УДК 004.021

Сортировка массивов коротких последовательностей на GPU для метода сортировки взаимодействий в молекулярной динамике

Э. С. Фомин

fomin@bionet.nsc.ru
ФГБУН «Институт цитогогии и генетики СО РАН» (ИЦиГ)

Поступило в редакцию 27.02.2012

Аннотация. Необходимость сортировки большого числа коротких массивов с числом элементов в диапазоне $30 \le n \le 120$ возникает в молекулярной динамике в методе сортировки взаимодействий. Представлена модификация алгоритма ранговой сортировки для решения данной задачи, реализованная для платформы GP GPU (Tesla C2050). Показано, что предложенная модификация алгоритма в данном диапазоне числа элементов превосходит по эффективности общедоступные для данной платформы библиотечные алгоритмы сортировок.

Ключевые слова. *Молекулярная динамика; сортировка взаимодействий; GPU*

Расчеты ближних несвязующих взаимодействий в молекулярной динамике (МД) занимают существенную, до 90%, часть от общего времени выполнения, что обусловлено следующими факторами:

- по сравнению с расчетами связующих взаимодействий резко (в десятки раз) возрастает число взаимодействий, приходящихся на один атом, что обусловлено необходимостью учета влияния всех $4\pi/3\ \rho r^3_{cutoff}$ атомов, распределенных с плотностью ρ в сфере радиуса, равного расстоянию отсечения взаимодействий r_{cutoff};
- заметно увеличивается неоднородность данных (в сфере взаимодействия оказываются атомы разных типов);
- в потоке данных, собранных из памяти и доставленных на вычислительное устройство, существенно возрастает доля незначимых пар атомов, то есть таких, которые при проверке расстояний приходится отбрасывать;
- существенно возрастает количество промахов кэша, поскольку атомы, хранимые в соседних ячейках памяти компьютера, в силу своего непрерывного движения в процессе моделирования могут существенно разойтись по координатам в пространстве и выйти из области учета взаимодействий.

Таким образом, поток данных, попадающий на вычислительные устройства на этапе расчетов несвязующих взаимодействий, в случае неудачного подбора алгоритмов, отсутствия их адаптации для конкретных вычислительных платформ или, как крайний случай, низкой квалификации пользователя при установке параметров моделирования характеризуется низкой плотностью «полезных» данных, что неизбежно сказывается на эффективности расчетов.

Высокую эффективность расчетов в молекулярной динамике могут обеспечить только адаптивные схемы организации вычислительного процесса, которые позволяют существенно увеличить вероятность нахождения нужных данных в потоке за счет подстраивания схемы вычислений под текущее состояние моделируемой системы и платформу выполнения. В настоящее время такие адаптивные схемы организации вычислений используют метод связанных ячеек (LC – linked cells) [1] для поиска ближайших соседей, метод сортировки взаимодействий (IS – interaction sorting) [2] для ограничения доли незначимых пар атомов, метод Верлет таблицы (VT – Verlet table) [3] для хранения временно взаимодействующих пар, методы переодического переупорядочения атомов (LCR – linked cell based reordering) [4] для минимизации

Работа поддержана грантами СО РАН: интеграционные проекты № 39, № 130, а также Министерством образования и науки РФ (Госконтракты № П857). Использованы ресурсы Суперкомпьютерного комплекса МГУ «ГрафИТ!» в рамках конкурса «Эффективное использование GPU-ускорителей при решении больших задач», проводимого компанией «Т-Платформы».

промахов кэша и автоматическую, «на лету», подстройку параметров расчетной схемы [5] под максимальную эффективность путем контролирования времени работы различных шагов алгоритма. Последние способы дополнительно позволяют избегать деградации выполнения, то есть замедления расчетов, в длительной молекулярной динамике.

Сущность вышеупомянутых методов заключается в следующем:

- *LC – метод связанных ячеек.* Для поиска ближайших соседей используется разделение пространства моделирования на пространственные ячейки с длиной ребра большей или равной радиусу отсечения взаимодействий r_{cutoff}. Для каждой ячейки составляется список соседних к ней ячеек (связанные ячейки). Все атомы системы распределяются по ячейкам согласно своим координатам. Поиск соседей для любого атома выполняется в ячейке самого атома, а также в связанных с ней ячейках.

- *IS – метод сортировки взаимодействий.* Метод связанных ячеек приводит к излишним расходам по оценке расстояний между парами атомов. Действительно, в ячейке с ребром, равным r_{cutoff}, и в 26 связанных с нею ячейках находится $N_1 = 27\rho r^3_{\text{cutoff}}$ атомов, при этом в пределах r_{cutoff} находится всего $N_2 = 4\pi/3\rho r^3_{\text{cutoff}}$ атомов. Таким образом, для всех связанных ячеек доля взаимодействующих атомов достаточно мала $N_1/N_2 \times 100\% \sim 16\%$. С целью избежать излишних проверок расстояний атомы сортируются согласно их координатам вдоль оси, соединяющей любые две соседние ячейки. Имея отсортированные последовательности атомов двух соседних ячеек при расчетах расстояний, возможно отбрасывать концы последовательностей, если расстояние между проверяемыми атомами превосходит r_{cutoff}.

