Лабораторная работа №4

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 CUDA. Операцию редукции (подсчет максимальной ошибки) реализовать с использованием библиотеки CUB. Сравнить скорость работы для разных размеров сеток на графическом процессоре с предыдущей реализацией на OpenACC.

Решение

Исходный код расположен в репозитории <https://github.com/SmirAlex/gpu-programming> в папке lab4. Решение представлено в виде 4-х файлов: один хэдер файл с объявлением констант, функций и типов данных, один файл с логикой чтения аргументов командной строки и проведения измерений времени, и 2 файла с логикой решения на CUDA, каждый с разным подходам к измерению ошибки в процессе работы алгоритма.

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): grid1 и grid2, которые представляют собой состояния сетки соответственно нечетных и четных итерациях алгоритма. Значения в этих сетках предварительно устанавливаются как в 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.

Описание работы с CUDA

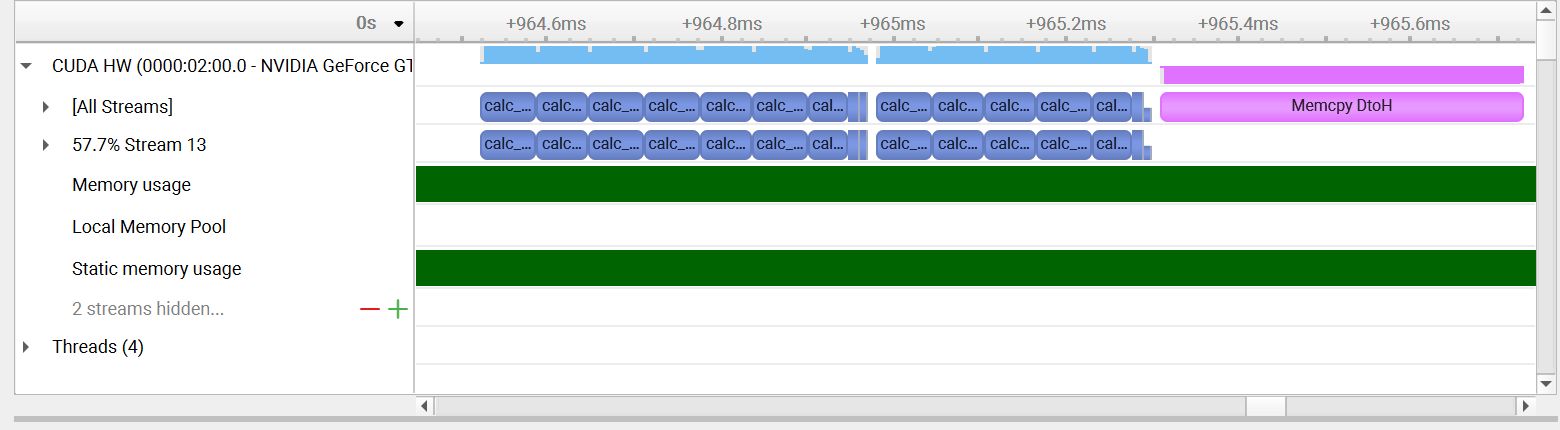
В данной лабораторной работе решение уравнение теплопроводности производилось с помощью набора инструментов для вычислений на GPU **CUDA** (**Compute Unified Device Architecture**). За основу была взята последняя версия программы для решения уравнения используя OpenACC. Далее будет описана последовательность библиотечных вызовов CUDA:

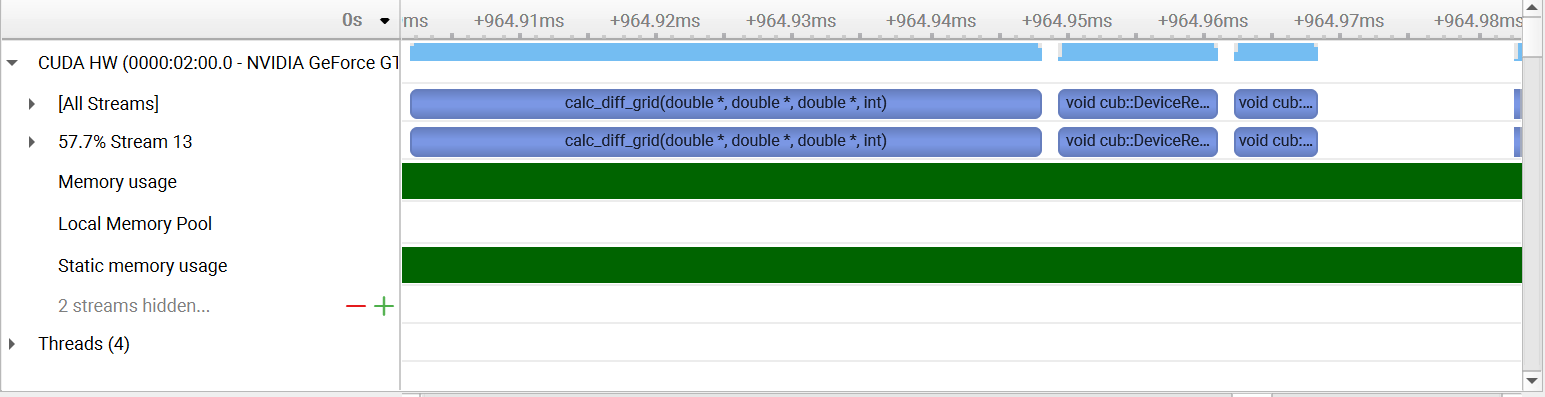
* cudaHostRegister – регистрация init\_grid из host для использования в CUDA (page-lock памяти)
* cudaMalloc – выделение памяти на device для сеток (начальное и следующие состояния)
* cudaStreamCreate – создание CUDA потоков для асинхронного исполнения задач
* cudaDeviceSynchronize – синхронизация всех потоков
* dim3 threads\_per\_block и num\_blocks – размер CUDA блока (количество потоков в нем) и количество CUDA блоков для нашей сетки. Исходя из документации было решено взять стандартный размер блока – 16\*16 (256 потоков)
* cudaHostAlloc – выделение памяти под ошибку на хосте (page-lock, благодаря чему может использоваться и на device)
* Для шага алгоритма было написано CUDA ядро (kernel) – особая функция, исполняющаяся в CUDA параллельно. Это ядро определяет индекс массива, соответствующий текущему потоку, и обновляет сетку для следующей итерации алгоритма.
* Все шаги алгоритма исполняются асинхронно. Синхронизация происходит только после вычисления ошибки для корректной оценки граничных условий цикла.

Версия 1 вычисления ошибки

В данной версии для вычисления ошибки в отдельном CUDA ядре находится разность между двумя сетками, после чего по новой сетке разностей производится редукция для нахождения максимальной разности (ошибки) с помощью библиотеки CUB (cub::DeviceReduce::Max). Данный способ вычисления ошибки похож на то, как мы вычисляли ошибку при помощи cuBLAS. Характерная черта – простота, но в результате мы вынуждены считать ошибку отдельно от вычисления шагов алгоритма, что добавляет накладные расходы.

Профилировка версии 1 с помощью NVIDIA Nsight Systems

(10 итераций с вычислением ошибки на каждой 6-й итерации)

