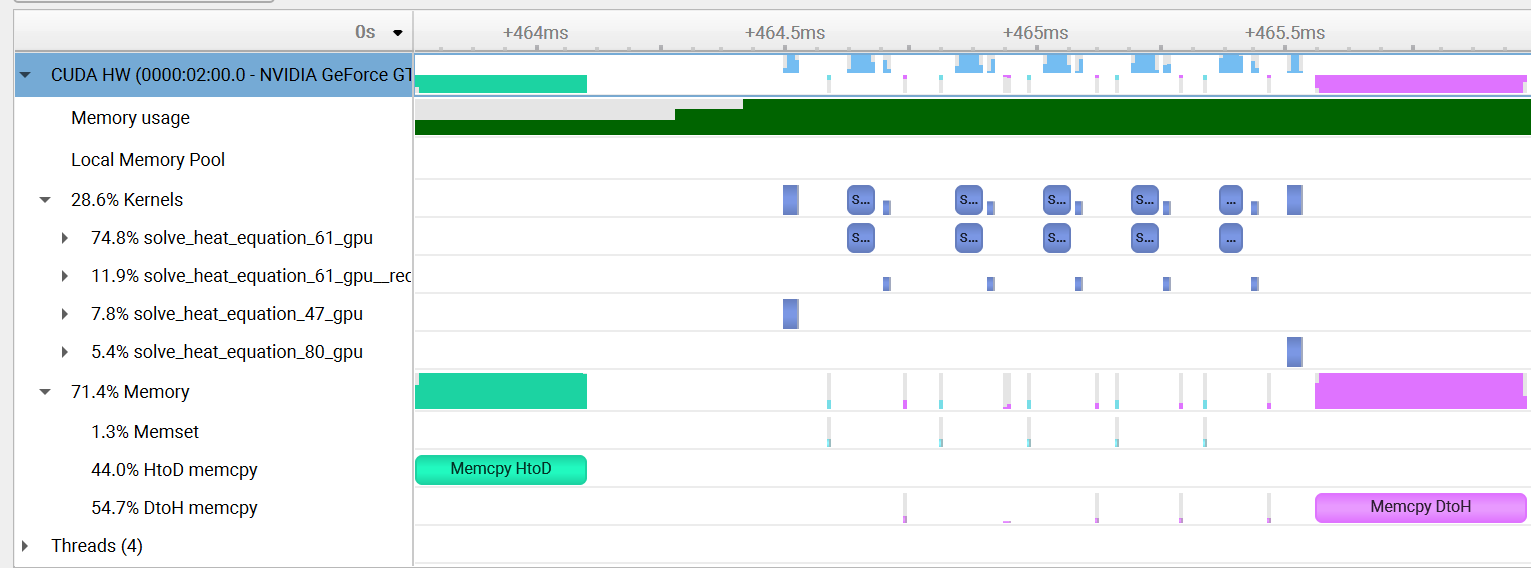
Сравнивались 3 версии компиляции программы: для CPU, для CPU с параллелизмом (с -ta=multicore), для GPU (параллелизм по умолчанию). Измерения проводились для трех размеров сетки: 128x128, 256x256, 512x512. Для каждого размера сетки подбиралась такая точность, чтобы на ~20000 итераций решение сходилось.

Для измерений была использована функция clock\_gettime из time.h, которая предоставляет дает время с точностью до наносекунд. Измерения проводились 5 раз, в результате выводилось среднее значение по итогу 5 испытаний.

