АНАЛИЗ ПРИМЕНИМОСТИ OPENCL-РЕАЛИЗАЦИЙ МЕТОДОВ ГАУССА, ХОЛЕЦКОГО И ЗЕЙДЕЛЯ ДЛЯ РЕШЕНИЯ ЛИНЕЙНОЙ ЗАДАЧИ НАИМЕНЬШИХ КВАДРАТОВ

Д.А. Баранов

Оренбургский госуниверситет

Рассматриваются вопросы реализации методов Гаусса, Холецкого и Зейделя с применением технологии OpenCL для задействования графического процессора. Полученные реализации анализируются на применимость к решению линейной задачи наименьших квадратов. Приводятся результаты экспериментов в виде временных затрат и погрешности вычислений.

Введение

Поиск решения переопределённой системы линейных алгебраических уравнений (СЛАУ)

$$A_{mxn}\,x = b\,(m>n) \qquad (1)$$

называют линейной задачей наименьших квадратов, где A_{mxn} – матрица коэффициентов СЛАУ размерности m строк и n столбцов, b – вектор-столбец свободных членов СЛАУ. Её суть заключается в нахождении псевдорешения $\tilde{x}$ системы (1), минимизирующего невязку $\|Ax-b\|$ [1]:

$$\tilde{x} : \|A\tilde{x}-b\|_2 = min_x \|Ax-b\|_2 . \qquad (2)$$

Существует несколько подходов к решению линейной задачи наименьших квадратов:

1. Приведение к нормальной системе уравнений.
2. QR-разложение.
3. SVD-разложение.

В данной работе исследуются OpenCL-реализации методов, основанных на первом подходе.

Приведение СЛАУ (1) к системе нормальных уравнений осуществляется умножением слева обеих частей матричного уравнения на матрицу A^T:

$$A^T Ax = A^T b \ . \qquad (3)$$

Матрица A^TA является квадратной, симметричной и неотрицательно определённой, следовательно, для решения системы (3) применимы такие методы, как схема Холецкого и итерационный метод Гаусса-Зейделя.

Прямые методы обычно не применяют для поиска решений систем размерностью более 100 [2]. Основными преимуществами итерационных методов на системах большой размерности является меньшая трудоёмкость (по сравнению с прямыми методами) и возможность контролировать точность результата (в то время как в случае применения прямых методов неизбежно появляется погрешность из-за ограниченности разрядной сетки). Тем не менее прямые методы, как правило, обладают большим потенциалом параллелизма. Однако проблема точности остаётся, но в задачах, где точность не так важна, как скорость вычислений (такие задачи возникают, в частности, в графосе-

мантическом моделировании), прямые методы решения СЛАУ, реализованные с применением технологии OpenCL, могут оказаться более приемлемыми.

Сначала отметим одну важную особенность стандарта OpenCL: стандарт допускает погрешность округления при выполнении операции деления до 3ulp (Unit at the Last Place), что означает, что при большом количестве таких операций результат может значительно отличаться от аналогичного, полученного на вычислительном устройстве, реализующем стандарт IEEE 754 для чисел с плавающей запятой одинарной точности [3].

Кроме того, при реализации алгоритма на OpenCL целевой платформой, как правило, подразумевается графический ускоритель. Однако память на современных графических ускорителях – крайне дефицитный ресурс, который, к тому же, нельзя наращивать по своему усмотрению, как оперативную память. Так, на большинстве современных графических ускорителей можно обнаружить маркировку «1Gb VRAM», но программисту доступна лишь часть этой памяти (по стандарту OpenCL – минимум ¼ от общего объёма). Данное ограничение не позволяет, например, одновременную работу с двумя матрицами размерности 5792x5792. Поэтому при выборе алгоритма для реализации с использованием OpenCL очень важным фактором является потребность в дополнительной памяти.

