УДК 519.684.6 DOI: 10.17587/prin.13.94-101

Н. С. Капралов, магистр[1], мл. инженер-программист[2], nskaprl@gmail.com,
А. Ю. Морозов, канд. физ.-мат. наук, ст. препод.[1], науч. сотр.[3], morozov@infway.ru,
С. П. Никулин, канд. физ.-мат. наук, доц.[1], sergeynp@yandex.ru
[1] Федеральное государственное бюджетное образовательное учреждение высшего образования "Московский авиационный институт (национальный исследовательский университет)" (МАИ),
[2] ООО "Технологический центр Дойче Банка", Москва
[3] Федеральный исследовательский центр "Информатика и управление" Российской академии наук (ФИЦ ИУ РАН), Москва

Параллельная аппроксимация многомерных тензоров с использованием графических процессоров

При решении многих исследовательских задач прикладного характера возникает необходимость работать с многомерными массивами (тензорами). На практике используется эффективное и компактное представление данных объектов в виде так называемых тензорных поездов. Рассматривается параллельная реализация алгоритма TT-cross, который позволяет получить разложение многомерного массива в тензорный поезд, с использованием графического процессора архитектуры CUDA. Представлены основные аспекты и особенности выполнения параллельной реализации алгоритма. На ряде примеров проведены апробации полученной параллельной версии алгоритма. Продемонстрировано существенное сокращение вычислительного времени по сравнению с аналогичной последовательной реализацией алгоритма, что свидетельствует об эффективности предлагаемых подходов к распараллеливанию.

Ключевые слова: *распараллеливание, тензорный поезд, тензорные разложения, большие размерности, проклятие размерности, многомерные массивы, крестовая аппроксимация, TT-cross, maxvol, малоранговая аппроксимация, CUDA, GPU, Nvidia*

Введение

В настоящее время существует много задач, для решения которых требуются высокопроизводительные ресурсы. Рассмотрим в качестве примера моделирование поведения объекта, зависящего от большого числа параметров, значения которых нельзя точно указать, но можно привести интервалы, в которых они находятся. Для решения таких задач был разработан [1], теоретически обоснован [1, 2], апробирован на прикладных задачах [3] и программно реализован [4, 5] алгоритм адаптивной интерполяции. В процессе работы алгоритма над множеством, образованным интервальными параметрами задачи, строится интерполяционная сетка, в которой число узлов экспоненциально зависит от числа параметров. В результате алгоритм имеет экспоненциальную сложность относительно числа интервальных неопределенностей как по вычислительным затратам, так и по необходимому объему памяти [3—5]. Также отметим, что зачастую моделируемый объект может изначально подразумевать использование некоторой пространственной сетки, что приводит к дополнительному увеличению общей вычислительной сложности.

Решение подобных задач, подразумевающих работу с многомерными структурами, требует очень больших вычислительных ресурсов. Одной из ключевых подзадач, которую можно выделить, является подзадача эффективного хранения и взаимодействия с многомерными массивами — тензорами (данная терминология используется в соответствии с работами [6, 7]). В результате такого подхода к решению рассматриваемой подзадачи хотелось бы, имея некоторое подмножество элементов с меньшей размерностью, уметь вычислять остальные значения многомерного массива. Если рассматривать тензор как функцию многих переменных, то необходимо провести разделение переменных. С помощью произведения функций одной переменной это получится крайне неточно. Однако если представить исходную функцию с помощью суммы таких произведений, то это уже позволит получить в некоторой мере желаемый результат и представление будет иметь $O(dnr)$ элементов, где d — размерность; r — ранг тензора (в данном случае многомерного массива); n — число значений параметров. Представление тензора в таком виде с минимальным r носит название канонического разложения [8].

К сожалению, нет методов, гарантирующих нахождение такого разложения.

Рассмотрим формат, называемый тензорным поездом (TT-формат) [6]. TT-формат или, по-другому, TT-разложение представляется в виде набора $(d-2)$ трехмерных тензоров и двух матриц. Вычисление любого элемента исходного тензора сводится к перемножению вектора-строки на цепочку матриц и на вектор-столбец. К преимуществам данного формата относятся: возможность получения искомого разложения без вычисления всех элементов исходного тензора; доступность всех арифметических (и не только) операций над тензорами в рамках этого представления.

Построение TT-разложения сводится к обычным матричным разложениям. Выполняется переход от d измерений к двум с помощью группировки индексов. Для полученной матрицы строится сингулярное разложение (SVD-разложение) [9]. Строки и столбцы, соответствующие несущественным сингулярным числам, отбрасывают. Урезанные таким способом матрицы разложения превращаются обратно в тензоры меньшей размерности, для которых выполняются все те же самые действия (тензоры превращаются в матрицы, строится SVD-разложение и т. д). В результате получается набор из трехмерных тензоров. Описанный выше подход лежит в основе алгоритма TT-SVD. Идея алгоритма TT-cross [7], позволяющего построить TT-разложение без вычисления всех элементов тензора, заключается в замене SVD-разложения на разложение с меньшей алгоритмической сложностью и не требующего полного знания всех элементов тензора. Здесь применяются подходы [10—13], позволяющие уменьшить вычислительную сложность задач за счет выявления скрытых структур и устранения избыточности, в частности, крестовая аппроксимация.

При решении прикладных задач получаемые тензоры могут иметь сложную структуру, и применение алгоритма TT-cross в общем случае будет связано с большими вычислительными затратами. В связи с этим обстоятельством существует необходимость в параллельной программной реализации данного алгоритма на высокопроизводительной вычислительной установке (системе).