Ядра для вычисления ошибки

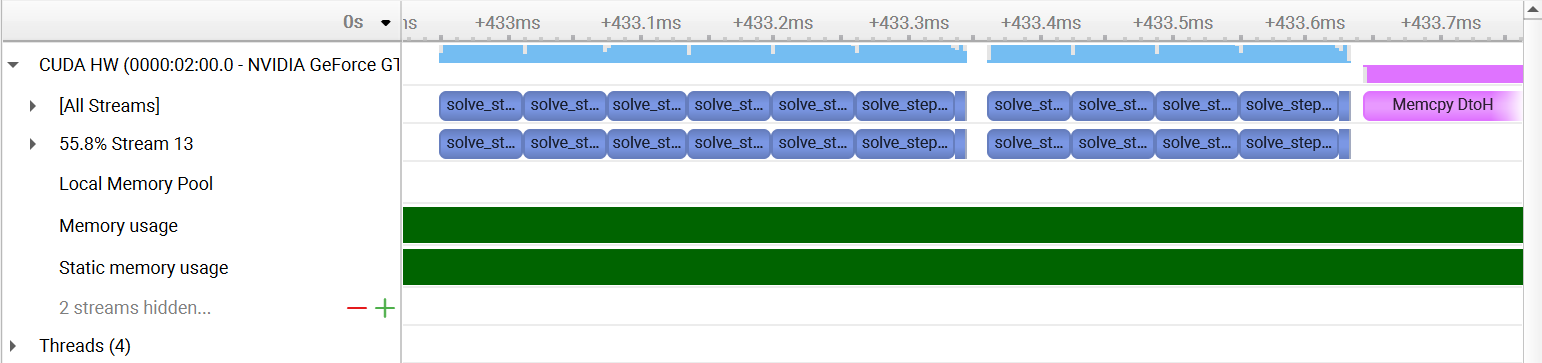
Можно видеть, что для каждого вычисления ошибки запускается 3 ядра – одно для вычисления разности, одно для редукции по блокам, и одно для общей редукции по device. Это не оптимально, особенно в сравнении с вариантом на OpenACC, где для вычисления ошибки нужно было всего лишь одно дополнительное ядро. Попробуем улучшить.

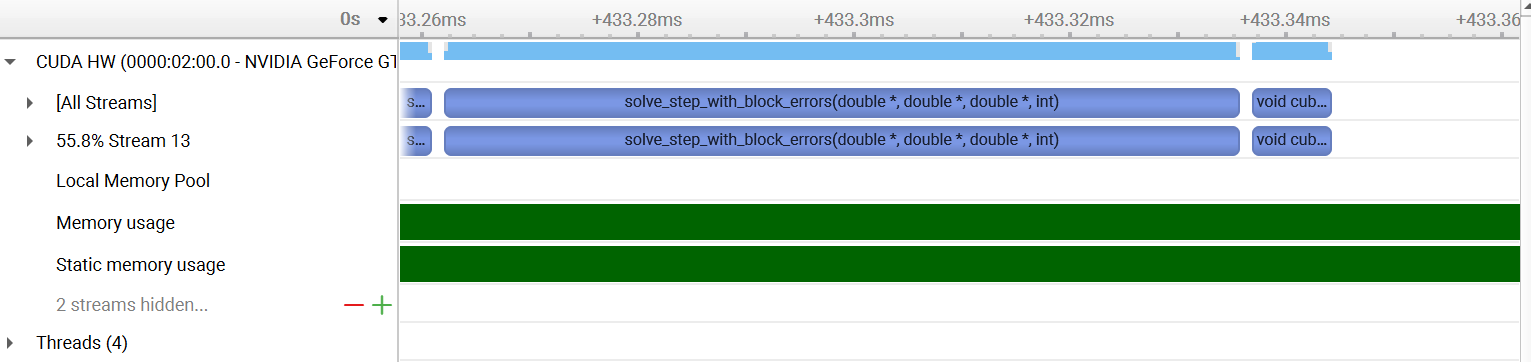
Версия 2 вычисления ошибки

В данной версии ошибка вычисляется явно для каждого отдельного CUDA блока с помощью редукции BlockReduce из CUB. Ошибки для блоков хранятся в отдельном массиве. Далее по ошибкам из этого массива вычисляется общая ошибка с алгоритма на текущем этапе с помощью уже знакомой функции cub::DeviceReduce::Max.

