УДК 004.021:004.94:519.683:519.684:544.223

ИСПОЛЬЗОВАНИЕ ГРАФИЧЕСКИХ ПРОЦЕССОРОВ И ТЕХНОЛОГИИ CUDA ДЛЯ ЗАДАЧ МОЛЕКУЛЯРНОЙ ДИНАМИКИ

А. С Боярченков[1], **С. И. Поташников**[2]

Рассмотрена параллельная реализация расчета парных межчастичных взаимодействий в методе молекулярной динамики при нулевых граничных условиях на графических процессорах с применением платформы NVIDIA CUDA. Впервые предложена эффективная реализация с использованием третьего закона Ньютона на основе технологии CUDA. Предложены приемы оптимизации кода. На видеокарте NVIDIA GeForce 8800 GTX по сравнению со скалярной версией на процессоре AMD Athlon64 2.1 ГГц достигнуто ускорение до 660 раз для системы из 49152 частиц.

Ключевые слова: молекулярная динамика, параллельные вычисления, графические процессоры, технология CUDA.

1. Введение. Метод динамики частиц (метод молекулярной динамики, или МД) — один из самых широко используемых в современной вычислительной науке [1]. Существуют две похожие задачи: гравитационная и электромагнитная динамика, где дальнодействующие силы (которые убывают пропорционально квадрату расстояния) считаются суперпозицией всех парных взаимодействий (закон Ньютона, закон Кулона), а динамика системы моделируется уравнениями движений Ньютона [2].

Задачи МД-моделирования условно можно разбить на два класса:

— моделирование больших открытых систем (10–1000 тысяч частиц) при нулевых граничных условиях (НГУ); методы расчета — прямое суммирование и иерархические мультипольные методы [3];

— моделирование бесконечных систем при периодических граничных условиях (ПГУ) транслированием области из 100–10000 частиц; методы расчета — суммирование Эвальда и "частица–сетка" (использующие переход к обратному пространству типа преобразования Фурье) [3].

Существенное развитие в области МД-моделирования связано с использованием параллельных расчетов: либо параллельный расчет одной большой системы (параллелизм по данным), либо большого числа маленьких систем (параллелизм по задачам). Параллельность второго типа использована авторами данной статьи в методе самосогласованного МД-восстановления межчастичных потенциалов (МЧП) в ионных системах по экспериментальным данным на основе ограниченной глобальной оптимизации (ОГО) [4, 5].

В настоящее время графические процессоры GPU (Graphic Processing Unit) являются оптимальной по соотношению цена–производительность параллельной архитектурой с общей памятью [6]. Например, GPU AMD Radeon HD4870x2 с тактовой частотой 0.75 ГГц, доступный на рынке по цене около 16000 руб., имеет пиковую теоретическую производительность 2400 GFLOPS (миллиардов операций плавающей арифметики в секунду), а последний 4-ядерный CPU (Central Processing Unit) Intel Core 2 Extreme QX9775 с частотой 3.2 ГГц — всего 102 GFLOPS, и продается по цене около 48000 руб. Несколько лучшим соотношением обладает процессор STI Cell BE, использующийся в игровой приставке Sony Playstation3 и обеспечивающий 154 GFLOPS при частоте 3.2 ГГц по цене 15000 руб. за приставку.

Бóльшая часть времени в МД-моделировании тратится на расчет парных взаимодействий: число пар частиц равно $N(N-1)/2$, тогда как число операций для интегрирования уравнений движения пропорционально N. Поэтому для достижения высокой производительности достаточно реализовать параллельный расчет сил на GPU.

В истории реализаций МД-расчетов на GPU можно выделить следующие важные этапы.