Наиболее доступными и высокопроизводительными системами на настоящее время являются такие, в основе которых находятся графические процессоры (GPU). Графический процессор состоит из большого числа специализированных ядер и в общей сложности позволяет выполнять намного больше операций в единицу времени по сравнению с центральным процессором. Использование этого преимущества накладывает некоторые ограничения, а именно — алгоритм должен иметь возможность выполняться в параллельном варианте, т. е. необходима возможность часть операций проводить независимо друг от друга, иначе ядра будут ждать получения промежуточных результатов, и это может не только не дать желаемого ускорения, но и значительно замедлить выполнение программы. Поэтому необходимо выполнить детальный анализ алгоритма в целях поиска мест, которые могут быть распараллелены.

Реализация алгоритма выполняется с использованием технологий CUDA [14] компании Nvidia. Технология CUDA реализуется в виде надстройки над языком C [15], и для компиляции ее программ используется специальный компилятор, поставляемый производителем. Удобство состоит в том, что при знании языка C гораздо проще использовать CUDA. Кроме того, компилятор поддерживает язык C++ на центральном процессоре и позволяет использовать его стандартную библиотеку [16]. В идеале при работе графического процессора центральный процессор не простаивает и готовит данные для последующей их обработки. За счет такого подхода при грамотной реализации, для которой необходимо знать детали работы графического процессора, можно сократить временны́е затраты во много раз. В настоящее время графический процессор имеется в каждом среднем компьютере, соответственно, реализация алгоритма построения TT-разложения с использованием технологии CUDA может быть широко использована общественностью.

Алгоритм TT-cross

Приведем формальное описание TT-формата. Рассматривается d-мерный тензор $\mathbf{A} \in \mathbb{R}^{n_1 \times n_2 \times \ldots \times n_d}$. Получение произвольного элемента с индексами $i_1, i_2, \ldots, i_d$ исходного тензора, представленного в TT-формате, выглядит следующим образом:

$$\mathbf{A}(i_1, i_2, \ldots, i_d) = \sum_{\alpha_1=1}^{r_1} \ldots \sum_{\alpha_{d-1}=1}^{r_{d-1}} \mathbf{G}_1(i_1, \alpha_1)\mathbf{G}_2(\alpha_1, i_2, \alpha_2)\ldots$$

$$\ldots \mathbf{G}_{d-1}(\alpha_{d-2}, i_{d-1}, \alpha_{d-1})\mathbf{G}_d(\alpha_{d-1}, i_d),$$

где $\mathbf{G}_1$, $\mathbf{G}_d$ — матрицы размером $n_1 \times r_1$ и $r_{d-1} \times n_d$ соответственно; а $\mathbf{G}_k$, $2 \leqslant k \leqslant (d-1)$ — трехмерные тензоры размером $r_{k-1} \times n_k \times r_k$.

Вычисление $\mathbf{A}(i_1, i_2, \ldots, i_d)$ по сути является перемножением вектора-строки на цепочку матриц и на вектор-столбец. Отметим, что у векторно-матричных операций зачастую высокая степень распараллеливания, и существует ряд готовых библиотек, их реализующих [17—19].

В данном контексте r_k называют рангами аппроксимации, а $\mathbf{G}_k$ — ядрами разложения или вагонами. Именно схожесть формата с поездом из-за сцепки ядер друг с другом индексами суммирования α_k повлекла за собой появление названия "тензорный поезд".

Для краткости далее при работе с тензорами или матрицами будет использована нотация MATLAB [20]. Например, запись $\mathbf{A}(i,:)$ означает взятие строк i, т. е. по первой размерности берется только индекс i, а по второй — все индексы. Заглавные буквы в индексах означают некоторое множество индексов. Диапазон значений от 1 до n будет обозначаться как $1{:}n$. Функция *reshape* — изменение размера матрицы (например, преобразование матрицы размером 3×4 к 6×2).

Разверткой тензора $\mathbf{A}_k$ будем называть матрицу, полученную в результате группировки первых k индексов как строчных, а остальных $d-k$ индексов как столбцовых:

$$\mathbf{A}_k = \mathbf{A}\left([i_1, i_2, ..., i_k], [i_{k+1}, i_{k+2}, ..., i_d]\right).$$

Таким образом, из тензора размером $n_1 \times n_2 \times ... \times n_d$ получается матрица размером $(n_1 n_2 ... n_k) \times (n_{k+1} n_{k+2} ... n_d)$. Приведя тензор к такому виду, можно удобно хранить значения в памяти компьютера и использовать все традиционные матричные методы. Отметим, что доказано существование TT-разложения с рангами r_k, равными рангу соответствующих матриц развертки $\mathbf{A}_k$ [21].

Описание всех алгоритмов приводится в соответствии с работами [21, 22]. Основная идея TT-cross заключается в последовательном применении матричного разложения к разверткам тензора и выборке наиболее важных элементов, по которым в дальнейшем можно будет восстановить все остальные.

