Понижение размерности методом градиентного спуска с использованием графических ускорителей

А.Н. Борисов[1], Е.В. Мясников[1]

[1]Самарский национальный исследовательский университет им. академика С.П. Королева, Московское шоссе 34А, Самара, Россия, 443086

Аннотация. В статье рассматриваются различные способы реализации алгоритма градиентного спуска, используемого для понижения размерности данных, предназначенные для графических процессоров. Представлены 4 варианта реализации градиентного спуска для графических процессоров, написанные на языке HIP. Удалось достичь 6-кратного улучшения производительности при использовании графического процессора AMD Radeon RX Vega 56 относительно многопоточной реализации, исполняемой на центральном процессоре.

1. Введение

В настоящее время все большую популярность набирает направление, собирательно называемое Data Mining, использующее анализ огромных объемов данных для выявления различного рода значимых закономерностей. Часто эти данные могут быть представлены в виде точек в многомерном пространстве.

Обработка данных больших размерностей представляет собой проблему. Во-первых, для хранения таких данных требуется больше места, во-вторых, время обработки данных в лучшем случае линейно возрастает с ростом размерности. Часто, с точки зрения конкретной задачи, данные большой размерности обладают информационной избыточностью, что позволяет нам перейти в пространство меньшей размерности без существенной потери информационной емкости. Однако данный переход должен быть проведен с максимально возможной точностью.

Наиболее известными алгоритмами понижения размерности являются метод главных компонент [1], метод независимых компонент [2], поиск наилучшей проекции, нелинейные методы CCA [3] и CDA [4]. В данной статье рассматривается метод понижения размерности на основе градиентного спуска, минимизирующего относительную ошибку переноса попарных расстояний между точками, представляющими многомерные данные (ошибку Сэммона) [5]. Данный метод в ряде случев позволяет повысить качество дальнейшего анализа данных вследствие минимальности вносимой ошибки [6].

2. Понижение размерности методом градиентного спуска

Пусть N - число точек, n - размерность исходного пространства, m - размерность целевого пространства.

Нам необходимо перенести точки из n-мерного пространства в m-мерное пространство с минимально возможными искажениями попарных расстояний между точками. Данная задача является задачей нелинейной оптимизации. В качестве минимизируемой функции (целевой функции, критерия) выступает ошибка Сэммона:

$$\epsilon^2 = \frac{1}{\sum_{i,j=1,j>i}^{N} d^n_{i,j}} \sum_{i,j=1,j>i}^{N} \frac{(d^n_{i,j} - d^m_{i,j})^2}{d^n_{i,j}}, \tag{1}$$

где $d^n_{i,j}$ - расстояние между точками с индексами i и j в исходном пространстве,
$d^m_{i,j}$ - расстояние между точками с индексами i и j в целевом пространстве.

В качестве расстояния между точками используется евклидово расстояние.

Одним из наиболее общих методов нелинейной оптимизации является итерационный алгоритм градиентного спуска, общее выражение которого выглядит следующим образом:

$$\mathbf{x}(t+1) = \mathbf{x}(t) - \alpha \nabla f \tag{2}$$

где $\mathbf{x} = \{x_0, ..., x_k\}$ - вектор параметров t,
t - номер итерации,
α - числовой параметр, задающий скорость градиентного спуска,
$\nabla f = \{\frac{\partial f}{\partial x_0}, ..., \frac{\partial f}{\partial x_k}\}$ - градиент целевой функции f при текущих значениях параметров.

Данный алгоритм сходится к минимуму функции f, при условии отсутствия локальных минимумов.

Поскольку целевой функцией является ошибка Сэммона, а вектором параметров выступает совокупность точек, отображенных в целевое пространство, конечное уравнение градиентного спуска примет следующих вид:

$$\mathbf{y}_i(t+1) = \mathbf{y}_i(t) + \mu \sum_{j=1,i\neq j}^{N} \frac{d^n_{i,j} - d^m_{i,j}}{d^n_{i,j} d^m_{i,j}} (\mathbf{y}_i(t) - \mathbf{y}_j(t)), \tag{3}$$

$$\mu = \frac{2\alpha}{\sum_{i,j=1,i\neq j}^{N} d^n_{i,j}}, \tag{4}$$

где $\mathbf{y}_i = \{y^0_i, ...y^{m-1}_i\}$ - координаты точки с индексом i в целевом пространстве,
t - номер итерации,
$d^n_{i,j}$ - расстояние между точками с индексами i и j в исходном пространстве,
$d^m_{i,j}$ - расстояние между точками с индексами i и j в целевом пространстве,
α - параметр, определяющий скорость градиентного спуска.