- *VT – метод Верлет таблицы*. Найденные пары взаимодействующих атомов сохраняются в таблице, в которой для каждого атома хранится список атомов, находящихся на расстоянии, не превышающем $r_{\text{cutoff}} + r_{\text{skin}}$, где r_{skin} – страховочная область. Регулируя ширину страховочной области, можно подобрать оптимальное соотношение между временем расчета взаимодействий и частотой перестройки таблицы. Необходимость перестройки таблицы возникает всякий раз, когда какой-либо атом в результате своего движения перешел пределы страховочной области r_{skin}.

- *LCR – метод переупорядочения атомов.* Все атомы и их таблицы параметров взаимодействий пересортировываются в памяти согласно текущим координатам таким образом, чтобы атомы, близкие в пространстве, были близки по адресам памяти. Такая пересортировка выполняется через заданный интервал модельного времени системы, который пропорционален подвижности молекул системы.

В любой адаптивной схеме для организации необходимого порядка обработки данных интенсивно используются различного вида сортировки. Именно благодаря сортировкам атомов по пространственным ячейкам снижается вычислительная сложность поиска ближайших соседей с $0(N^2)$ до $0(N)$ (метод LC [1], комбинированный метод LC + VT [6]), существенно увеличивается доля взаимодействующих пар в потоке данных от ~16% до ~60% (метод IS [2]) и обеспечивается поддержка пространственной и временной локальности данных, повышая вероятность нахождения данных в кэше с 90,9 % (без упорядочения) до 99,4 % на процессорах стандартной архитектуры *x*86 (метод LCR [4]).

Любая сортировка данных требует соответствующих накладных расходов, и может быть использована только, если выгода от ее использования превышает потери. Сортировка взаимодействий увеличивает общую производительность молекулярной динамики для реализаций, основанных на широкораспространенном комбинированном методе LC + VT до 10 % (за счет ускорения этапа построения таблицы более чем в два в половиной раза) [7]. Сортировка взаимодействий более чем в два раза увеличивает общую производительность динамики для реализаций, основанных на оригинальном методе LC [8], что позволяет последнему при расчетах плотных систем с высокой мобильностью атомов превзойти по эффективности комбинированный метод CL + VT. Относительные потери на сортировку в обоих подходах не превышают 5–10 % для соответствующих этапов алгоритма для последовательных версий и увеличиваются до 15 % для векторизованных SSE версий [5]. Увеличение относительной доли затрат на алгоритмы сортировки в векторизованом коде обусловлено невозможностью их эффективной векторизации.

Метод сортировки взаимодействий, обладая доказанной эффективностью для стандарной архитектуры *x*86, не гарантированно даст эффект при его использовании в программах молекулярной динамики, выполняющихся на GPU. Наиболее существенной проблемой является необходимость многочисленных сортировок

данных, и далеко не очевидно, что GPU позволит обеспечить необходимую эффективность. Основные особенности алгоритма сортировки взаимодействий, которые необходимо учитывать при портировании, следующие:

- Сортировке подвергаются массивы с весьма небольшим числом элементов в них. Действительно, среднее число молекул воды на любую пространственную ячейку размером 10 × × 10 × 10 Å^3 при нормальной плотности не превышает ~30 (полное число атомов ~90).
- Общий объем необходимых сортировок значителен, то есть число небольших по ~30 элементов последовательностей велико. Действительно, общее число определяется числом ячеек, на которые разбита пространственная область [N_x, N_y, N_z] и числом пар, которые образует любая ячейка с соседями, то есть 26. Таким образом, полное число массивов для сортировки равно 26 N_x N_y N_z, что для среднего расчета достигает величины в несколько десятков тысяч. При этом такие сортировки требуется выполнять на каждом шаге МД.

Итого для обеспечения эффективности молекулярной динамики на аппаратуре, позволяющей «мелкозернистый» параллелизм (векторные операции с большим числом компонент, либо огромное число легких потоков, как на GPU) необходимо построение алгоритма сортировки большого объема коротких последовательностей. В данной работе анализируется возможность применения алгоритма ранговой сортировки на платформе GPU для решения этой задачи. Выбор данного алгоритма обусловлен тем, что наряду с тем, что он имеет модификации с вычислительной сложностью O(N × log N), такой же как у любых прочих эффективных алгоритмов сортировки, он имеет «соблазнительную» теоретическую возможность выполняться за время O(1). А когда есть выбор – допустить простаивание потоков или бросить их все на выполнение задачи пусть и неэффективным O(N^2) образом, результат выбора очевиден.

РАНГОВАЯ СОРТИРОВКА ПОСЛЕДОВАТЕЛЬНОСТИ

Алгоритмы ранговой сортировки [9] основаны на вычислении рангов элементов и их упорядочении в соответствии с полученным рангом. Ранги элементов могут быть определены любым способом, однако в практике зачастую достаточным является следующая формула вычисления ранга: $\mathrm{rank}(s_i) = \Sigma_j \mathrm{rank}(R(s_i, s_j))$, где ранжируются результаты $R(s_i, s_j)$ операций попарного сравнения элементов и выполняется суммирование полученных рангов. В общем случае в зависимости от выбора операции сравнения, значение ранга может принимать как целочисленное, так и вещественное значение. Заметим, что для наиболее естественного выбора в качестве операции сравнения R операции «<» (меньше) значение ранга имеет булевский тип, который транслируется в целочисленный, согласно преобразованию true → 1 и false → 0. Если операция R между элементами неопределенна, то ее ранг принимается равный нулю. Поскольку для получения ранга элемента при таком выборе функции ранжирования требуется попарное сравнение всех элементов, то вычислительная сложность таких алгоритмов равна O(N^2), где N – число элементов в сортируемой последовательности. Такая высокая вычислительная сложность означает неприменимость этих алгоритмов в практических расчетах систем с огромным числом элементов $N >> 1$.