Входные данные алгоритма: тензор **A** размером $n_1 \times n_2 \times ... \times n_d$, ранги аппроксимации $r_1, r_2, ..., r_{d-1}$. Результат выполнения алгоритма — ядра разложения $\mathbf{G}_1, \mathbf{G}_2, ..., \mathbf{G}_d$:

1) $N = \prod_{k=2}^{d} n_k$;

2) $\mathbf{A}_1 = reshape\left(\mathbf{A}, [n_1, N]\right)$ // первая развертка тензора **A**;

3) $\mathbf{Q}\widehat{\mathbf{Q}}^{-1}\mathbf{R} = crossApproximation\left(\mathbf{A}_1, r_1\right)$ // крестовая аппроксимация матрицы-развертки $\mathbf{A}_1$, где **Q** — матрица размера $n_1 \times r_1$; $\widehat{\mathbf{Q}}$ — матрица размера $r_1 \times r_1$; **R** — матрица размера $r_1 \times N$, составленная из r_1 строк матрицы $\mathbf{A}_1$;

4) $\mathbf{G}_1 = \mathbf{Q}\widehat{\mathbf{Q}}^{-1}$ // первое ядро разложения;

5) for $k = 2$ to $(d - 1)$ do // цикл по всем размерностям;

6) $N = \frac{N}{n_k}$;

7) $\mathbf{A}_k = reshape\left(\mathbf{R}, [r_{k-1} n_k, N]\right)$ // i-я развертка оставшейся части тензора;

8) $\mathbf{Q}\widehat{\mathbf{Q}}^{-1}\mathbf{R} = crossApproximation\left(\mathbf{A}_k, r_k\right)$ // крестовая аппроксимация матрицы-развертки $\mathbf{A}_k$, где **Q** — матрица размера $r_{k-1} n_k \times r_k$; $\widehat{\mathbf{Q}}$ — матрица размера $r_k \times r_k$ и **R** — матрица размера $r_k \times N$;

9) $\mathbf{G}_k = reshape\left(\mathbf{Q}\widehat{\mathbf{Q}}^{-1}, [r_{k-1}, n_k, r_k]\right)$ // преобразовать матрицу $\mathbf{Q}\widehat{\mathbf{Q}}^{-1}$ в тензор и приравнять k-е ядро разложения к нему;

10) end for;

11) $\mathbf{G}_d = \mathbf{R}$.

Важным свойством данного алгоритма является тот факт, что он имеет фиксированные ранги аппроксимации, которые в зависимости от ситуации можно увеличивать или уменьшать. Для хранения тензора в таком формате необходимо $O(dnr^2)$ элементов. Для приведения тензора в формат тензорного поезда необходимо выполнить $O(dnr^3 + n^d)$ операций, что составляет линейную сложность относительно числа элементов тензора.

Ключевым моментом в TT-cross является алгоритм крестовой аппроксимации *crossApproximation* [7]. Он позволяет построить компактное представление матрицы за счет выявления скрытых зависимостей с использованием при этом только части элементов исходной матрицы. По сути, *crossApproximation* выполняет сжатие данных (с потерями, если ранг матрицы превышает ранг аппроксимации). Входные данные алгоритма: матрица **A** размером $m \times n$, ранг аппроксимации r, параметр остановки δ. Результат выполнения алгоритма: разложение матрицы $\mathbf{Q}\widehat{\mathbf{Q}}^{-1}\mathbf{R} \approx \mathbf{A}$, где **Q** — матрица размера $m \times r$, полученная с помощью *QR*-разложения матрицы, составленной из r столбцов матрицы **A**; $\widehat{\mathbf{Q}}$ — матрица размера $r \times r$, составленная из r строк матрицы **Q**; **R** — матрица размера $r \times n$, составленная из r строк матрицы **A**:

1) $\mathbf{I} = random\left([1, m], r\right)$ // r случайных строчных индексов;

2) $\mathbf{A}_0 = zeros\left(m, n\right)$ //$zeros\left(m, n\right)$ — матрица с нулевыми элементами размером $m \times n$;

3) $k = 1$;

4) do;

5) $\mathbf{R} = \mathbf{A}(\mathbf{I},:)$// строки с индексами **I**;

6) $\mathbf{R} = \mathbf{R}^{\mathrm{T}}$;

7) $\mathbf{QT} = QR\left(\mathbf{R}\right)$ // *QR*-разложение матрицы **R**;

8) $\mathbf{J} = maxvol(\mathbf{Q})$ // поиск индексов подматрицы максимального объема;

9) $\mathbf{C} = \mathbf{A}(:,\mathbf{J})$ // столбцы с индексами **J**;

10) $\mathbf{QT} = QR(\mathbf{C})$;

11) $\mathbf{I} = maxvol(\mathbf{Q})$;

12) $\widehat{\mathbf{Q}} = \mathbf{Q}(\mathbf{I}:)$;

13) $\mathbf{A}_k = \mathbf{Q}\widehat{\mathbf{Q}}^{-1}\mathbf{A}\left(\mathbf{I},:\right)$ // k-е приближение;

14) $k = k + 1$;

15) while$\left(\left\|\mathbf{A}_k - \mathbf{A}_{k-1}\right\| > \delta\left\|\mathbf{A}_k\right\|\right)$;

16) $\mathbf{R} = \mathbf{A}(\mathbf{I},:)$.

Алгоритм *maxvol* используется для нахождения индексов подматрицы максимального объема и является основным вспомогательным алгоритмом, используемым в ходе работы крестовой аппроксимации. На каждой итерации *crossApproximation* рассматривается подматрица, состоящая из r строк исходной матрицы, для которой определяются номера r столбцов, содержащие подматрицу максимального объема. Далее берется подматрица матрицы **A**, состоящая уже из r соответствующих столбцов, и в ней определяются номера r строк, содержащих подматрицу максимального объема. Алгоритм завершается, когда полученные на двух подряд идущих итерациях аппроксимации исходной матрицы отличаются меньше, чем на некоторое значение, характеризующееся параметром остановки δ.