Очевидно, что сложность одной итерации алгоритма равна $O(N^2(n+m))$. Однако следует заметить, что вычисление обновленных координат для каждой точки на каждой итерации производится независимо от других точек.

3. HIP

HIP [7] - новый инструмент программирования для графических процессоров, разработанный под эгидой инициативы GPUOpen. HIP играет роль прослойки, предоставляющей язык написания программ для графических процессоров и интерфейс взаимодействия с графическими процессорами, дублирующие их аналоги в CUDA (в основном посредством макросов C++). Программы, написанные на CUDA, могут быть перенесены на HIP с относительно небольшими изменениями (AMD предоставляет средство автоматической миграции hipify). В зависимости от используемой в системе платформы, HIP может производить компиляцию либо с помощью HCC (AMD), либо с помощью NVCC (NVIDIA). Таким образом, код, написанный на HIP, может быть скомпилирован как для графических процессоров AMD, так для графических процессоров NVIDIA.

Терминология и программная модель HIP совпадают с терминологией и программной моделью CUDA. Взаимодействие с устройством производится с помощью посыла команд графическому процессору через API. Каждая команда привязывается к определенному потоку команд (stream). Команды в разных потоках команд, по возможности, исполняется параллельно. К командам относятся команды копирования, синхронизации и исполнения функций. Функция, предназначенная для запуска на графическом процессоре, называется функцией ядра. Каждая функция ядра исполняется множеством потоков, или в устоявшемся переводе терминологии CUDA - нитей (thread). Нити объединяются в блоки, блоки образуют сетку блоков. Из доступных программисту типов памяти достаточно отметить глобальную память - основную память графического процессора, и разделяемую память - сверхбыструю память малого объема.

4. Реализация градиентного спуска с помощью HIP

Из вида выражения (3) следует, что обновление координат точки на каждой итерации производится независимо от обновления координат других точек. Следовательно, имеет смысл обработку каждой отдельной точки проводить в отдельном потоке. При этом, чем выше число одновременно исполняемых потоков, тем больше выигрыш производительности.

В таких условиях использование графических процессоров может обеспечить существенный прирост производительности, поскольку одновременно могут исполнять тысячи потоков.

При исполнении алгоритма используются 3 буфера:

(i) буфер, содержащий данные в исходном пространстве (далее обозначен как src);

(ii) буфер, содержащий данные в целевом пространстве, полученные на предыдущей итерации(далее обозначен как $prev$);

(iii) буфер, содержащий данные в целевом пространстве (далее обозначен как $next$);

Основной алгоритм итерации градиентного спуска приведен ниже. Выражение (4) представляет собой константу, и потому выступает параметром алгоритма.

Мы использовали 2 возможных подхода к реализации алгоритма. В первом случае, все точки обрабатывались в строгом соответствии с выражением (3). Во втором подходе мы используем тот факт, что $d_{i,j} = -d_{j,i}$, следовательно, внутренний цикл можно выполнять не для $j \neq i$, а для $j > i$, что уменьшает число итераций с $N(N-1)$ до $\frac{N(N-1)}{2}$, однако в этом случае обновление координат точки должно выполняться атомарно, поскольку несколько потоков могут пытаться обновить координаты точки одновременно. Дополнительно, мы в данном случае переходим к линейной индексации обрабатываемой верхнетреугольной матрицы, для лучшей загрузки вычислительных мощностей.

Дополнительно мы рассмотрели 2 возможных варианта расположения точек в памяти.

Algorithm 1 Итерация градиентного спуска

Input: $src, prev, \mu$
Output: $next$

for $i \leftarrow 0$ to N **do**
    $next[i] \leftarrow prev[i]$
end for
for $i \leftarrow 0$ to N **do**
    for $i \leftarrow 0$ to N **do**
        if $i = j$ **then continue**
        end if
        $d^n_{i,j} \leftarrow distance(src[i], src[j])$
        $d^m_{i,j} \leftarrow distance(prev[i], prev[j])$
        $next[i] \leftarrow next[i] + \mu \frac{d^n_{i,j} - d^m_{i,j}}{d^n_{i,j} d^m_{i,j}} (prev[i] - prev[j])$
    end for
end for

Набор точек, представляющий данные, можно представить в виде двумерного массива или матрицы. В первом случае координаты точек расположены в матрице построчно, что дает возможность одному потоку эффективнее считывать координаты и вычислять расстояние между точками (координаты считываются и обрабатываются группами по 4).