Если операция сравнения элементов последовательности является транзитивной, исчезает необходимость сравнения всех N^2 пар. В этом случае возможно использование подхода «разделяй и властвуй», что позволяет для алгоритмов ранговой сортировки строить схемы с вычислительной сложностью O(N × log N), как и для стандартных алгоритмов быстрой сортировки, типа qsort или merge [9]. Примером подобного алгоритма является алгоритм, описанный в [10] и реализованный для платформы GPU.

Однако при использовании вычислительной платформы с «идеальным» параллелизмом, алгоритмы ранговой сортировки превосходят по скорости вычислений стандартные алгоритмы, поскольку имеют временную сложность O(1) [11]. Здесь под «идеальной» параллельной системой понимается система, которая может обеспечить сколь угодно большое число реально одновременно выполняемых потоков и поддерживает на машинном уровне параллельные операции для векторов, такие как scan, reduce, transposition [12]. Причем важным условием является то, что данные параллельные операции выполняются за такое же время, как и любая скалярная операция, например за один такт.

Для такой идеальной параллельной системы алгоритм ранговой сортировки выполняется всего за 3 операции:

- расчет рангов операции сравнения с помощью N^2 потоков для всех пар элементов мно-

жества и формирование матрицы, в которой на пересечении k-строки с m-столбцом находится результат сравнения элементов s_k и s_m;

- выполнение операции reduce (суммирование) над всеми рядами полученной матрицы с формированием вектора, содержащего ранги всех элементов последовательности;
- выполнение операции transpose для переупорядочения элементов последовательности в соответствии с вектором рангов.

Поскольку любая реальная параллельная вычислительная система является приближением к «идеальной», то достижение вычислительной сложности O(1) невозможно. Причины этого очевидны, поскольку:

- реальная параллельная система может обеспечить большое, но ограниченное число параллельных потоков,
- даже при поддержке на машинном уровне параллельных операций для векторов, вряд ли реализация параллельных операций, требующих суммирования, таких как scan и reduce, будет выполняться за то же время, что и скалярные операции.

Первое возражение не является существенным, поскольку для задачи сортировки N элементов достаточно N^2 потоков. Например, для задачи сортировки коротких последовательностей длиной не более 32 элементов достаточно иметь 1024 потока, что уже могут обеспечить GPU устройства с compute capability версии 2.0 и выше [13].

Второе возражение существенно, поскольку в настоящее время даже наиболее востребованные параллельные операции, типа scan и reduce, реализованы с помощью библиотечных функций, например библиотеки CUDA C SDK [14], и включают циклы по элементам обрабатываемых последовательностей. Таким образом, хотя теоретическая временная сложность алгоритма равна O(1), ожидаемая временная сложность реализации алгоритма, как описано ниже, может быть равной $O(\log^2 N)$.

В данной работе выполнена реализация алгоритмов ранговой сортировки для платформы Tesla C2050 с помощью технологии CUDA. Выполненно сравнение эффективности разработанного алгоритма с эффективностью наиболее широко используемых параллельных алгоритмов сортировки, которые имеют временную сложность $O(\log^2 N)$, таких как mergeSort, bitonic Sort, odd-even, входящими в пакет CUDA SDK [14]. Сравнение выполнено как для диапазона $1 \le n \le 32$ элемента, в котором теоретическая временная сложность равна O(1) и для диапазона $32 \le n \le 1024$, где теоретическая временная сложность пропорциональна O(N).

Сортировка последовательности целых чисел без дубликатов

Временная сложность O(1) для сортировки последовательности чисел достигается при условии, что число доступных потоков выполнения не менее чем N^2, где N – число элементов в последовательности. Такое число потоков необходимо, чтобы за один шаг можно было получить таблицу сравнения $R(s_i, s_j)$ для всех пар элементов s_i и s_j.

Ранг операции сравнения выбирается как $\text{rank}(s_i, s_j) = (\text{bool})\ (s_i > s_j)$. В матрицу сравнения записываются значения целые числа {0, 1} представляющие булевские значения {false, true}. Ранг элемента рассчитывается как сумма значений в рядах матрицы $\text{rank}(s_i) = \sum_j \text{rank}(s_i, s_j)$.

Если некоторая последовательность S упорядочена, $s_0 \le s_1 \le .. \le s_{n-1}$, то для любого ее элемента s_k выполняется: $s_k > s_0$ (true), $s_k > s_1$ (true), .. $s_k > s_{k-1}$ (true), $s_k > s_k$ (false), $s_k > s_{k+1}$ (false), $s_k > > s_{n-1}$ (false). То есть полное число парных сравнений данного элемента со всеми остальными элементами последовательности, которое имеет значение равное true, равно индексу элемента в данной последовательности. То есть выполняется формула $\sum_i \text{rank}(s_k, s_i) = k$. Значение ранга элемента не изменится, если последовательность будет переупорядочена произвольным образом, поскольку это приведет к перестановке элементов в сумме, но не к изменению результата, поскольку результат операции сложения коммутативен. Таким образом, ранг элемента в последовательности не зависит от упорядочения последовательности. Этот факт позволяет легко находить для любого элемента неупорядоченной последовательности его позицию в упорядоченной последовательности. Для этого достаточно просуммировать все значения парных сравнений данного элемента со всеми остальными элементами.