Подматрица максимального объема — это подматрица, имеющая максимальный по модулю определитель. На практике обычно ищется такая подматрица $\widehat{\mathbf{C}}$ матрицы **C** $(m \times r, m > r)$, что элементы матрицы $\mathbf{C}\widehat{\mathbf{C}}^{-1}$ по модулю не превосходят 1 [22].

Входные данные алгоритма *maxvol*: матрица **A** размером $m \times r$, $m > r$. Результат выполнения алгоритма: массив размером r, содержащий номера строк матрицы **A**, составляющих подматрицу максимального объема:

1) $\mathbf{I} = 1 : m$ // массив, который отображает индекс строки в исходной матрице;

2) for $i = 1$ to r do // выбрать случайные строки в исходной матрице и передвинуть их в верхний блок;

3) $ind = random\ (2, m)$ // случайный индекс;

4) $swapRows\left(\mathbf{A}, i, ind\right)$// поменять местами строки матрицы;

5) $swap\left(\mathbf{I}, i, ind\right)$// поменять местами значения в массиве индексов;

6) end for;
7) $\hat{\mathbf{A}} = getRows(\mathbf{A}, [1:r])$ // взять первые r строк;
8) $\mathbf{B} = \mathbf{A}\hat{\mathbf{A}}^{-1}$;
9) do;
10) $i, j = argmax(\mathbf{B})$ // индексы максимального по модулю элемента в $\mathbf{B}$;
11) $B_{\max} = |\mathbf{B}(i, j)|$;
12) if $B_{\max} > 1$ then;
13) $\mathbf{B} = \mathbf{B} - \frac{1}{\mathbf{B}(i, j)}\left(\mathbf{B}(:, j) - \mathbf{e}_j + \mathbf{e}_i\right)\left(\mathbf{B}(i,:) - \mathbf{e}_j^{\mathrm{T}}\right)$;
14) $swap(\mathbf{I}, i, j)$;
15) end if;
16) while$(B_{\max} > 1 + \varepsilon)$;
17) $resultIndices = \mathbf{I}(1:r)$ // индексы строк изначальной матрицы $\mathbf{A}$, которые составляют подматрицу максимального объема.

Общая вычислительная сложность *maxvol* $O\left(r^3 + 3rk(n-r)\right)$, где k — число итераций алгоритма. Параметр ε — небольшое число (обычно $\varepsilon = 0{,}02$), позволяющее ускорить сходимость алгоритма (получить дополнительный прирост производительности).

В данном разделе рассмотрен алгоритм TT-cross, в основе которого лежат алгоритм крестовой аппроксимации и алгоритм *maxvol*. Описанные алгоритмы построены на операциях с векторами и матрицами и, следовательно, имеют потенциал для параллельной реализации, а также использования возможностей технологии CUDA.

Распараллеливание и реализация

Выполнена реализация алгоритма TT-cross в двух вариантах — последовательном и параллельном. В TT-cross используются операции с векторами и матрицами, их реализация была взята из готовых библиотек.

В последовательной реализации была использована библиотека LAPACKE, которая представляет собой интерфейс к открытой библиотеке LAPACK [23] на языке Fortran. Данная библиотека является широко распространенной библиотекой алгоритмов линейной алгебры. За долгое время ее существования она была качественно отлажена и оптимизирована.

Для реализации, выполненной на графическом процессоре, была использована библиотека MAGMA [17]. Данная библиотека является аналогом LAPACK для графических процессоров и предоставляет реализацию алгоритмов линейной алгебры уже в параллельном варианте. Стоит принимать во внимание, что обычные графические процессоры изначально предназначены для работы с числами одинарной точности, так как для решения задач компьютерной графики большая точность не требуется. Однако численные методы чаще всего подразумевают использование чисел двойной точности. Отметим, что для работы с числами повышенной точности компании выпускают специальные научные графические процессоры, которые оптимизированы специально для этого. Были проведены сравнительные тесты производительности [24], которые показали, что MAGMA при работе с 64-битными вещественными числами не уступает по производительности библиотеке cuBLAS [18] от разработчиков Nvidia, при этом имеет более широкий функционал, поэтому именно она была выбрана к использованию.

Все многомерные объекты в памяти хранятся в линеаризованном виде. Например, матрицы хранятся в массиве по столбцам. Это означает, что элемент $\mathbf{A}(i, j)$ в одномерном массиве будет иметь индекс $z = i + jm$, где $\mathbf{A}$ — матрица размером $l \times m$. Аналогично для трехмерного тензора $l \times m \times n$ индекс для элемента $\mathbf{A}(i, j, k)$ будет преобразован в линейный индекс $z = i + jm + kmn$. Нетрудно заметить, что организация многопоточной программы для графического процессора с выбором номера потока имеет схожие принципы, поэтому данный метод линеаризации хорошо ложится на GPU. Следует отметить, что при данном способе хранения операция *reshape*, которая используется в алгоритме TT-cross для получения следующей развертки тензора, выполняется за константное время и не требует перестановки элементов тензора.

С использованием описанных выше подходов была выполнена реализация алгоритмов TT-cross, крестовой аппроксимации и *maxvol* на языке программирования C++ с использованием технологии CUDA.