Во втором случае координаты располагаются по столбцам, что теоретически повышает эффективность работы памяти в целом. Соседствующие потоки в этом случае считывают соседствующие данные, что, позволяет вместо инструкций чтения 4 байт использовать инструкции чтения 32 байт или более (memory coalescing) для предоставления данных сразу нескольким потокам.

2 варианта алгоритма итерации и 2 варианта расположения данных дают в итоге 4 возможных варианта, эффективность которых исследуется в дальнейшем.

5. Результаты экспериментов

Для удобства, реализации были пронумерованы следующим образом.

Номер варианта реализации	Описание
1	построчное расположение координат стандартный алгоритм итерации
2	построчное расположение координат, модифицированный алгоритм итерации
3	расположение координат по столбцам, стандартный алгоритм итерации
4	расположение координат по столбцам, модифицированный алгоритм итерации

В качестве исходных данных выступал набор псевдослучайных чисел из интервала [0, 255]. Для каждого выбранного числа блоков, размера блока и числа точек запускались отдельно 4 варианта реализации. В случае, если в результате определенных причин запуск оказывался неудачным(внутренние ограничения HIP, драйвера и аппаратного обеспечения),

набор параметров пропускался. Время копирования данных между оперативной памятью и памятью графического процессора не учитывалось вследствие незначительного вклада в общее время исполнения. Максимальный размер блока нитей установлен в 64, поскольку большее число нитей либо не приводит к улучшению производительности, либо вовсе приводит к ошибке запуска.

Параметры эксперимента:

Число точек, N	1'000, 5'000, 10'000, 50'000, 100'000
Исходная размерность, n:	200
Целевая размерность, m:	3
Параметр скорости спуска, μ	0.5
Число блоков нитей:	1024
Размер блока нитей:	16, 32, 64
Число итераций:	5

Дополнительно в целях сравнения была написана многопоточная реализация градиентного спуска для центрального процессора на основе OpenMP, тождественная реализации 1. При запуске использовались 16 потоков. Компилятор g++ автоматически проводил автовекторизацию циклов с использованием AVX-инструкций.

Эксперименты запускались на следующей аппаратной конфигурации:

ЦП: AMD Ryzen 7 3700X, 8 ядер, 3,6 ГГц;
ГП: AMD Radeon RX Vega 56, 3584 потоковых процессоров, 8 ГБ HBM2;
ОЗУ: 16 ГБ;
ОС: Ubuntu 18.04 LTS.

Результаты экспериментов показаны на рисунках 1 и 2. Наилучшие результаты, соответсвующие им параметры, базовая и итоговая ошибки Сэммона показаны в таблице 1, сравнение производительности центрального и графического процессоров приводится в таблице 2. Сравнение результирующей ошибки градиентного спуска приведено в таблице 3.

Таблица 1. Результаты налучших вариантов реализации для графического процессора

Число точек	Время ЦП (1 поток), мс	Время ЦП (16 потоков), мс	Время ГП, мс	Вариант	Число нитей
1000	142,46	16,33	7,61	2	16
5000	3 556,06	398,08	88,36	1	32
10000	14 251,50	1 497,72	279,33	1	64
50000	370 820,00	38 433,20	6 342,90	2	16
100000	1 462 767,20	152 011,00	25 245,20	2	16

Как показывают результаты экспериментов, для больших объемов данных использование графических процессоров позволяет добиться существенного уменьшения затрачиваемого времени. Наиболее эффективной оказывается реализация 2. Следует отметить, что расположение координат точек по столбцам хотя и дает некоторый эффект, позволяющий увеличить производительность, но тем не менее недостаточный, чтобы превзойти варианты с построчным расположением координат точек.