Таким образом, возвращаясь к алгоритму, чтобы получить позицию любого элемента в отсортированной последовательности, необходимо суммировать все значения в рядах матрицы сравнений. Для идеальной параллельной машины такое суммирование выполняется за одну параллельную операцию типа scan или reduce. Для реальной машины подобная парал-

лельная операция реализуется либо программными средствами, либо на машинном уровне. В обоих случаях эффективный алгоритм такого суммирования требует $\log^2 N$ шагов. Поскольку все суммирование по рядам может выполняться независимо и число рядов меньше числа потоков $N < P$, то суммарное время получения рангов всех элементов не изменяется и равно $\log^2 N$.

Последним этапом данного алгоритма является восстановление правильного порядка элементов последовательности. Это также может быть сделано за один шаг, поскольку знание индекса элементов позволяет делать их запись независимо друг от друга.

Итого, полное число шагов, необходимых алгоритму, равно 3 для идеальной параллельной машины, либо для реальной параллельной машины определяется шагом суммирования, то есть пропорционально $\log^2 N$.

Алгоритм является эффективным только в случае, когда число доступных потоков подчиняется условию $P \geq N^2$. Если оно нарушается, то невозможно будет получить полную таблицу парных сравнений за один шаг, что в свою очередь в худшую сторону изменит эффективность алгоритма. Действительно, если допустить, что число потоков является недостаточным, чтобы обработать таблицу за один шаг, то есть выполняется $P < N^2$, но достаточно, чтобы обработать за один шаг хотя бы один ряд целиком, то есть $P > N$, то таблица парных сравнений должна обрабатываться частями. Полная высота обрабатываемой части таблицы (при условии ее ширины N) равна $k = P / N$, а число частей таблицы равно $m = N / k$, или после подстановки значения k величина m равна N^2 / P. Учитывая, что число шагов для нахождения позиций элементов не меняется $\log^2 N$, получаем, что полное число шагов равно $(N^2 / P) \times \log^2 N$. Таким образом, эффективность данного алгоритма при числе потоков P, удовлетворяющем условию $N \leq \leq P \leq N^2$\$, находится в диапазоне $[N \times \log^2 N, \log^2 N]$.

Сортировка последовательности целых чисел с дубликатами

Наличие дубликатов в последовательности приводит к неработоспособности вышеприведенного алгоритма. Это происходит по той причине, что все дубликаты имеют один и тот же ранг. Как результат, запись элементов исходной последовательности по позициям, определяемым рангами, приводит к тому, что в конечной последовательности появляются позиции, в которые вообще не производится запись. То есть в пересортированной последовательности появляются элементы, значение которых не определено. Легко понять, что позиции с неопределенными элементами появляются сразу за элементом, имеющим дубликаты, и их число равно n_d, где n_d – число дубликатов (рис. 1). Под числом дубликатов здесь и далее будем понимать полное число элементов, совпадающих по значению, без включения исходного элемента. Дополнительно ограничимся также и тем предположением, что в последовательности находятся дубликаты только с одним значением. Последнее условие никак не ограничивает корректность нижеприведенных рассуждений, которые верны и в случае дубликатов с различными значениями, однако это упрощает описание.

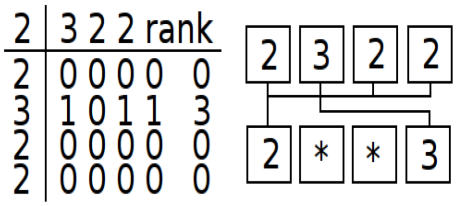

Рис. 1. Пример таблицы сравнений для последовательности целых с дубликатами (слева) и результирующая последовательность после обработки алгоритмом (справа)

Идея, позволяющая разрешить проблему с дубликатами, заключается в том, чтобы найти такое преобразование матрицы сравнений $U \rightarrow U'$, которое с одной стороны изменит конечные ранги дубликатов, упорядочив их в области $[\text{rank}(s), \text{rank}(s) + n_d]$, где $\text{rank}(s)$ – ранг дубликата s и n_d – число дубликатов, и с другой стороны – не изменит ранги элементов без дубликатов.

Сделаем следующее модификацию определения ранга сравнения, определив его как $\text{rank}_{\text{new}}(s_i, s_j) = s_i - s_j$. Очевидно, что операция $\text{bool}(x) = \{1, x > 0 \mid 0, x \leq 0\}$ приводит модифицированный ранг rank_{new} к значению исходного ранга $\text{bool}(\text{rank}_{\text{new}}(s_i, s_j)) = \text{rank}(s_i, s_j)$, то есть при таком расширении не потеряна информация об упорядочении элементов. Операции $\text{bool}(x)$ имеет следующее свойство: $\text{bool}(x + n) = \text{bool}(x)$, для всех $|x| > n$, где n – целое положительное число, а операция $|x| = \{x, x>0 \mid -x, x < 0\}$ есть операция взятия абсолютного значения. Таким образом, матрица сравнений допускает добавление любого числа n к элементу s_{ij} в любой позиции, тем самым изменяя модифицированный

ранг, но без изменения значений исходных рангов элементов при условии, что число $n < |s_{ij}|$. Поскольку для дубликатов $|s_{ij}| = 0$, то для них при таком преобразовании ранг меняется всегда.