Результаты

Каждая часть алгоритма TT-cross тестировалась отдельно: сначала алгоритм *maxvol*, далее крестовая аппроксимация, а в заключение весь алгоритм TT-cross. Характеристики CPU: Intel Core i5-9400F, ОП: 48 Гбайт, 3200 МГц; GPU: GeForce GTX 1060 6 Гбайт.

Апробация реализаций алгоритма *maxvol* выполнялась на нескольких матрицах, сгенерированных случайным образом. В табл. 1 представлены замеры времени работы.

Таблица 1

Результаты работы алгоритма *maxvol*

Результаты	Размер матрицы, $m \times n$			
	500×3000	5000×500	10000×300	10000×500
Время CPU, мс	12 589	31 622	25 218	60 195
Время GPU, мс	316	501	398	630
Ускорение, раз	39,8	63,1	63,4	95,5

По данным табл. 1 видно, что за счет использования GPU удалось сократить расчетное время более чем в 95 раз. Самой трудозатратой частью алгоритма является обращение верхнего блока матрицы, а данная операция имеет хороший потенциал для распараллеливания, что позволяет получать отличные результаты.

Реализации крестовой аппроксимации тестировались на матрицах определенного ранга. Для того чтобы получить матрицу $m \times n$ рангом $\leqslant r$, следует перемножить матрицы $m \times r$ и $r \times n$ ранга r. Алгоритм корректно работает, если при ранге аппроксимации, большем или равном r, матрица будет в точности восстановлена из полученного разложения, причем аппроксимация рангом более чем r должна находиться корректно. Следует отметить, что при ранге аппроксимации меньшем r должны возникать погрешности.

Сначала выполняется апробация на матрице размером 10000×10000 ранга 1000. В табл. 2 представлены замеры времени при различном значении задаваемого ранга аппроксимации.

Максимальное ускорение в 40 раз достигается, когда заданный ранг совпадает с фактическим рангом матрицы. Время нахождения аппроксимации сокращается примерно с 4 мин до 6 с, что является очень значительным приростом. Качество восстановления матрицы при этом остается неизменным. Поэлементные разности между исходными и восстановленными матрицами, полученными в результате работы обеих реализаций, составляют около 10^{-16}, что граничит с машинной точностью для 64-битных вещественных чисел. А Евклидова норма разности исходной и восстановленной матриц составила около 10^{-9}.

Далее выполняется тестирование на матрицах разного размера с одинаковым рангом — 95 (табл. 3).

Для матрицы размером 100 × 100 реализация на CPU оказалась быстрее, так как подготовка запуска вычислений на GPU требует дополнительного времени, сопоставимого со временем самих вычислений. Однако при увеличении размера матрицы ускорение возрастает до 25 раз, хотя ранг является довольно малым относительно размера матрицы.

Таблица 2

Результаты работы крестовой аппроксимации на матрице 10000×10000 ранга 1000

Результаты	Ранг аппроксимации		
	10	100	1000
Время CPU, мс	9347	21 032	240 763
Время GPU, мс	310	781	6046
Ускорение, раз	30,2	26,9	39,8

Таблица 3

Результаты крестовой аппроксимации на матрицах разного размера

Результаты	Размер матрицы $m \times n$		
	100×100	1000×1000	10000×10000
Время CPU, мс	9	488	19 640
Время GPU, мс	32	126	740
Ускорение, раз	0,3	3,9	26,5

Алгоритм TT-cross тестировался на нескольких тензорах, заданных в виде функций. Ранги аппроксимации, участвующие в алгоритме, подбирались так, чтобы по построенному тензорному поезду можно было в точности восстановить все элементы исходного тензора.

Элементы тензора Гильберта [7] задаются следующим образом:

$$\mathbf{A}(i_1, i_2, \ldots, i_d) = \frac{1}{i_1 + i_2 + \ldots + i_d}, \quad 1 \leqslant i_j \leqslant n, \quad 1 \leqslant j \leqslant d.$$

В табл. 4 приведены временные затраты на построение тензорного поезда и восстановление тензора при различных значениях n с фиксированной размерностью $d = 5$. Ранги для данного тензора

Таблица 4

Результаты разложения и восстановления тензора Гильберта $d = 5$

Операция	Результаты	n		
		10	20	40
Построение тензорного поезда	Время CPU, мс	47	866	22 674
	Время GPU, мс	150	222	2415
	Ускорение, раз	0,3	3,9	9,4
Восстановление элементов тензора	Время CPU, мс	4	107	3648
	Время GPU, мс	2	4	35
	Ускорение, раз	2	26,8	104,2

равны 5, и при построении тензорного поезда они задавались соответствующими. Здесь наблюдаются десятикратное ускорение при построении тензорного поезда и стократное ускорение при его восстановлении. Получение такого большого ускорения связано с тем обстоятельством, что восстановление состоит исключительно из операции перемножения матрицы на вектор, которая обладает высокой степенью параллелизма.

Исследуем, как изменяется время работы при разных рангах аппроксимации на тензоре размером $80 \times 80 \times 80 \times 80$ (табл. 5).

Здесь ускорение при построении тензорного поезда составляет около 17 раз, а при восстановлении тензора — 70—80 раз. Данный расчет показал, что увеличение ранга аппроксимации приводит к увеличению вычислительного времени. Это связано с тем, что возрастают размеры соответствующих матриц, получающихся в процессе работы алгоритма.