Метод Гаусса

Реализация алгоритма метода Гаусса с использованием OpenCL разделена на четыре функции («ядра» в терминологии OpenCL), по два на прямой и обратный ход. На каждом шаге прямого хода вызывается сначала первое, затем второе ядро. Пусть текущий шаг имеет номер *k*, размерность СЛАУ – *n*. В первом ядре прямого хода производится вычисление делителей для каждой строки на текущем шаге:

$$A_{ik}^{k+1} = \frac{A_{ik}^{k}}{A_{kk}^{k}}, i = \overline{k+1, n}. \tag{4}$$

Во втором ядре прямого хода вычисляются новые значения матрицы *A* и вектора *b*:

$$\begin{gathered} A_{ij}^{k+1} = A_{ij}^{k} - A_{kj}^{k} A_{ik}^{k}, i = \overline{k+1, n}, j = \overline{k+1, n}, \\ b_{i}^{k+1} = b_{i}^{k} A_{ik}^{k}, i = \overline{k+1, n}. \end{gathered} \tag{5}$$

Заметим, что формула (5) не обнуляет элементы *k*-го столбца, это необходимо, т.к. все элементы A_{ij}^{k+1} вычисляются параллельно и используют значения *k*-го столбца. Обнуление этого столбца должно производиться после вычисления всех элементов A_{ij}^{k+1}, однако это не обязательно – алгоритм в дальнейшем просто игнорирует эти элементы (т. е. все элементы ниже главной диагонали), считая их нулевыми.

Рассмотрим обратный ход метода Гаусса. На каждом шаге обратного хода так же вызываются два ядра. В первом ядре обратного хода вычисляется x_k :

$$x_k = \frac{b_k^k}{A_{kk}^k}. \tag{6}$$

Очевидно, для первого шага эта формула работает, а для последующих это становится возможно благодаря второму ядру:

$$b_i^{k+1} = b_i^k - x_k A_{ik}^k, i = \overline{1, k-1}. \tag{7}$$

Как видно из (6), вычисленное значение x_k сразу используется для модификации следующих уравнений. Фактически, происходит перенос компонент $x_k A_{ik}$ в правую

часть уравнений, но в левой части они не обнуляются, а просто игнорируются в дальнейшем. Ядро (5), очевидно, неоптимально, поскольку использует всего один процессор, но его использование более оправданно, чем пересылка из памяти графического процессора в оперативную память и обратно.

Метод Холецкого

Данный метод использует симметричность матрицы *A*, что позволяет сократить необходимое число операций вдвое по сравнению с методом Гаусса. Следует отметить, что в данном методе присутствует *n* операций вычисления корня, которые по стандарту OpenCL [3] имеют ошибку округления 3ulp, т. е. равную ошибке операции деления.

Метод Холецкого можно разделить на 3 этапа:

- вычисление матрицы *U*;
- вычисление вектора *y*;
- вычисление вектора *x*.

Каждый из этапов реализуется с помощью двух ядер OpenCL (как и в методе Гаусса, здесь не используется редукция). Все ядра каждого этапа вызываются последовательно для всех строк матрицы. Для хранения промежуточных значений и результата не требуется дополнительная матрица, однако необходим отдельный вектор *D* размерности *N* для хранения диагональной матрицы знаков [4]. Первое ядро первого этапа вычисляет значение вектора d_i и значение матрицы u_{ii}:

$$\begin{aligned} d_i &= sign(a_{ii}), \\ u_{ii} &= \sqrt{a_{ii}}\,. \end{aligned} \tag{8}$$

Здесь и далее *i* означает номер итерации. Очевидно, что первое ядро выполняется для одного элемента, но выгода использования аналогична таковой в методе Гаусса. Так же как и в методе Гаусса, первое ядро возвращает правильный результат для a_{11}, но для его корректной работы на последующих элементах необходимо второе ядро:

$$\begin{aligned} a_{ij} &= \frac{a_{ij} - \sum_{k=1}^{i-1} a_{ki} d_k a_{kj}}{a_{ii} d_i}, j = \overline{i+1, n}, \\ a_{jj} &= a_{jj} - \left(\frac{a_{ij} - \sum_{k=1}^{i-1} a_{ki} d_k a_{kj}}{a_{ii} d_i} \right)^2 d_i \; j = \overline{i+1, n}. \end{aligned} \tag{9}$$