Разное поведение рангов для элементов с дубликатами и без них позволяет построить преобразование матрицы сравнения, в котором позволяет изменить ранги дубликатов с сохранением прежних рангов у всех остальных элементов. Легко проверить, что матрица T, состоящая из одних единиц и с нулями на главной диагонали обладает требуемым свойством. Действительно:

- Для элементов без дубликатов вне главной диагонали возможны два варианта $s_{ij} < 0$ и $s_{ij} > 0$. Добавление 1 к целому отрицательному числу даст либо отрицательное значение, либо нуль, то есть не изменит значение функции bool. Если $s_{ij} > 0$, то значение функции bool также не меняется при увеличении s_{ij}.
- Для элементов с дубликатами $s_{ij} = 0$. При добавлении 1 значение функции bool меняется: bool(0) = 0 и bool(1) = 1.

К сожалению, преобразование $U = U + T$ изменяет ранги всех дубликатов одинаково. Это изменение равно $\text{rank}(s) \to \text{rank}(s) + n_d$, где n_d – число дубликатов. Действительно, в ряду i ранги сравнений меняются только у дубликатов (за исключением лежащего на главной диагонали), а так как изменение равно 1 и их число n_d, то ранг элемента меняется на n_d. Это не то, что хотелось бы от алгоритма. От алгоритма требуется, чтобы ранг первого дубликата не менялся, второго изменился на 1, третьего на 2 и т. д.

Чтобы обеспечить различное изменение ранга дубликатов, заметим, что допустимы дополнительные модификации матрицы T, сохраняющие ранги элементов без дубликатов, а именно удаление единиц с любой позиции матрицы T. Действительно, если позиция, с которой удаляется 1, соответствует позиции, не связанной с дубликатами, то ранг элемента на ней вообще не изменяется при замене 1 на 0. Однако такие модификации матрицы T для позиций, связанных с дубликатами, приводят уменьшению ранга $\text{rank}(s) \to \text{rank}(s) - 1$. Последнее свойство позволяет построить матрицу T, которая обладает нужными свойствами. Для этого нужно оставить один ряд неизменным, затем убрать одну единичку из любого оставшегося ряда матрицы T, затем две единички из любого из оставшихся рядов и т. д. Порядок удаления единички в ряду и порядок рядов несущественен. Таким образом, возможно получить $(N - 1)! \sum(1/k!)$ матриц T. Поскольку любая из таких матриц годится, выберем наиболее простую из них, а именно ту, которая получается, если первый ряд не менять, из второго убрать единичку слева от главной диагонали, из второго – две единички слева от главной диагонали и т. д., то есть получить верхнетреугольную матрицу.

Полный алгоритм выглядит следующим образом:

- Составляем матрицу сравнений U с использованием ранга $\text{rank}(s_i, s_j) = s_i - s_j$.
- Делаем преобразование $U \to U + T$, где T – верхнетреугольная матрица, состоящая из единиц, с нулями на главной диагонали.
- Выполняем операцию bool над каждым элементом матрицы U, то есть $U \to \text{bool}(U)$.
- Вычисляем ранг каждого элемента исходной последовательности S по алгоритму, описанному в разделе «сортировка целых без дубликатов».

Сортировка последовательности вещественных чисел без дубликатов

Выполняется по алгоритму, описанному в разделе «сортировка целых без дубликатов».

Сортировка последовательности вещественных чисел с дубликатами

- Рассчитываем ранги элементов по алгоритму, описанному в разделе «сортировка целых без дубликатов». Полученная последовательность рангов $I = \{i_0, i_1, .., i_{n-1}\}$ является неупорядоченной последовательностью целых чисел с дубликатами.
- Для полученной последовательности I рассчитываем ранги I' по алгоритму, описанному в разделе «сортировка целых с дубликатами».
- Пересортировываем исходную последовательность согласно рангам I', полученным на предыдущем шаге.

ДЕТАЛИ РЕАЛИЗАЦИИ

Алгоритм ранговой сортировки реализован на языке C++ с использованием CUDA Toolkit 3.2 на базе программы MOLKERN [15]. Реализованы три версии алгоритма «1», «lg N» и «N»:

- «1» версия, которая использует максимальное доступное число потоков ядра (1024) таким образом, чтобы достичь временной слож-

ности близкой O(1) на последовательностях до 32 элементов;

- «lg *N*» базовая версия, построенная на подходе «разделяй и властвуй» по аналогии с [10], с вычислительной сложностью $O(\log^2 N)$ для обработки последовательностей до 1024 элементов;
- «*N*» дополнительная версия, построенная на полном парном сравнении всех элементов последовательностей с числом используемых потоков равном числу элементов последовательности, то есть с вычислительной сложностью O(*N*) в исследуемом диапазоне.