1. *Осень 2004 г.* Первое упоминание расчета парных межчастичных взаимодействий на GPU — доклады на конференциях ACM Workshop on General Purpose Computing on Graphics Processors 2004 и

[1] Институт математики и механики УрО РАН, ул. С. Ковалевской, д. 16, 620219, г. Екатеринбург; аспирант, e-mail:boyarchenkov@gmail.com

[2] Уральский государственный технический университет, физико-технический факультет, ул. Мира, д. 19, 620002, г. Екатеринбург; ст. преподаватель, e-mail: potashnikov@gmail.com

IEEE Visualization 2004. В [7, 8] предложена простейшая схема расчета сил со степенным потенциалом Леннарда–Джонса, в которой на GPU выполняется тело внутреннего цикла по частицам. Отсутствуют данные о производительности и применении полученной программы. Использовались язык программирования Brook и графические процессоры GPU ATI Radeon X800 XT и NVIDIA GeForce 6800.

2. *Зима 2004–2005 г.* Авторами данной статьи реализована программа для МД-моделирования оксидного ядерного топлива с расчетом сил на GPU: использованы языки программирования C#, HLSL, GPU-ассемблер, библиотека Microsoft DirectX 9.0c, GPU ATI Radeon 9600 Pro (программная модель Shader model 2.0). Достигнуто ускорение в три раза (здесь и далее будет проводиться сравнение с CPU-версией, запускаемой на одном ядре и не использующей SIMD-инструкции) для 24-битной арифметики с плавающей запятой. Так как в модели Shader model 2.0 отсутствовала поддержка циклов, то для снижения издержек при вызовах функций DirectX было реализовано разворачивание нескольких итераций цикла по частицам на GPU-ассемблере. Позднее на GPU NVIDIA GeForce 6600 была реализована версия в рамках Shader model 3.0 (32-битная арифметика), в которой достигнуто 10-кратное ускорение по сравнению с CPU [9].

3. *Осень 2005 г.* В докладе Харриса [6] рассмотрен алгоритм расчета сил, в котором каждое парное взаимодействие сохраняется в отдельном пикселе текстуры, после чего в отдельном ядре производится суммирование парных взаимодействий. Автором заявлена производительность 17 GFLOPS (21% теоретического максимума в 83 GFLOPS) на NVIDIA GeForce 7800 GTX в случае гравитационного взаимодействия. Число частиц, обрабатываемых за один вызов GPU, в данной реализации равно 2048 из-за ограничений на максимальные размеры текстур в Shader model 3.0, а скорость расчетов ограничена операциями с текстурой сил, расположенной в общей памяти, — одно только количество записей в нее (с учетом суммирования) равно $N(2N-1)$.

4. *Лето 2006 г.* Авторами данной статьи предложен метод МД-восстановления на GPU межчастичных потенциалов по экспериментальным данным. Реализован метод суммирования Эвальда для периодических граничных условий [10]. Локальная оптимизация — методами Nelder–Mead и PRAXIS без вычисления производных, глобальная — random restart. До 25 раз быстрее, чем CPU AMD Athlon64 2 ГГц, на GPU ATI Radeon 1900 XT для системы из 768 частиц [4]. Для МД-моделирования в НГУ предложена версия, использующая упорядоченность (сортировку) частиц по типам, в 80 раз быстрее, чем CPU-реализация [11].

5. *Декабрь 2006 г.* В [12] предложена версия с использованием режима Multiple Render Targets (MRT, в рамках Shader model 3.0) для более эффективной загрузки шины памяти и 4-векторных конвейеров GPU; реализована в рамках проекта Folding@Home. Получено около 78 GFLOPS (31% теоретического максимума в 250 GFLOPS, в случае степенного МЧП) на GPU ATI Radeon 1900 XTX — в 25 раз быстрее, чем SSE-оптимизированная (Streaming SSE Extensions) версия на Intel Pentium 4 3.0ГГц. Количество частиц в системе равно целой степени двойки, что позволяло эффективно загрузить все вычислители GPU.

6. *Январь 2007 г.* Авторами настоящей статьи реализована программа, использующая режим MRT и сортировку по частицам. Достигнуто 133-кратное ускорение по сравнению с CPU AMD Athlon64 2 ГГц; на ее основе в течение года было проведено МД-моделирование процессов переноса в нанокристаллах с размерами 10–100 тысяч частиц на временны́х интервалах в 1–2 микросекунды [13].