Далее рассмотрим тензор, элементы которого задаются с помощью функции Гривонка [25]:

$$\mathbf{A}(i_1, i_2, ..., i_d) = 1 + \frac{1}{4000}\sum_{k=1}^{d} i_k^2 - \prod_{k=1}^{d} \cos\left(\frac{i_k}{\sqrt{k}}\right),$$
$$1 \leqslant i_j \leqslant n,\ 1 \leqslant j \leqslant d.$$

В отличие от тензора Гильберта, здесь есть тяжелые в вычислительном плане функции, такие как cos и $\sqrt{}$.

Для восстановления тензора с абсолютной точностью 10^{-9} достаточно использовать ранги аппроксимации, равные 3. В табл. 6 показаны замеры времени на тензорах разной размерности при $n = 75$.

При размерности тензора больше 3 параллельная реализация TT-cross работает в 10 и более раз быстрее прямого вычисления всех элементов и последовательной реализации. Необходимо отметить, что суммарное время на построение тензорного поезда и восстановление всех его элементов меньше, чем время на прямое вычисление всех элементов тензора, что еще раз дополнительно подтверждает эффективность тензорных поездов. Кроме этого, для хранения элементов пятимерного тензора с двойной точностью (8 байт на элемент) потребовалось бы 18 Гбайт памяти, что достаточно существенно. При этом отметим, что построение тензорного поезда успешно выполнилось на видеокарте с 6 Гбайт графической памяти.

Приведенные в данном разделе результаты демонстрируют эффективность параллельной реализации алгоритмов TT-cross, крестовой аппроксимации и *maxvol*.

Таблица 5

Временны́е затраты на разложение и восстановление тензора Гильберта размером $80 \times 80 \times 80 \times 80$ при разных значениях ранга

Операция	Результаты	Ранг аппроксимации			
		5	10	20	40
Построение тензорного поезда	Время CPU, мс	7963	10 454	19 822	51 097
	Время GPU, мс	450	601	1142	2979
	Ускорение, раз	17,7	17,4	17,4	17,2
Восстановление элементов тензора	Время CPU, мс	1269	1613	2566	4822
	Время GPU, мс	16	23	32	62
	Ускорение, раз	79,3	70,1	80,2	77,8

Таблица 6

Временны́е затраты на разложение и восстановление тензора, полученного с помощью функции Гривонка

Операция	Результаты	Размерность, d		
		3	4	5
Построение тензорного поезда	Время CPU, мс	65	5755	220 923
	Время GPU, мс	71	249	11 628
	Ускорение, раз	0,9	23,1	19
Восстановление элементов тензора	Время CPU, мс	7	873	31 622
	Время GPU, мс	1	13	502
	Ускорение, раз	7	67,2	63
Прямое вычисление всех элементов тензора, CPU, мс		44	4086	350 603

Заключение

Необходимость работать с многомерными данными возникает во многих современных областях. Формат тензорного поезда является эффективным форматом представления многомерных данных. В работе рассматриваются вопросы распараллеливания алгоритма TT-cross, который позволяет строить разложение в тензорный поезд без вычисления всех элементов исходного многомерного тензора, на графических процессорах. Алгоритм TT-cross включает в себя алгоритм крестовой аппроксимации и алгоритм *maxvol*. Каждый из алгоритмов был реализован с использованием технологии CUDA и библиотеки параллельных матрично-векторных операций MAGMA. Выполнено сравнение параллельных реализаций алгоритмов с последовательными реализациями. По результатам, полученным на представительном наборе тестовых примеров, можно сделать следующие выводы: алгоритм *maxvol* ускорился в 90 раз; алгоритм крестовой аппроксимации — в 40 раз; алгоритм TT-cross — в 20 раз. Отметим, что вычисления выполнялись с использованием обычной видеокарты прошлого поколения. Полученные результаты демонстрируют эффективность разработанных параллельных реализаций соответствующих алгоритмов.

Список литературы

1. **Морозов А. Ю., Ревизников Д. Л.** Алгоритм адаптивной интерполяции на основе kd-дерева для численного интегрирования систем ОДУ с интервальными начальными условиями // Дифференциальные уравнения. 2018. Т. 54, № 7. С. 963—974. DOI: 10.1134/S0374064118070130.

2. **Morozov A. Yu., Reviznikov D. L.** Modelling of Dynamic Systems with Interval Parameters on Graphic Processors // Программная инженерия. 2019. Vol. 10, No. 2. P. 69—76. DOI: 10.17587/prin.10.69-76.

3. **Морозов А. Ю.** Параллельный алгоритм адаптивной интерполяции на основе разреженных сеток для моделирования динамических систем с интервальными параметрами // Программная инженерия. 2021. Т. 12, № 8. С. 395—403. DOI: 10.17587/prin.12.395-403.

4. **Морозов А. Ю., Журавлев А. А., Ревизников Д. Л.** Анализ и оптимизация алгоритма адаптивной интерполяции численного решения систем обыкновенных дифференциальных уравнений с интервальными параметрами // Дифференциальные уравнения. 2020. Т. 56, № 7. С. 960—974. DOI: 10.1134/S0374064120070122.

5. **Гидаспов В. Ю., Морозов А. Ю., Ревизников Д. Л.** Алгоритм адаптивной интерполяции с использованием TT-разложения для моделирования динамических систем с интервальными параметрами // Журнал вычислительной математики и математической физики. 2021. Т. 61, № 9. С. 1416—1430. DOI: 10.31857/S0044466921090106.