Дополнительная версия использована только для сравнительной демонстрации временной сложности. Все версии алгоритма обрабатывают последовательности с длиной, равной $2k$, где k – целое число. Операции reduce и transform были реализованы программным образом, при этом все данные для них размещались в разделяемой памяти. Так что для последовательностей длиной менее 32 элементов временная сложность операции reduce в лучшем случае составляла $O(\log^2 N)$, а операции transform – O(1). В худшем случае для операции transform, поскольку для произвольных последовательностей нельзя гарантировать какой либо определенный порядок доступа к памяти, временная сложность могла составлять O(*N*), по причине наличия конфликтов блоков при обращении к разделяемой памяти.

Время выполнения сортировки измерялось функциями тайминга, которые предоставляются SDK. Поскольку время измеряется на процессоре общего назначения и включает в себя время загрузки/выгрузки ядра, то при замерах ядро принудительно выполнялось многократно (не менее 1000 раз), тем самым относительный вклад от времени загрузки/выгрузки ядра падал в 1000 раз.

Для использованных примеров сортировки был написан выполняемый на ядре параллельный генератор случайных чисел. Был использован алгоритм генерации случайных чисел Парка-Миллера [16]. Для уничтожения нежелательных корреляций между сериями случайных чисел, генерируемых разными потоками, на ядро передавался массив различных простых чисел для инициализации генераторов. Размерность массива совпадала с максимально возможным числом потоков на ядре.

Расчеты выполнялись на процессоре Tesla C2050. Данный процессор может обеспечить 1024 потоков на ядро. Таким образом, «1» версия алгоритма может показать время близкое к константному только для последовательностей с числом элементов не выше 32. При большем числе элементов в последовательностях его временная сложность является квадратичной $O(N^2)$. Дополнительная версия «*N*» алгоритма обладает линейной временной сложностью O(N) вплоть до числа элементов в последовательности, не превышающем максимально возможного числа потоков на ядро. При большем числе элементов данная версия обладает также квадратичной временной сложностью $O(N^2)$.

В реализации использовались следующие модификации, отличающие ее от алгоритма [11]:

- внесено отображение вещественных чисел (координаты молекул в ячейках) на короткие целые;
- полученное целое значение записывается в верхние 2 байта целого числа, а номер элемента в последовательности записывается в нижние 2 байта. Данный прием ликвидирует косвенность обращения к данным при сортировке и позволяет избежать проблемы с сортировкой дубликатов [11], которые возникают в случае отображения разных вещественных чисел на одинаковые целые;
- используется двухэтапный подход (создается план сортировки по базисным атомам молекулы и план сортировки используется для всех остальных атомов тех же молекул).

Выполненная реализация алгоритма оптимизирована для случая длины последовательностей в пределах 256 элементов, для последовательностей длиной менее 32 элементов дополнительно исключена излишняя синхронизация между потоками. Тестирование выполнялось на коротких последовательностях элементов, длиной не более 256 элементов. Выполнялось сравнение полученного алгоритма со стандартными алгоритмами сортировок из библиотеки CUDA SDK.

РЕЗУЛЬТАТЫ РАСЧЕТОВ

Скорость работы разработанного алгоритма и эффективность полученной реализации сравнивалась с эффективностью работы различных сортирующих алгоритмов, предоставленных CUDA SDK. Эти алгоритмы включали в себя сортировку слиянием, radix сортировку и сортирующие сети (битональную и четно-нечетную) [14]. Результаты сравнения в логарифмическом

масштабе приведены на рис. 2. Такое сравнение позволяет оценить качество реализации алгоритма ранговой сортировки и соответствие оценок и реальных значений теоретической сложности.

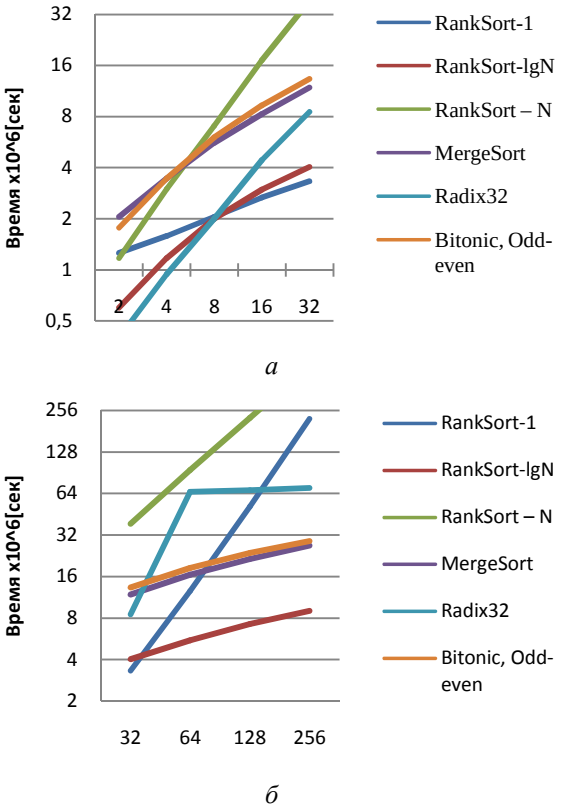

Рис. 2. Время сортировки последовательностей длиной до 32 (*а*) и 256 (*б*) элементов

Сравнение времени работы сделано для трех вариантов «1», «lg *N*» и «N» алгоритма ранговой сортировки. Вариант «1» соответствовал реализации, в которой использовались все доступные в ядре потоки, то есть 256 для Tesla C2050. Данный вариант эмулирует счет с временной сложностью O(1). Вариант «*N*» является дополнительным, в нем число используемых потоков соответствовало числу сортируемых элементов, он эмулирует вычисления с временной сложностью O(*N*). И вариант «lg *N*» соответствует реализации, в которой используется свойство транзитивности элементов и подход «разделяй и властвуй». Его временная сложность равна $O(\log^2 N)$. Сравнение сделано для сортировки последовательностей длиной до 256 элементов и состоящей из вещественных чисел одинарной точности float.