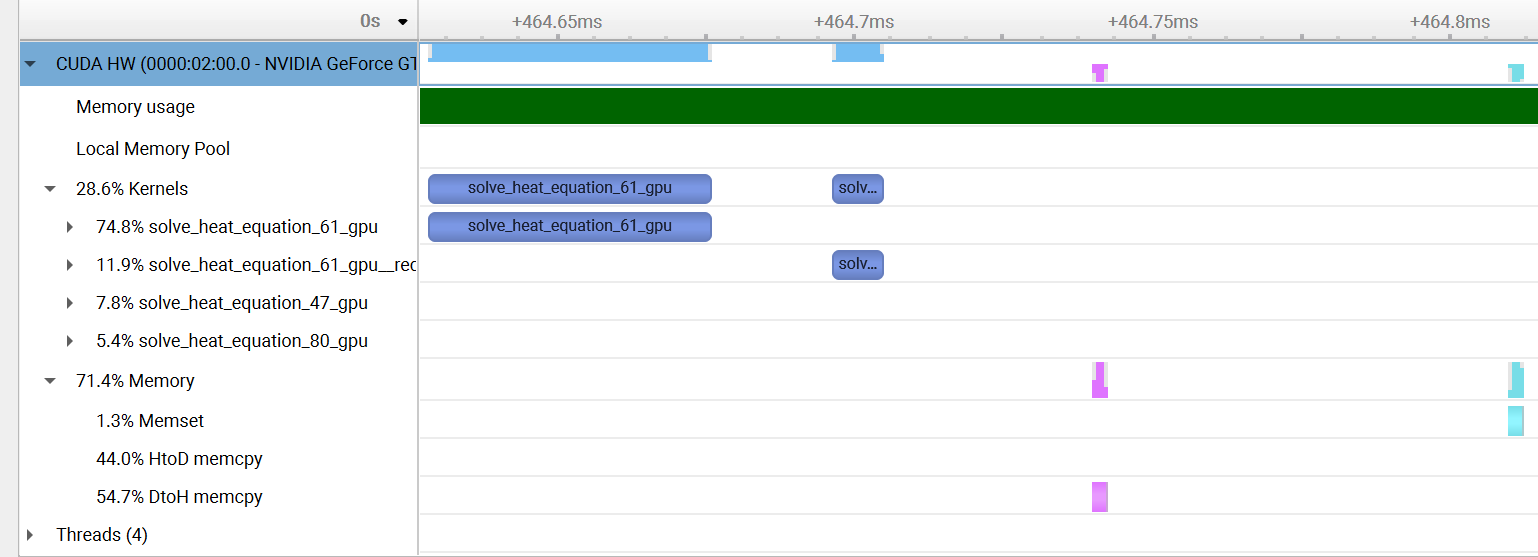
Из измерений можно сделать вывод, что вычисления на GPU на малых объемах данных (сетка 128x128) проигрывает параллельному исполнению на CPU, но с ростом объема данных GPU становится значительно быстрее. Вариант с обычным исполнением на CPU (не параллельным) всегда оказывается колоссально медленнее.

Результаты измерений отражены на следующих графиках:

На скриншотах ниже представлены результаты профилировки программы на GPU на 5 итерациях (в разных масштабах)