6. **Oseledets I. V.** Tensor-train decomposition // SIAM Journal on Scientific Computing. 2011. Vol. 33, No. 5. P. 2295—2317. DOI: 10.1137/090752286.

7. **Oseledets I., Tyrtyshnikov E.** TT-cross approximation for multidimensional arrays // Linear Algebra and its Applications. 2010. Vol. 432, Is. 1. P. 70—88. DOI: 10.1016/j.laa.2009.07.024.

8. **Hitchcock F. L.** The expression of a tensor or a polyadic as a sum of products // J. Math. Phys. 1927. Vol. 6, No. 1. P. 164—189. DOI: 10.1002/sapm192761164.

9. **Press W. H., Teukolsky S. A., Vetterling W. T., Flannery B. P.** 2.6 Singular Value Decomposition. Numerical Recipes in C. 1992. 2nd ed. Cambridge: Cambridge University Press.

10. **Тыртышников Е. Е.** Тензорные аппроксимации матриц, порожденных асимптотически гладкими функциями // Математический сборник. 2003. Т. 194, № 6. С. 147—160. DOI: 10.4213/sm747.

11. **Тыртышников Е. Е., Щербакова Е. М.** Методы неотрицательной матричной факторизации на основе крестовых малоранговых приближений // Журнал вычислительной математики и математической физики. 2019. Т. 59, № 8. С. 1314—1330. DOI: 10.1134/S0044466919080179.

12. **Желтков Д. А., Тыртышников Е. Е.** Параллельная реализация матричного крестового метода // Вычислительные методы и программирование. 2015. Т. 16. С. 369—375. DOI: 10.26089/NumMet.v16r336.

13. **Горейнов С. А., Замарашкин Н. Л., Тыртышников Е. Е.** Псевдоскелетные аппроксимации при помощи подматриц наибольшего объема // Матем. заметки. 1997. Т. 62, Вып. 4. С. 619—623. DOI: 10.4213/MZM1644.

14. **CUDA** Zone. URL: https://developer.nvidia.com/cuda-zone

15. **Керниган Б. У., Ритчи Д. М.** Язык программирования С. М.: Вильямс, 2019. 288 с.

16. **Джосаттис Н. М.** Стандартная библиотека C++. Справочное руководство. М.: Вильямс, 2017. 1129 с.

17. **Matrix** Algebra on GPU and Multicore Architectures (MAGMA). URL: https://icl.cs.utk.edu/magma

18. **cuBLAS**. URL: https://docs.nvidia.com/cuda/cublas

19. **ScaLAPACK**. URL: http://www.netlib.org/scalapack

20. **MATLAB.** URL: https://www.mathworks.com/help/matlab

21. **Оселедец И. В.** Тензорные методы и их применение: дис. ... д-ра физ.-мат. наук: 01.01.07. М.: Учреждение Российской академии наук Институт вычислительной математики РАН, 2009. 282 с.

22. **Goreinov S. A., Oseledets I. V., Savostyanov D. V., Tyrtyshnikov E. E.** How to find a good submatrix. Institute for Computational Mathematics Hong Kong Baptist University, China, 2008. 10 p. DOI: 10.1142/9789812836021_0015.

23. **LAPACK.** URL: http://www.netlib.org/lapack

24. **Chrzeszczyk A., Anders J.** Matrix computations on the GPU. Jan Kochanowski University, Poland, 2017. 455 p.

25. **Griewank A. O.** Generalized Decent for Global Optimization // Journal of Optimization Theory and Applications. 1981. Vol. 34. P. 11—39. DOI:10.1007/BF00933356.

Parallel Approximation of Multivariate Tensors using GPUs

N. S. Kapralov[1,2], nskaprl@gmail.com, **A. Yu. Morozov**[1,3], morozov@infway.ru,
S. P. Nikulin[1], sergeynp@yandex.ru
[1]Moscow Aviation Institute (MAI), Moscow, Russian Federation
[2]Deutsche Bank Technology Center, Moscow, Russian Federation
[3]Federal Research Center "Computer Science and Control" of Russian Academy of Sciences (FRC CSC RAS), Moscow, Russian Federation

Corresponding author:

Morozov Alexander Yu., Senior Lecturer, Moscow Aviation Institute (MAI), Moscow, Russian Federation, Researcher, Federal Research Center "Computer Science and Control" of Russian Academy of Sciences (FRC CSC RAS), Moscow, Russian Federation
E-mail: morozov@infway.ru

Received on December 16, 2021
Accepted on December 28, 2021

When solving many applied and research problems, it becomes necessary to work with multidimensional arrays (tensors). In practice, an efficient and compact representation of these objects is used in the form of so-called tensor trains. The paper considers a parallel implementation of the TT-cross algorithm, which allows one to obtain a decomposition of a multidimensional array into a tensor train using a graphics processor of the CUDA architecture. The main aspects and features of parallelization and implementation of the algorithm are presented. The obtained parallel implementation was tested on a representative number of examples. A significant reduction in computational time is demonstrated in comparison with a similar sequential implementation of the algorithm, which indicates the effectiveness of the proposed approaches to parallelization.

Keywords: *parallelization, tensor train, tensor decomposition, high dimensions, curse of dimensionality, multidimensional arrays, cross approximation, TT-cross, maxvol, low-rank approximation, CUDA, GPU, Nvidia*

For citation:

Kapralov N. S., **Morozov A. Yu.**, **Nikulin S. P.** Parallel Approximation of Multivariate Tensors using GPUs, *Programmnaya Ingeneria*, 2022, vol. 13, no. 2, pp. 94—101.