На рис. 2 видно, что варианты «*N*» и «lg *N*» показывают временную сложность, совпадающую с теоретической. Вариант «1» при длине последовательности до 32 элементов, то есть до той длины, когда все возможные 1024 потока могут быть использованы для расчета матрицы сравнений за «один» шаг, имеет наилучшее время выполнения, хотя его временная сложность все же выше константной. При большем числе потоков данный вариант показывает квадратичную зависимость от числа элементов, но в любом случае в рассматриваемой области он оказывается эффективней варианта «*N*», хотя второй имеет линейную сложность. При обработке последовательности длиной 1024 элементов оба варианта «1» и «*N*» имеют равную производительность (данная точка не захватывается графиками). В целом вплоть до ~100 элементов время расчетов вариантом «1» алгоритма ранговой сортировки сравнимо со временем работы прочих популярных алгоритмов сортировки.

Несмотря на то, что алгоритм с «константной» временной сложностью показал наилучшие результаты в области малых длин последовательностей, эффект является незначительным, чтобы быть полезным для практических расчетов на данном этапе развития GPU устройств. Оптимальные, «ровные» и практически полезные результаты в заданной области длин последовательностей демонстрирует «lg *N*» вариант алгоритма, построенный на подходе «разделяй и властвуй» и имеющий вычислительную сложность $O(\log^2 N)$.

Графики показывают также, что качество реализации алгоритма соответствует общедоступным библиотечным реализациям алгоритмов сортировки. Полученная реализация алгоритма ранговой сортировки (вариант «lg *N*») при сортировке коротких последовательностей превосходит почти втрое рассмотренные библиотечные алгоритмы. Исходя из того, что «lg *N*» вариант ранговой сортировки по построению полностью аналогичен алгоритму MergeSort [10], разница в производительности связана только с адаптацией кода под специальный случай коротких последовательностей.

ЗАКЛЮЧЕНИЕ

В работе проанализирована возможность портирования на платформу GPU алгоритма сортировки взаимодействий, использующегося в расчетах ближних невалентных взаимодействий в молекулярной динамике. Показано, что для данного алгоритма возможно преодоление его «узких мест», ограничивающих область его применимости, и связанных с сортировкой большого объема коротких последовательно-

стей пар взаимодействующих атомов. Разработана модификация алгоритма параллельной ранговой сортировки, специально адаптированая под случай последовательностей с длиной, не превышающей 256 элементов, требуемой методом. Проанализирована эффективность разных версий предложеного алгоритма, имеющих различную вычислительную и временную сложность. Выполнено сравнение эффективности разработанного алгоритма с реализациями алгоритмов MergeSort, RadixSort, BitonicSort и OddEvenSort, представленными в библиотеке CUDA SDK, и в диапазоне длины последовательностей, не превышающем 256 элементов, показана существенно (~3 раза) большая эффективность предложенного алгоритма.

СПИСОК ЛИТЕРАТУРЫ

1. **Quentrec B., Brot C.** New methods for searching for neighbours in molecular dynamics computations // J. Comp. Phys. 1973. V. 13. P. 430–432.

2. **Gonnet P.** A simple algorithm to accelerate the computation of non-bonded interactions in cell-based molecular dynamics simulations // J. Comp. Chem. 2007. V. 28 (2). P. 570–573.

3. **Verlet L.** Computer Experiments on Classical Fluids // Phys Rev. 1967. V. 159. P. 98–103.

4. **Meloni S., Rosati M.** Efficient particle labeling in atomistic simulations // J. Chem. Phys. 2007. V. 126. P. 102–121.

5. **Fomin E. S.** Consideration of Data Load Time on Modern Processors for the Verlet Table and Linked Cell Algorithms // J. Comp. Chem. 2011. V. 32(7). P. 1386–1399.

6. **Yao Z., Wang J.-S., Liu G.-R., Cheng M.** Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method // Comp. Phys. Comm. 2004. V. 161. P. 27–35.

7. **Cui Z. W., Sun Y., Qu J. M.** The neighbor list algorithm for a parallelepiped box in molecular dynamics simulations // Chinese Sci Bull. 2009. V. 54(9). P. 1463–1469.

8. **Фомин Э. С.** Сравнение метода Верлет таблицы и метода связанных ячеек для последовательной, векторизованной и многопоточной реализаций // Вычислительные методы и программирование. 2010. Т. 11. С. 299–305.

9. **Knuth D. E.** The Art of Computer Programming. V. 3. Sorting and Searching, Addison-Wesley, 1973.

10. **Satish N., Harris M., Garland M.** Designing efficient sorting algorithms for manycore GPUs // Parallel and Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on 23-29 May 2009, P. 1–10.

11. **Louri A., Hatch J. A., Na J.** A Constant-Time Parallel Sorting Algorithm and Its Optical Implementation // EEE Micro. P. 60–71.

