Санкт-Петербургский Государственный Университет

Клещин Антон Сергеевич

Отслеживание решений систем дифференциальных уравнений

Научный руководитель:
к.ф.-м.н. Васильев В. А.

Содержание

1 Введение

Работа, которой мы занимаемся уже второй семестр, посвящена так называемой проблеме отслеживания решений — проверке условий, при выполнении которых приближенному решению дифференциального уравнения соответствует точное решение, отличающееся от найденного не более, чем на заданную константу.

Предоставляет подобные условия так называемая теория затенения (shadowing) динамических систем. Эта область только начала активное развитие, первые работы появились в 90-е годы прошлого века. Обзор основных материалов по теме можно найти в статье [1]. В ней же приводятся и обосновываются условия, которые мы проверяем. Чл.-кор. РАН В. А. Плисс сформулировал их еще в 2002 году, но с тех пор полноценная программная проверка так и не появилась.

Однако, появилось первое приближение. В своей кандидатской диссертации [2] наш научный руководитель, В. А. Васильев, ослабил эти условия и написал первую реализацию на MATLAB для систем трех уравнений. Этот результат важен с исследовательской точки зрения — показана практическая применимость данного подхода. Если, например, в первой работе говорится о необходимости существования определенных значений, то во второй уже предлагается, как их лучше выбирать. Но окончательной версией программу на MATLAB назвать тоже нельзя. Во-первых, в реальных задачах системы состоят из гораздо большего числа уравнений (до нескольких десятков). Во-вторых, на длинных промежутках существующая реализация работает дольше, чем хотелось бы.

Таким образом, долговременной целью проекта яаляется реализация алгоритма численной верификации решений систем дифференциальных уравнений, основывающегося на существующих теоретических наработках. При этом на текущий семестр была вынесена задача создания независимого окружения, в котором было бы удобно работать в предметной области основной задачи. Часть окружения, которая требовалась от меня:

- Предоставление высокоуровневой работы с матричными вычислениями;
- Нахождение собственных чисел и векторов.

2 Используемые технологии

Еще в прошлом семестре на этапе планирования с участием научного руководителя было принято решение оптимизировать систему посредством распараллеливания на GPU[1]. Тогда же мы выбирали, какую технологию будем использовать, и остановились на NVIDIA CUDA[3]. Для больших систем самой трудоемкой частью вычислений становится линейная алгебра, и NVIDIA предоставляет cuBLAS[4], свою реализацию интерфейса BLAS[5], являющегося стандартом де-факто в данной области.

Первая проблема, с которой мы столкнулись еще тогда, состоит в том, что кроме основных операций линейной алгебры, не удается найти открытых реализаций нужных нам численных алгоритмов. От готового решения нам нужна была прежде всего удобная поддержка операций линейной алгебры: матричные операции, нахождение собственных чисел, решение линейных систем. На первый взгляд, есть много крупных математических библиотек, на словах поддерживающих оптимизацию на CUDA.

Например, Armadillo[6], которую мы опробовали в прошлом семестре. Ее удается собрать под linux, но технологии NVIDIA гораздо лучше приспособлены к работе под Windows, есть интеграция с Visual Studio и приемлемое окружение. А вот Armadillo под Windows уже собирается с большими ограничениями, конфигурируемость в этом случае совершенно не задокументирована, о подстановке нужной реализации BLAS речи не идет.

Нашим вторым вариантом была библиотека Eigen[8]. Возможность использовать CUDA есть, но опять с ограничениями, в том числе без поддержки Visual Studio[7]. Кроме того, эта библиотека не была написана с учетом специфики работы на GPU, поэтому, даже если подключить cuBLAS, для каждой матричной операции будет производиться копирование с хоста (оперативная память процессора) на девайс (внутренняя память видеокарты), что обесценивает такой подход.

Из крупных библиотек, специально созданных для работы с CUDA, мы также рассматривали ArrayFire[9], но она оказалась огромной, проприетарной, рассчитаной на задачи компьютерного зрения. Поэтому включать ее в проект ради пары численных алгоритмов нецелесообразно.