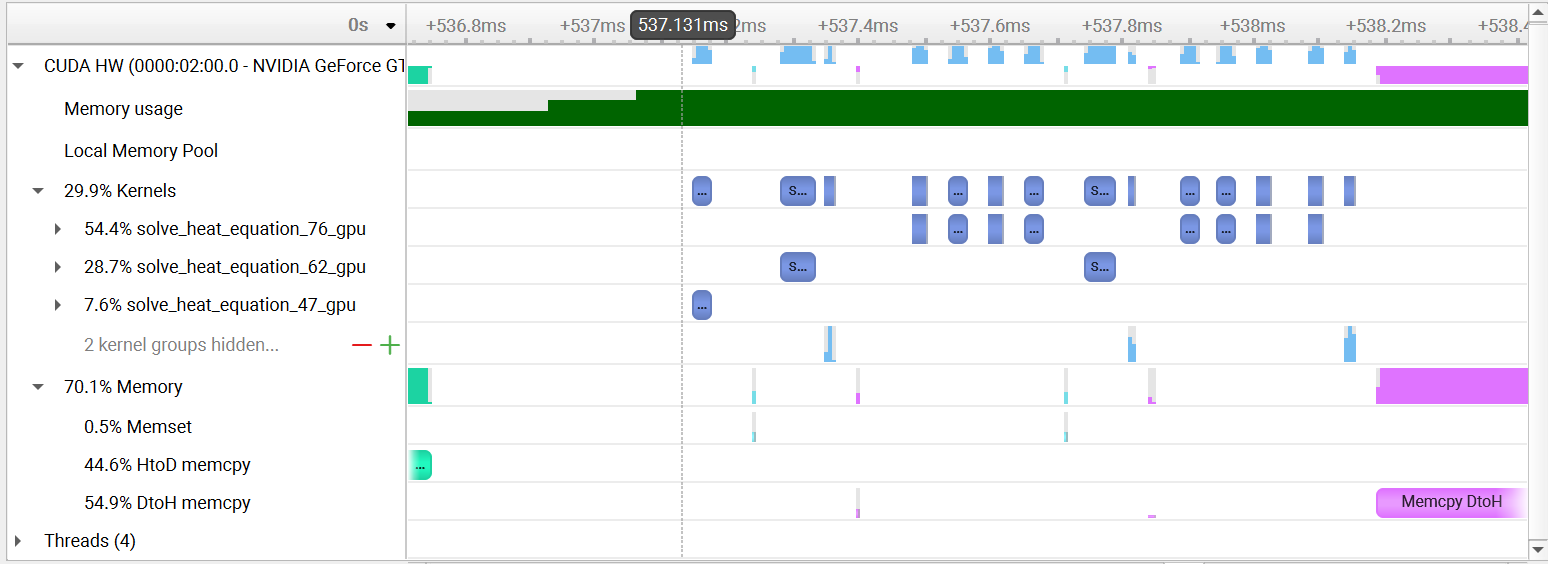




Оптимизация копирования ошибки

На скриншотах видно розовый интервал – Memcpy DtoH, бирюзовый – Memset. Можно сделать вывод, что на каждой итерации ошибка копируется с GPU на CPU и обратно. Это можно оптимизировать, если считать ошибку не на каждой итерации, а на каждой n-той итерации. В программу [был добавлен](https://github.com/SmirAlex/gpu-programming/commit/c9bffddba2d7d400b302505906089d3b7885f9a7) еще один опциональный параметр – error\_calc\_interval, который отвечает за интервал, с которым считается ошибка. По умолчанию он равен 1.

Профилировка с оптимизацией при 10 итерациях и error\_calc\_interval=5:



Можно видеть, что теперь мы скопировали ошибку только 2 раза вместо 10 раз без оптимизации.

Были проведены дополнительные измерения с разными значениями error\_calc\_interval. В результате скорость работы программы на GPU увеличилась в разы, теперь и на 128x128 сетке GPU значительно быстрее. В целом по ходу выполнения лабораторной работы можно сделать вывод, что при написании программ на GPU для наибольшего выигрыша нужно как можно сильно минимизировать передачу данных на GPU и обратно.

Результаты измерений представлены на гистограммах ниже:

Оптимизация async/wait

Из профилирования можно увидеть, что имеется большая задержка между запусками ядер, т.е. итерациями алгоритма. Задержку можно уменьшить, если ядро начнет запускаться еще до конца предыдущей итерации – это можно сделать через async и wait. Причем итерации с async могут быть только те итерации, где не происходит копирования ошибки. Асинхронные итерации попадают в одну очередь и в конечном итоге порядок исполнения не нарушается. На итерации с копированием ошибки же мы применяем wait и уже с готовыми предыдущими итерациями вычисляем ошибку (не async). Изменение представлено [здесь](https://github.com/SmirAlex/gpu-programming/commit/23a9d04b615090d279dd065a93fa6685ac6d5884).