Данные профилировщика показывают, что основным ограничивающим фактором в данном случае является работа с памятью. Память HBM2, установленная на используемой видеокарте, характеризуется очень большой шириной шины, но низкой частотой и

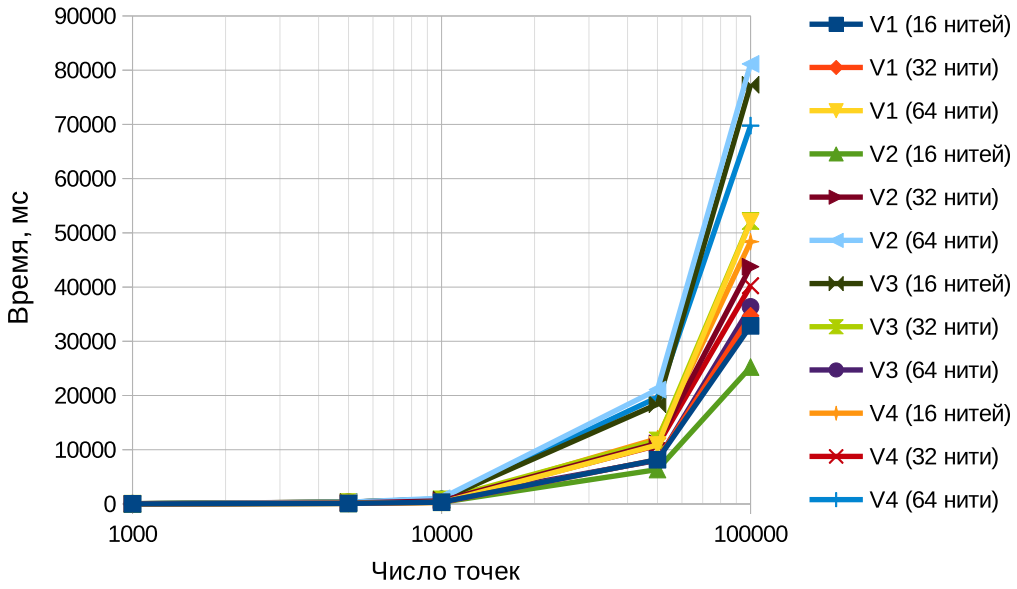

Рисунок 1. Результаты экспериментов для графического процессора

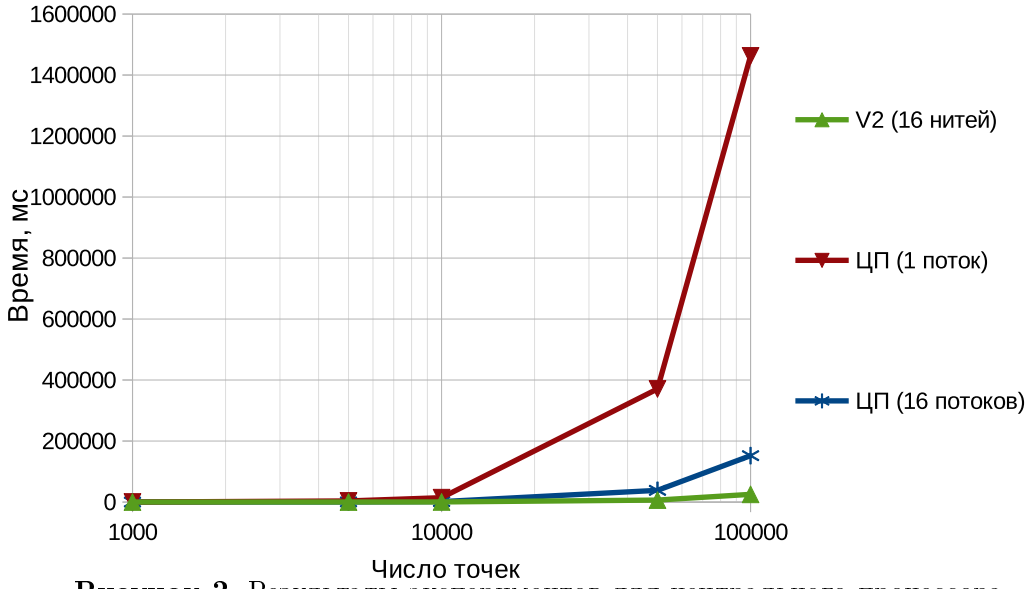

Рисунок 2. Результаты экспериментов для центрального процессора

большими задержками (несколько сотен тактов). Множество одновременных обращений к данным, находящимся в различных участках выделенного буфера (случайный доступ), приводят к перегрузке контроллера памяти и простою вычислительных мощностей в ожидании данных. Использование атомарных операций лишь усугубляет ситуацию. Отчасти, данная ситуация объясняет превосходство размера блока в 16 нитей. Несмотря на то, что загрузка вычислительных мощностей составляет всего 25% (в архитектуре AMD 100% загрузка достигается при размере блока нитей, кратном 64), итоговая нагрузка на контроллер памяти снижается, что позволяет избегать простоев в ожидании данных. Использование устройств с иными типами памяти (GDDR5, GDDR6), возможно, позволит получить совершенно иные результаты.