В итоге, после нескольких неудачных попыток, было принято решение писать собственную обертку над низкоуровневым интерфейсом cuBLAS.

[1]GPU — graphical processing unit, технология программирования графических процессоров, позволяющая массивно распараллеливать выполнение однотипных задач

3 Необходимое окружение

Теперь, раз мы отказались от использования готовых библиотек, нужно было самим реализовать окружение и ряд численных алгоритмов, необходимых для решения задачи.

3.1 Матричные вычисления

Базовый набор для работы с CUDA, предоставляемый NVIDIA, включает в себя две библиотеки, которые легли в основу нашей работы: cuBLAS и cuSOLVER [10]. Они эффективно реализуют в себе множество базовых операций, таких как сложение, умножение, LU-декомпозиция, решение систем линейных уравнений. Проблема безобёрточного их использования заключается в гигантских промежуточных работах. К примеру: функция умножения двух матриц требует 14 параметров, каждый из которых нужно хранить на протяжении долгого времени, передавать в различные функции, выделять и освобождать память для них вручную. Также необходимо выполнять некоторые вспомогательные процедуры типа "подсчёт дополнительной рабочей памяти которую тоже надо предоставлять функциям. Ещё есть алгоритмы, которые разделены на части (вычисление обратной матрицы требует предварительную декомпозицию, которая также используется в решении систем линейных уравнений). Всё это ведёт к сильному разрастанию кода и усложнению написания самих алгоритмов, которые требуют этих матричных вычислений.

В связи с этим, было реализовано три класса матриц, с одной стороны полностью скрывающие все вышеперечисленные действия, с другой стороны сохраняющих высокопроизводительность, а также гибкость исходных функций cuBLAS и cuSOLVER (автоматически используются ленивые вычисления и кэширование, но программист сам должен указать, когда перенести данные матрицы с хоста на девайс и наоборот).

Первый класс реализует в себе все необходимые нам методы и свойства, присущие всем матрицам: сложение, вычитание, умножение на скаляр, доступ к элементу матрицы и другие. Оставшиеся два класса наследуются от первого, расширяя возможности для конкретного вида матриц: вектора или матрицы любой размерности. Первый из них вносит такие возможности как скалярное произведение и линейное отображение в соответствии с переданной матрицей. Второй же может, например, находить собственные числа и вектора, матричную экспоненту, решать линейные уравнения и возвращать всевозможные нормы соответствующих матриц.

3.2 Нахождение собственных чисел и векторов

Существует множество решений данной проблемы, но все они применимы только для матриц специального вида (симметричные, трёхдиагональные, эрмитовы и т.д.). Нам же нужно было находить одновременно все собственные числа и их векторы для матриц общего вида. Для решения этой задачи самым эффективным (и чуть ли не единственным) является QR-алгоритм [11], который, правда, применим для верхней матрицы Хессенберга. Такой вид существует для любой матрицы, и алгоритм приведения к нему также нами реализован. Все статьи, описывающие QR-алгоритм, так или иначе ссылаются на пособие для решения проблем линейной алгебры на Алгол60 [13], на основе алгоритмов которого базируются большинство современных решений, и наше решение не стало исключением. Помимо прочего, наша реализация писалась с учётом соответствующей функции на Fortran в EISPACK [14], который был специально разработан для решения проблем, связанных с собственными числами, и также базировался на всё том же пособии. В итоге удалось повысить точность для некоторых матриц до нескольких порядков.

К примеру:

$$\begin{pmatrix} 2 & -3 & 2 & -1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Собственные числа этой матрицы - две комплексно-сопряжённые пары, вещественные части которых равны 0.5
Алгоритм, приведённый в пособии выдавал 0.4999990842 и 0.500000991538
Результаты Matlab [15]: 0.500000001735324 и 0.499999998264674
Наша текущая версия возвращает 0.500000000089865 и 0.4999999999101347, что на 4 порядка точнее исходной версии и на 2 порядка точнее результатов matlab.