12. **Blelloch G. E.** Scans as primitive parallel operations // Computers, IEEE Transactions on. 1989. V. 38(11). P. 1526–1538.

13. NVIDIA CUDA Programming Guide 3.1, 2010, © 2006–2010 NVIDIA Corporation.

14. NVIDIA CUDA Toolkit 3.2, 2010, © 2006-2010 NVIDIA Corporation.

15. Библиотека программных компонент MOLKERN для построения программ молекулярного моделирования / Э. С. Фомин [и др.] // Биофизика. 2006. Т. 51, № 7. С. 110–113.

16. **Park S. K., Miller K. W.** Random Number Generators: Good Ones are Hard to Find // Communications of the ACM. 1988. V. 31 (10). P. 1192–1201.

ОБ АВТОРЕ

Фомин Эдуард Станиславович, ст. науч. сотр. сектора высокопроизводительных вычислений (ИЦиГ СО РАН, г. Новосибирск). Канд. физ.-мат. наук по химической физике (ИНХ СО РАН, 1997, г. Новосибирск). Иссл. в обл. квантовой химии, рентгеновской спектроскопии, моделирования структуры и динамики сложных белковых комплексов.

METADATA

Title: Sorting short arrays on the GPU for the interaction sorting method in molecular dynamics.

Authors: E. S. Fomin

Affiliation: Institute Cytology and Genetics (IC&G), SB RAS, Russia.

Email: fomin@bionet.nsc.ru.

Language: Russian.

Source: Vestnik UGATU (Scientific journal of Ufa State Aviation Technical University), 2013, Vol. 17, No. 2(55), pp. 75–84. ISSN 2225-2789 (Online), ISSN 1992-6502 (Print).

Abstract: The need for sorting large number of short arrays with the number of elements in the range of $30 \le n \le 120$ occurs for the interaction sorting method in molecular dynamics. A modification of the rank-sorting algorithm for solving this problem, implemented for the GPGPU (Tesla C2050), is presented. It is shown that the proposed algorithm outperforms other sorting algorithms implemented for the GPGPU.

Key words: molecular dynamics; interaction sorting; GPGPU.

References (English Transliteration):

1. **Quentrec B., Brot C.** New methods for searching for neighbours in molecular dynamics computations. // J. Comp. Phys. 1973, V. 13, P. 430-432.

2. **Gonnet P.** A simple algorithm to accelerate the computation of non-bonded interactions in cell-based molecular dynamics simulations. // J. Comp. Chem. 2007, V. 28 (2), P. 570-573.

3. **Verlet L.** Computer Experiments on Classical Flu-ids. // Phys Rev 1967, V. 159, P. 98-103.

4. **Meloni S., Rosati M.** Efficient particle labeling in atomistic simulations // J. Chem. Phys. 2007, V. 126, P. 121102.

5. **Fomin E. S.** Consideration of Data Load Time on Modern Processors for the Verlet Table and Linked Cell Algorithms. // J. Comp. Chem., 2011, V. 32(7), P. 1386-1399.

6. **Yao Z., Wang J.-S., Liu G.-R., Cheng M.** Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method. // Comp. Phys. Comm., 2004, V. 161, P. 27-35.

7. **Cui Z. W., Sun Y., Qu J. M.** The neighbor list al-gorithm for a parallelepiped box in molecular dynamics simulations. // Chinese Sci Bull, 2009, V. 54(9), P. 1463-1469.

8. **Fomin E. S.** Comparison of the Verlet table and cell-linked list algorithms for sequential, vectorized and multithreaded implementations // Numerical Methods and Programming, 2010, Vol. 11, pp. 299-305.

9. **Knuth D. E.** / The Art of Computer Programming. V. 3. Sorting and Searching, Addison-Wesley, 1973.

10. **Satish N., Harris M., Garland M.** Designing effi-cient sorting algorithms for manycore GPUs. // Parallel and Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on 23-29 May 2009, P. 1-10.

11. **Louri A., Hatch J. A., Na J.** A Constant-Time Parallel Sorting Algorithm and Its Optical Implementation. // EEE Micro, P. 60-71.

12. **Blelloch G. E.** Scans as primitive parallel operations. // Computers, IEEE Transactions on, 1989, V. 38(11), P. 1526 - 1538.

13. NVIDIA CUDA Programming Guide 3.1, 2010, © 2006-2010 NVIDIA Corporation.

14. NVIDIA CUDA Toolkit 3.2, 2010, © 2006-2010 NVIDIA Corporation.

15. **Fomin E. S., Alemasov N. A., Chirtsov A. S., Fomin A. E.** MOLKERN: A library of software components for molecular modeling programs. // Biophysics, 2006, Vol. 51(1), supplement, pp. 110-112.

16. **Park S. K., Miller K. W.** Random Number Generators: Good Ones are Hard to Find. // Communications of the ACM, 1988, V. 31 (10), P. 1192-1201.

About author:

Fomin Eduard Stanislavovichh, Senior Researcher, Dept. Department of High Performance Computing in Bioinformatics, Institute of Cytology and Genetics, SB RAS. Dipl. Chemical Physicist (Novosibirsk State Univ., 1985). Cand. of Phys.-Math. Sci. (Institute of Inorganic Chemistry, SB RAS, 1996).