Таблица 2. Соотношение производительности центрального и графического процессоров

Число точек	Время ЦП (1 поток), мс	Время ГП, мс	Ускорение
1000	142,46	7,61	18,72
5000	3 556,06	88,36	40,25
10000	14 251,50	279,33	51,02
50000	370 820,00	6 342,90	58,46
100000	1 462 767,20	25 245,20	57,94

Таблица 3. Итоговая ошибка

Число точек	Базовая ошибка	Итоговая ошибка, ЦП	Итоговая ошибка, ГП
100000	0,785323	0,110451	0,110452

6. Заключение

В данной статье были рассмотрены различные подходы к реализации алгоритма градиентного спуска для графических процессоров в контексте решении задачи понижения размерности. Были рассмотрены несколько вариантов реализации алгоритма градиентного спуска с помощью языка HIP для графических процессоров AMD. Максимальное преимущество относительно многопоточной версии, исполнявшейся на 8-ядерном центральном процессоре, полученное в ходе экспериментов, составило 6 раз на используемом оборудовании. В дальнейшем планируется проведение экспериментов с участием как графических процессоров AMD других поколений, так и графических процессоров NVIDIA.

7. Благодарности

Работа выполнена при финансовой поддержке РФФИ в рамках научного проекта № 18-07- 01312 а в частях «2. Понижение размерности методом градиентного спска» - «5. Результаты экспериментов» и Министерства науки и высшего образования РФ в рамках госзадания ФНИЦ «Кристаллография и фотоника» РАН В частях «1. Введение» и «6. Заключение».

8. Литература

[1] Andrecut, M. Parallel GPU Implementation of Iterative PCA Algorithms / Journal of computational biology. – 2009. – Vol. 16. – P. 1593-1599.

[2] Yanshan, J. GPU-based Parallel Group ICA for functional magnetic resonance data / J. Yanshan, W. Zeng, N. Wang, T. Ren, Y. Shi, J. Yin, Q. Xu // Computer methods and programs in biomedicine. – 2015. – Vol. 119(1). – P. 9-16.

[3] Demartines, P. Curvilinear Component Analysis: a Self-Organizing Neural Network for Nonlinear Mapping of Data Sets / P. Demartines, J. Herault // Neural Networks, IEEE Transactions on. – 1997. – Vol. 8. – P. 148- 154.

[4] Lee, J. A Robust Nonlinear Projection Method / J.A. Lee, A. Lendasse, N. Donckers, M.Verleysen // Proceeding sof ESANN,8th European Symposium on Artificial Neural Networks - Bruges: D-Factopublic, 2000. – P. 13-20.

[5] Sammon, J. A Nonlinear Mapping for Data Structure Analysis / J.W. Sammon, Jт. // IEEE Transactions on Computers. – 1969. – Vol. C-18(5). – P. 401-409.

[6] Myasnikov, E. Evaluation of Stochastic Gradient Descent Methods for Nonlinear Mapping of Hyperspectral Data / E. Myasnikov // Lecture Notes in Computer Science. – Heidelbetg: Springer, 2016. – Vol. 9730. – P. 276-283.

[7] Sun, Y. Evaluating Performance Tradeoffs on the Radeon Open Compute Platform / Y. Sun, S. Mukherjee, T. Baruah, S. Dong, J. Gutierrez, P. Mohan, D. Kaeli // IEEE International Symposium on Perfotmance Analysis of Systems and Software, 2018. – P. 209-218.

Implementation of "Kuznyechik"encryption algorithm using NVIDIA CUDA

A.N. Borisov[1], E.V. Myasnikov[1]

[1]Samara National Research University, Moskovskoe Shosse 34A, Samara, Russia, 443086

Abstract. In this paper we discuss possible realizations of GPU-targeted gradient descent algorithm used for dimensionality reduction. 4 realizations of gradient descent algorithm were created using HIP - a new framework for GPGPU programming. We got 6 times perfomance improvement over multithreaded CPU version using AMD Radeon RX Vega 56.