Ядра второго и третьего этапов аналогичны обратному ходу методу Гаусса. Как видно из формулы (9), во втором ядре вычисляется сумма (очевидно, в обеих формулах сумма одна, поэтому она может быть вычислена один раз), вычисление которой реализуется через цикл. Фактически, такой подход не является самым оптимальным, но полностью параллельная реализация вычисления данной суммы требует хранения промежуточных результатов и этапа редукции (фактически, метод Map-Reduce). Оптимальная редукция может быть реализована на OpenCL, но для этого необходим анализ оптимальных размеров рабочих групп (и, возможно, их сечение).

Метод Зейделя

В качестве итерационного метода будет рассмотрен метод Зейделя, поскольку система (3) удовлетворяет условию сходимости этого метода, а малое количество опера-

ций деления (и отсутствие извлечений корня) позволяет свести погрешность, вносимую OpenCL, до минимума.

Поскольку в методе Зейделя (как и в любом итерационном методе) на каждой итерации в вычислениях используется исходная матрица и правая часть (в данном случае $A^T A$ и $A^T b$), для хранения промежуточного результата необходима дополнительная память. Представленная ниже реализация использует дополнительный вектор T размерности N+1 (на 1 больше, чем в методе Холецкого) для хранения значений

$$t_i = \sum_{j=1}^{i-1} a_{ij} x_j^k + \sum_{j=i+1}^{n} a_{ij} x_j^{(k-1)}. \quad (10)$$

Метод Зейделя реализуется через 2 ядра OpenCL, вызываемых последовательно на каждой итерации. На k-ой итерации первое ядро сохраняет предыдущее значение $x_i^{(k-1)}$ в последний элемент вспомогательного вектора T и вычисляет значение x_i^k:

$$\begin{aligned} t_n &= x_i^{(k-1)}, \\ x_i^k &= \frac{t_i}{a_{ii}}. \end{aligned} \quad (11)$$

Во втором ядре производится обновление значений элементов вектора T:

$$t_j = t_j + t_n a_{ji} - x_i a_{ji}, j = \overline{1,n}, j \neq i. \quad (12)$$

Таким образом, использование t_n для хранения значения $x_i^{(k-1)}$ позволяет поддерживать вектор T в актуальном состоянии.

Следует отметить, что матрица $A^T A$ обусловлена хуже A [1]:

$$cond A_2 (A^T A) = cond A_2^2 (A). \quad (13)$$

Как следствие, при подстановке решения, полученного методом Зейделя, в систему (1) ошибка будет больше заданной точности.

На рис. 1 приведён график зависимости времени вычислений описанной реализации метода Зейделя от размерности матрицы $A^T A$, а также результаты для однопоточной CPU-версии.

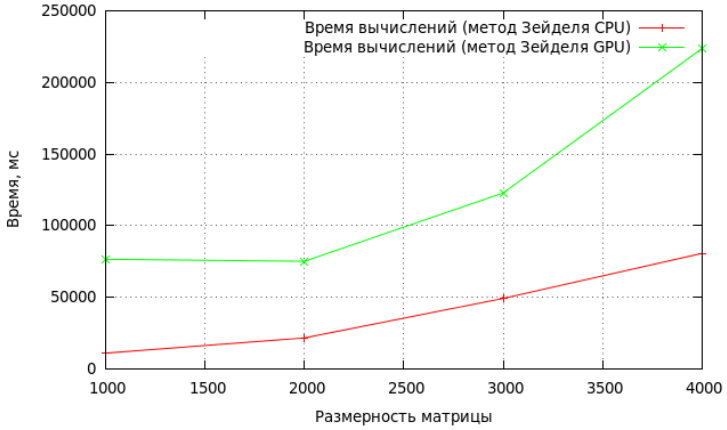

Рис. 1. Зависимость времени вычислений от размерности СЛАУ для метода Зейделя (CPU и GPU)