DOI: 10.17587/prin.13.94-101

References

1. **Morozov A. Y., Reviznikov D. L.** Adaptive Interpolation Algorithm Based on a kd-Tree for Numerical Integration of Systems of Ordinary Differential Equations with Interval Initial Conditions, *Differential Equations,* 2018, vol. 54, no. 7, pp. 945—956, DOI: 10.1134/S0012266118070121.

2. **Morozov A. Yu., Reviznikov D. L.** Modelling of Dynamic Systems with Interval Parameters on Graphic Processors, *Programmnaya Ingeneria*, 2019, vol. 10, no. 2, pp. 69—76. DOI: 10.17587/prin.10.69-76.

3. **Morozov A. Yu.** Parallel Adaptive Interpolation Algorithm based on Sparse Grids for Modeling Dynamic Systems with Interval Parameters, *Programmnaya Ingeneria,* 2021, vol. 12, no. 8, pp. 395—403. DOI: 10.17587/prin.12.395-403 (in Russian).

4. **Morozov A. Y., Zhuravlev A. A., Reviznikov D. L.** Analysis and Optimization of an Adaptive Interpolation Algorithm for the Numerical Solution of a System of Ordinary Differential Equations with Interval Parameters, *Differential Equations,* 2020, vol. 56, no. 7, pp. 935—949, DOI:10.1134/s0012266120070125.

5. **Gidaspov V. Yu., Morozov A. Yu., Reviznikov D. L.** Adaptive Interpolation Algorithm Using TT-Decomposition for Modeling Dynamical Systems with Interval Parameters, *Computational Mathematics and Mathematical Physics,* 2021, vol. 61, no. 9, pp. 1387—1400, DOI: 10.1134/S0965542521090098.

6. **Oseledets I. V.** Tensor-train decomposition, *SIAM Journal on Scientific Computing,* 2011, vol. 33, no. 5, pp. 2295—2317. DOI: 10.1137/090752286.

7. **Oseledets I., Tyrtyshnikov E.** TT-cross approximation for multidimensional arrays, *Linear Algebra and its Applications*, 2010, vol. 432, is. 1, pp. 70—88. DOI: 10.1016/j.laa.2009.07.024.

8. **Hitchcock F. L.** The expression of a tensor or a polyadic as a sum of products, *J. Math. Phys.* 1927, vol. 6, no. 1, pp. 164—189. DOI: 10.1002/sapm192761164.

9. **Press W. H., Teukolsky S. A., Vetterling W. T., Flannery B. P.** *2.6 Singular Value Decomposition, Numerical Recipes in C.* 1992, 2nd edition, Cambridge, Cambridge University Press.

10. **Tyrtyshnikov E. E.** Tensor approximations of matrices generated by asymptotically smooth functions, *Sbornik: Mathematics*, 2003, vol. 194, no. 6. pp. 941—954. DOI: 10.1070/SM2003v194n06ABEH000747.

11. **Tyrtyshnikov E. E., Shcherbakova E. M.** Methods for nonnegative matrix factorization based on low-rank cross approximations, *Computational Mathematics and Mathematical Physics,* 2019, vol. 59, no. 8, pp. 1251—1266. DOI: 10.1134/S0965542519080165.

12. **Zheltkov D. A. Tyrtyshnikov E. E.** A parallel implementation of the matrix cross approximation method, *Numerical Methods and Programming,* 2015, vol. 16, no. 3, pp. 369—375. DOI: 10.26089/NumMet.v16r336.

13. **Goreinov S. A., Zamarashkin N. L., Tyrtyshnikov E. E.** Pseudo-skeleton approximations by matrices of maximal volume, *Mathematical Notes*, 1997, vol. 62, pp. 515—519. DOI:10.1007/2FBF02358985.

14. **CUDA** Zone, available at: https://developer.nvidia.com/cuda-zone

15. **Kernighan B. W., Ritchie D. M.** *C Programming Language*, Prentice-Hall, 1988.

16. **Josuttis N. M.** *The C++ Standard Library: A Tutorial and Reference*, Addison Wesley, 2012, 1136 p.

17. **Matrix** Algebra on GPU and Multicore Architectures (MAGMA), available at: https://icl.cs.utk.edu/magma

18. **cuBLAS,** available at: https://docs.nvidia.com/cuda/cublas

19. **ScaLAPACK,** available at: http://www.netlib.org/scalapack

20. **MATLAB,** available at: https://www.mathworks.com/help/matlab

21. **Oseledets I. V.** Tensor methods and their applications: Dissertation of Doctor of Physical and Mathematical Sciences: 01.01.07. Moscow Center of Fundamental and Applied Mathematics at INM RAS, Moscow, 2009, 282 p. (in Russian).

22. **Goreinov S. A., Oseledets I. V., Savostyanov D. V., Tyrtyshnikov E. E.** *How to find a good submatrix*, Institute for Computational Mathematics Hong Kong Baptist University, China, 2008, 10 p. DOI: 10.1142/9789812836021_0015.

23. **LAPACK**, available at: http://www.netlib.org/lapack

24. **Chrzeszczyk A., Anders J.** *Matrix computations on the GPU,* Jan Kochanowski University, Poland, 2017, 455 p.

25. **Griewank A. O.** Generalized Decent for Global Optimization, *Journal of Optimization Theory and Applications,* 1981, vol. 34, pp. 11—39. DOI: 10.1007/BF00933356.