Сравнение работы алгоритма с функцией matlab

Было сгенерировано по тысяче тестов для каждого из порядков матриц (10, 50, 100, 500) по следующему принципу: сначала создаётся матрица, на диагонали которой находятся либо вещественные собственные числа, либо блоки 2x2 комплексно-сопряжённых собственных чисел. Все собственные числа не превосходят по модулю 100. Затем получившаяся матрица сопрягается случайной вещественной. Таким образом становятся известными точные значения собственных чисел, с которыми и производится сравнение.

Для получения собственных чисел в matlab использовалась функция eig.

Были измерены следующие значения:
a) среднее арифметическое модулей погрешностей
b) среднее арифметическое время работы программы (в секундах)
c) среднеквадратичное отклонение времени работы (в секундах)

Matlab			
порядок	a	b	c
10	3.339723e-11	3.720338e-05	1.016465e-06
50	3.215125e-09	1.437581e-03	4.753575e-05
100	1.127971e-08	5.794034e-03	1.115444e-04
500	5.602750e-05	1.920054e-01	2.760415e-04
Наш алгоритм			
порядок	a	b	c
10	2.986188e-11	6.305154e-05	1.157387e-05
50	1.474388e-09	7.678510e-04	4.785714e-06
100	7.301919e-09	4.789964e-03	1.524413e-05
500	7.099496e-07	5.703809e-01	3.848371e-04

3.3 Прочее

Кроме всего вышеперечисленного, необходимое окружение требует еще много мелкой работы, в том числе:

- Углы между подпространствами. Алгоритм поиска углов между линейными подпространствами в $\mathbb{R}^n$, основывающийся на вычислении собственных чисел матрицы специального вида
- Таймер. Неожиданно, с ним оказались проблемы: стандартный таймер C++ замеряет только вермя работы процессора, стандартный из CUDA – видеокарты и, иногда, процессора тоже. Пришлось несколько раз переделывать систему замеров до получения приемлемых результатов

3.4 Оптимизация

Написан собственный алгоритм выделениря памяти, основывающийся на пулах, расширяющихся в два раза в случае нехватки. Интересно, что внедрение пулов прошло безболезненно, так как у нас уже были собственные умные указатели, поэтому сами матричные классы оказались

вообще незатронутыми. Эти оптимизации в совокупности с остальными позволили сократить время работы дискретизации в 2.5-3 раза.

4 Результаты

В результате работы в течение семестра нами было создано все необходимое окружение для реализации алгоритма отслеживания решений линейных систем 3x3. Для требовавшихся численных методов произведено исследование предметной области, написаны и протестированы собственные версии. Произведена оптимизация узких мест системы.

Список литературы

[1] В. А. Плисс, Существование решения дифференциального уравнения, близкого к приближенному решению, Дифференциальные уравнения, 2002, том 38, номер 7, 897–906

[2] В. А. Васильев, Условия существования решения системы дифференциальных уравнений, близкого к приближенному решению, Дифференциальные уравнения, 2011, том 47, номер 3, 310-321.

[3] https://developer.nvidia.com/cuda-zone Доступ 06.06.2018

[4] https://developer.nvidia.com/cublas Доступ 06.06.2018

[5] http://www.netlib.org/blas/ Доступ 06.06.2018

[6] Conrad Sanderson, Ryan Curtin, Armadillo: a template-based C++ library for linear algebra, Journal of Open Source Software, Vol. 1, pp. 26, 2016

[7] https://www.visualstudio.com/ Доступ 06.06.2018

[8] http://eigen.tuxfamily.org/ Доступ 06.06.2018

[9] https://arrayfire.com/ Доступ 06.06.2018

[10] https://developer.nvidia.com/cusolver Доступ 06.06.2018

[11] https://en.wikipedia.org/wiki/QR_algorithm Доступ 06.06.2018

[12] Дж. Деммель, Вычислительная линейная алгебра, Мир, 2001г

[13] J.H.Wilkinson, C.Reinsch, Handbook for Automatic Computation: Volume II: Linear Algebra, Машиностроение, 1976г

[14] http://netlib.sandia.gov/eispack/hqr2.f Доступ 06.06.2018

[15] https://www.mathworks.com/products/matlab.html Доступ 06.06.2018