Более того, нахождение ошибки для отдельных CUDA блоков выполняется одновременно с вычислением шага алгоритма, т.е. не требуется запускаться для этого дополнительное ядро.

Профилировка версии 2 с помощью NVIDIA Nsight Systems

(10 итераций с вычислением ошибки на каждой 6-й итерации)

Ядра для вычисления ошибки

Можно видеть, что процесс вычисления ошибки для блока совмещен с шагом алгоритма (вычислением следующего значения сетки), а для вычисления общей ошибки запускается всего одно дополнительное ядро.

Замеры времени

Сравнивались 3 версии реализации алгоритма на GPU: версия на OpenACC из лабораторной 2, версия на CUDA с вычислением ошибки по способу 1, версия на CUDA с вычислением ошибки по способу 2. Измерения проводились для двух размеров сетки: 256x256, 512x512. Для каждого размера сетки подбиралось такое количество итераций, чтобы алгоритм сходился с точность 10^(-6). Ошибка вычисляется на каждой 100-й итерации. Для CUDA брались размеры блоков 16\*16.

Исходя из профилирования, можно видеть, что версия на CUDA работает значительно медленнее, чем на OpenACC. Причем с ростом данных ситуация ухудшается. Причину такого поведения мне найти не удалось… Возможно, дело в неправильной синхронизации или есть более быстрые функции. Из положительного – имеется небольшой выигрыш от оптимизации вычисления ошибки в версии “CUDA block errors”.

Оптимизация программы на CUDA

Для выяснения причины столь сильного проигрыша во времени версии на CUDA к версии на OpenACC было решено сравнить время вычисления отдельных шагов алгоритма (время работы ядер). В ходе профилирования было выявлено, что причина разницы общего времени выполнения 2-х версий программ кроется именно в медленном вычислении одного шага алгоритма – на OpenACC вычисление занимало 20 μs, а на CUDA вычисление занимало 60 μs (в 3 раза дольше). В ходе тщательного анализа кода было выявлена главная причина медленного выполнения – заполнение значений новой сетки производилось в неправильном порядке (не в том, как выделена память), т.е. индексы вычислялись через интервал N (ребро сетки).

Индекс массива раньше: int grid\_index = i \* N + j;

Индекс массива раньше: int grid\_index = j \* N + i;

Вероятнее всего здесь имеет место работы кэша: раньше были последовательные кэш-промахи, в то время как при правильном обходе массива мы идем вдоль уже загруженной кэш-строки. В результате исправления данной ошибки скорость работы увеличилась примерно в 2 раза (50%).

Другая важная оптимизация – добавление ключевого слова \_\_restrict\_\_ перед аргументами в функциях, соответствующих обрабатываемым сеткам. Данное ключевое слово говорит компилятору, что указатель адресует область памяти, на которую не ссылается никакой другой указатель. Данная оптимизация увеличила скорость примерно на треть (30%).

Также экспериментально было выявлено, что наиболее оптимальный размер CUDA-блока равен 8 (раньше было 16). Таким образом, общее количество блоков возросло в 4 раза, что позволило увеличить параллелизм и общее время выполнения примерно на 10%.

Дополнительные оптимизации:

* Использование cudaMallocPitch вместо cudaMalloc и других аналогичных функций. Преимущество использования функций с *pitch* заключается в более эффективной для последующего доступа аллокации данных 2D / 3D массивов.
* Аллокация и освобождение 2х сеток теперь производится одним вызовом, вместо двух (теперь один кусок памяти, разделенный на 2 части).
* Использование *unsigned int* вместо *int* для избежания лишних кастов

[Ссылка на изменение с оптимизациями](https://github.com/SmirAlex/gpu-programming/commit/e1b35be9c21ed0782457c00d70e6ebaa78a82775)

Замеры времени

Можно видеть, что после произведенных оптимизаций над CUDA-программой она выполняется быстрее, чем на OpenACC, причем с ростом данных разница в скорости возрастает.