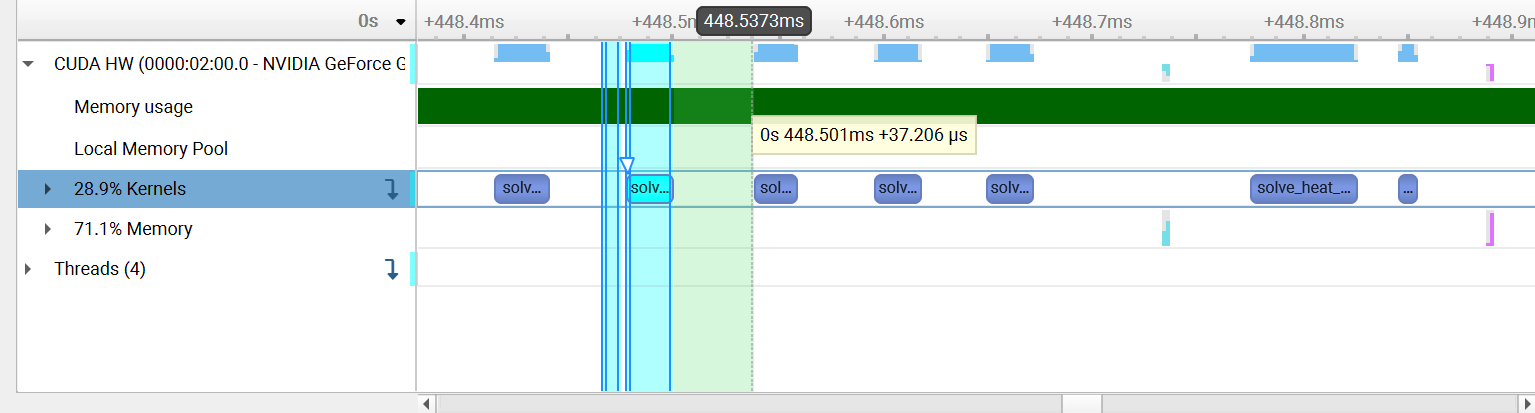
Оптимизация avoid swap

Также можно заметить, что на каждой итерации мы меняем указатели (делаем swap) текущей и следующей сетки (current\_grid и next\_grid). Swap можно не делать, если выполнять 2 итерации алгоритма на одной итерации цикла (сначала вычислять next\_grid по current\_grid, а потом current\_grid по next\_grid и т.д.). В результате они просто становятся grid1 и grid2. В результате в одной секции *acc kernels* можно исполнить сразу 2 итерации, таким образом задержка между запусками ядер для этих двух итераций должна свестись к минимуму. То есть фактически время половины всех задержек должно быть уменьшено. Изменение представлено [здесь](https://github.com/SmirAlex/gpu-programming/commit/77195c2e556ed14f6a0b236795a0c47c4df97638).

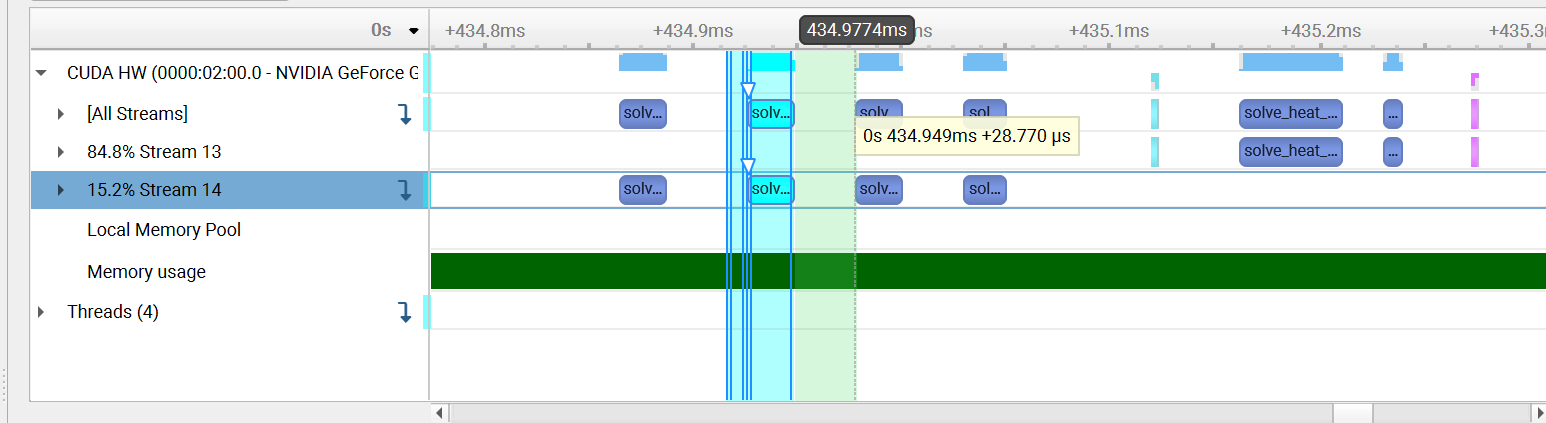
Далее будет представлены результаты профилирования и замеры времени 3-х версий программы:

* Only error\_calc\_interval optimization (version 1)
* Error\_calc\_interval + async/wait optimizations (version 2)
* Error\_calc\_interval + async/wait + avoid swap optimizations (version 3)

Version 1:

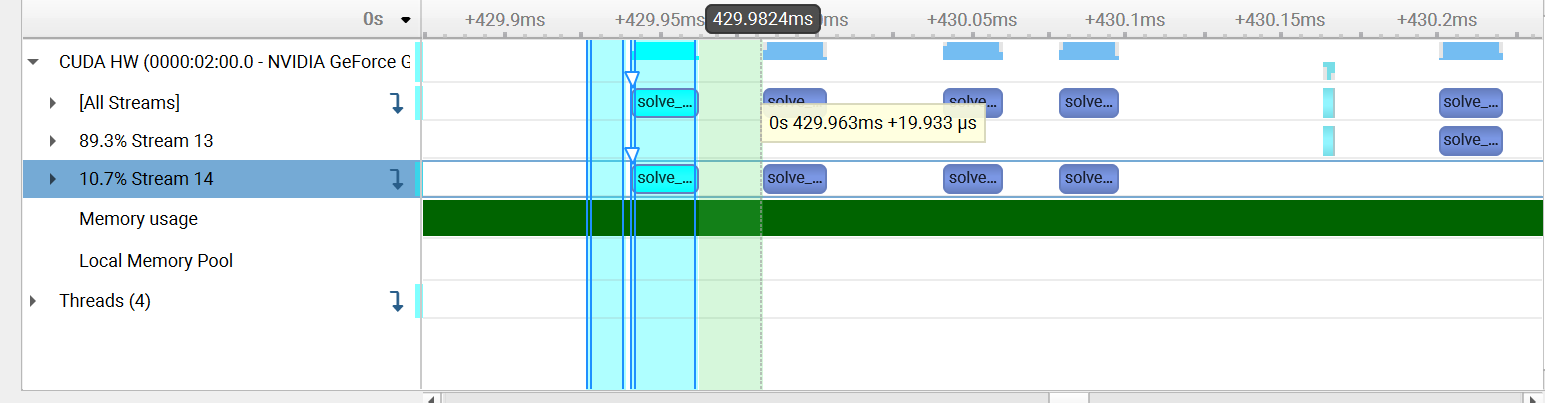


Version 2:



Задержка значительно уменьшена. Так же можно видеть, что у нас появилось 2 потока (Stream 13 и Stream 14) за счет async/wait.

Version 3:



Можно видеть что задержка уменьшена между 1-м и 2-м, а также 3-м и 4-м запусками ядер, как и предполагалось.

Можно видеть, что оптимизация async/wait принесла наибольший прирост скорости в сравнении с avoid swap.