Московский государственный университет имени М. В. Ломоносова
Факультет вычислительной математики и кибирнетики
Кафедра автоматизации систем вычислительных комплексов

Отчёт по заданию №1

Оптимизация программ.

Вариат №2

Никифоров Никита Игоревич, 321 группа

Москва
2019

Оглавление

Постановка задачи

Основная задача в этой работе – оптимизировать данную программу. Изначальная программа была реализованная на языке си, для её оптимизации возможно было использовать всё, при условии что сохраняется семантика программы.

Конфигурация ВС

Процессор	AMD Ryzen 7 2700X 4.2GHz
Видеокарта	Nvidia GTX 1060 3GB
Оперативная память	DDR4 16 Gb 3400 MHz
Операционная система	Arch Linux

История исследования

Изначально было принято решение попробовать скомпилировать программу с максимальной оптимизацией компилятора `-O3`, что дало прирост примерно в 4.5 раза. После было прочитанно несколько статей на тему оптимизации вычислений с матрицами. Практически в каждой предлогалось использовать векторные инструкции процессора. После просмотра ассемблерного кода, компилятора, выяснилось что после применения `-O3`, компилятор сам применил векторные инструкции (Это видно на данном куске `objdump -D`).

```
11a4: 48 83 c0 08 add 0x8, %rax
11a8: c4 a1 7b 10 04 06 vmovsd (rsi,r8,1),%xmm0
11ae: c5 fb 58 40 f8 vaddsd -0x8(%rax)%xmm0,%xmm0
11b3: c5 f3 58 c8 vaddsd %xmm0,%xmm1,%xmm1
11b7: c5 fb 11 09 vmovsd %xmm1,(%rcx)
```

Соответственно необходимоть использовать их вручную отпала. После было принято решение распараллеливания данной задачи, поскольку в пользовании находится 8 ядерный 16 поточный процессор, в программе можно было использовать большое количество потоков (точнее все 16). К счастью, задача очень просто раскладывалась на потоки, которые вообще не пересекались между собой. После распараллеливания программы прирост составил ещё 7.5 раза. Как итог программа стала выполняться за 90мс, против изначальных 3200мс. Далее было принято решение попробовать провести вычисления на видеокарте. Была написана программа с использованием CUDA, хотя и вычисления проходили быстро, технические функции (инициализация ГПУ, копирование данных в память и синхронизация с основным потоком программы) занимали примерно 300мс, что уже было в три раза медленнее, чем вычисления на процессоре. Но при больших объёмах данных вычисления на ГПУ будут куда более быстрые, чем на процессоре. Спустя несколько дней, когда уже часть отчёта была написана, в голову пришла светлая идея, а что если реализовать данные вычисления не с помощью кубического алгоритма (как это было изначально), а придумать квадратичный алгоритм. И такой алгоритм был найден (подробное описание в параграфе ...), время выполнения на одном потоке стало 11мс, что дало общий прирост в 290 раз. К сожалению дальнейшие попытки распараллеливания квадратичного алгоритма не увенчались успехом (программа выполнялась либо больше, либо столько же)

Измерения времени

Все измерения производились с помощью утилиты `time`, по 5 раз на тест, и бралось среднее значение.

описание	**время (mc)**	**ускорение (раз)**	**название файла**
Исходный код без оптимизаций	3200	1	p2var.c
Исходный код скомпилированный с `-O3`	1190	2.6	test_1.c
Распараллеливание на 16 потоков	500	6.4	test_4.c
Распараллеливание на 16 потоков с `-O3`	90	35.5	test_4.c
Решение задачи на видеокарете спомощью `CUDA`	500	6.4	test.cu
Решение задачи квадратичным алгоритмом	11	290.9	test_6.c

Описание методов оптимизации

Применение оптимизаций компилятора — компилятор сам анализирует программу и оптимизирует её под конкретную архитектуру.
Использование векторных инструкций процессора — позволяет процессору складывать вектора чисел за 1 одну инструкцию, что даёт огромное ускорение
Использование ГПУ для вычислений — позволяет значительно ускорить вычисления с матрицами, для достаточно больших объёмов данных.
Написание более оптимального алгоритма — если существует возможность реализовать алгоритм с меньшей асимптотической сложностью (а потом к нему применять предыдущие оптимизации)

Описание оптимизаций компилятора

Компилятор умеет однотипные операции: такие как циклы, разворачивать и применять векторные инструкции процессора, что позволяет ускорить программу в несколько раз. Это видо из первых двух тестов. Компиляция с `-O3` всегда полезна в независимости от программы (кроме случая, когда её нужно отладить).

Описание использования ГПУ

GPU или ГПУ отличное устройство для вычислений с матрицами, так как GPU – множество потоковых процессоров, которые могут выполнять небольшие задачи. Соответственно данную программу можно было обсчитать на GPU очень быстро. Но, набор данных в изначальной программе слишком мал, чтобы увидеть преимущество вычислений на GPU, так как одна инициализация занимает почти 300mc. Для наглядной демонстрации были запущены тесты с рапараллеливанием, исходным алгоритмом с оптимизациями, квадратичный алгоритм и вычисления на GPU на матрице с размером стороны 8192.

описание	**время (mc)**	**ускорение (раз)**	**название файла**
Исходный код скомпилированный с `-O3`	423165	1	test_1.c
Распараллеливание на 16 потоков с `-O3`	35238	12	test_4.c
Решение задачи на видеокарете спомощью `CUDA`	500	846.33	test.cu
Решение задачи квадратичным алгоритмом	490	863.6	test_6.c

Таким образом квадратичный алгоритм на таком объёмеданных почти сопоставим с вычислениями на видеокарте, но при больших объёмах (при матрицах со сторонами большими 2^{14}) видеокарта будет выигрывать.

Описание распараллеливания алгоритма

По скольку у нас идут приметивные операции с матрицами, их можно разбить на непересекающиеся задачи (по выходным данным) Т.е. у нас даже не будет критических секций в программе. Прирост от 16 потоков примерно 13 раз, что говорит о почти полноценном использовании всего потенциала многопоточности. Но возникает вопрос, почему прирост не в 16 раз? Всё очень просто, у процессора динамически меняется частота, т.е при нагрузке на одно ядро, его частота достигает 4.3Ghz, а при нагрузке на все ядра, всего 4.1Ghz

Описание квадратичного алгоритма

Исходный алгоритм: дано две матрицы исходная `b[i][j]` и выходная `a[i][j]`, соответственно выходная матрица задавалась формулой `a[i][j] = b[i][k] + b[j][k]` для любых `i,j,k = 0....size`. Данная формула по смыслу означает что каждый элемент выходной матрицы равен сумме строк исходной матрицы с номера равными координатам данного элемента. Квадратичный алгоритм: можно создать дополнительный массив, в который записать суммы элементов по строкам исходной матрицы, и в двойном цикле просто посчитать элементы выходной матрицы. Т.е. `c[i] = b[i][0] + ... + b[i][size], a[i][j] = c[i] + c[j];` Тем самым асимптотическая сложность нового алгоритма — $O(n^2)$