Как видно из рис. 1, последовательная версия работает быстрее параллельной. Это объясняется иерархической организацией памяти. Иерархия включает «частную» память, локальную и глобальную. Эта архитектура унаследована от графических процессоров, поэтому следует учитывать их особенности. В частности, доступ к глобальной

памяти является медленной операцией, поэтому реализация алгоритмов с большой долей операций с глобальной памятью на OpenCL может дать отрицательный результат, выражающийся в падении производительности по сравнению с последовательной версией.

Сопоставление

На рис. 2 приведены графики зависимости времени вычислений рассмотренных выше реализаций методов решения СЛАУ.

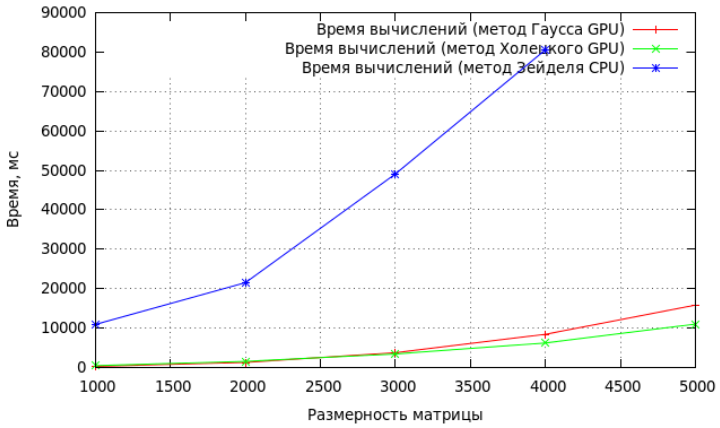

Рис. 2. Зависимость времени вычислений от размерности СЛАУ для методов Гаусса, Холецкого и Зейделя

В табл. 1 приведены значения невязки (2) для каждой из рассмотренных реализаций на каждой из тестовых матриц.

Таблица 1. Невязки (2) для методов Гаусса, Холецкого и Зейделя

N	Метод		
	Гаусса	Холецкого	Зейделя (eps=0.001)
1000	0,000740	0,00070	0,06330
2000	0,001543	0,00170	0,06570
3000	0,020710	0,01990	0,06970
4000	0,012975	0,06520	0,09250
5000	0,037140	0,13590	0,11340

В результате анализа рис. 2 и табл. 1 можно сделать вывод, что реализация метода Гаусса на OpenCL является наиболее оптимальной в задачах, где скорость важнее точности, т. к. она незначительно уступает реализации метода Холецкого на матрицах размерности >3000 и при этом имеет меньшую погрешность. В то же время методу Зейделя требуется на порядок больше времени для достижения сравнимой точности.

Исследование выполнялось при финансовой поддержке Российского гуманитарного научного фонда (проект № 12-04-12034в) и Федеральной целевой программы «Научные и научно-педагогические кадры инновационной России» на 2009–2013 гг. (контракт N 14.B37.21.0176).

Литература

1. Белов С.А., Золотых Н.Ю. Лабораторный практикум по численным методам линейной алгебры. – Нижний Новгород: Изд-во Нижегородского госуниверситета, 2005. – 235 с.
2. Самарский А.А., Гулин А.В. Численные методы. – М.: Наука, 1989. – 432 с.
3. Khronos OpenCL Working Group The OpenCL Specification 1.1., 2011. – [http://www.khronos.org/registry/cl].
4. Богачёв К.Ю. Практикум на ЭВМ. Методы решения линейных систем и нахождения собственных значений. – М.: Изд-во МГУ, 1998. – 137 с.