7. *Март 2007 г.* Гравитационная динамика [14] на CUDA, 256 GFLOPS (74% теоретического максимума в 346 GFLOPS) на NVIDIA GeForce 8800 GTX (131072 частиц) для варианта с расчетом на GPU только сил.

8. *Июль 2007 г.* Гравитационная динамика [15] на CUDA, 340 GFLOPS (98% теоретического максимума) на GeForce 8800 GTX (131072 частиц) — только расчет сил.

9. *Сентябрь 2007 г.* Авторами настоящей статьи реализован метод суммирования Эвальда с использованием преимуществ платформы CUDA.

10. *Март 2008 г.* В работе [16] нами предложена версия с использованием параллелизма по задачам и распределенных вычислений для решения задачи восстановления потенциалов методами ограниченной глобальной оптимизации.

В настоящей работе мы подробно рассмотрим следующие моменты реализации МД на GPU с использованием платформы NVIDIA CUDA:

— различные реализации треугольного цикла по частицам, сокращающего количество вычислений в два раза за счет учета третьего закона Ньютона;

— детали оптимизации кода;

— неадекватность оценки производительности в GFLOPS;

— вопрос выбора точности плавающей арифметики на разных стадиях расчета.

2. Сравнение архитектуры центральных и графических процессоров. Центральные процессоры (CPU) разработаны для самого широкого круга задач и ограниченно используют возможности параллельных вычислений. Увеличение их производительности в основном связано с увеличением тактовой частоты и размеров высокоскоростной кэш-памяти. Программирование на CPU для ресурсоемких научных вычислений подразумевает тщательное структурирование данных и порядка инструкций для эффективного использования всех уровней кэш-памяти. Напротив, графические процессоры (GPU) изначально спроектированы для параллельной обработки данных (например, закрашивания массива пикселей), когда обработка отдельных элементов может производиться независимо. GPU являются примером потокового процессора, который использует явный *параллелизм по данным* для увеличения скорости вычислений и уменьшения зависимости от задержек доступа к памяти. За последние восемь лет GPU эволюционировали от устройств с фиксированной функциональностью для ускорения примитивных графических операций до программируемых процессоров, превосходящих обычные CPU при выполнении векторизуемых операций [6, 17].

В концепции потоковых вычислений данные представляются в виде потоков из независимых элементов, а независимые стадии обработки (наборы операций) — в виде ядер. Ядра можно представлять как функции преобразования элементов входных потоков в элементы выходных потоков. Такое представление позволяет параллельно применять ядро сразу ко многим элементам входного потока. Вместо подгонки *формата* данных под архитектуру кэшей каждого конкретного CPU, данные разбиваются на потоки, разные по *содержанию* и способу обработки. Предсказуемость потокового доступа к памяти позволяет выполнять его параллельно с вычислениями.

В табл. 1 представлено краткое сравнение архитектуры CPU и GPU.

Таблица 1

CPU	GPU
Память оптимизирована под минимальную латентность. Много транзисторов “управления” (предсказание ветвлений, планировщики и пр.) + “кэши” = мало “вычислителей”. Архитектура оптимизирована для программ со сложным потоком управления (максимальная интерактивность).	Память оптимизирована под максимальную пропускную способность. Бóльшая часть транзисторов является вычислителями. Архитектура оптимизирована для программ с большим объемом вычислений (максимальная скорость вычислений). Латентность скрывается вычислениями во время запросов к памяти (за счет потоковой обработки). Управляющие блоки разделяются между вычислителями (обработка ветвлений менее эффективна).

В табл. 2 обозначены тенденции, рассчитанные нами по данным о CPU производства Intel (с 1999 г.) и GPU производства NVIDIA и ATI (с 2002 г.) [18].

Таблица 2

Число транзисторов: удвоение каждые 17 месяцев (+60% за год).	
Технологический процесс: уменьшение в два раза каждые 48 месяцев (−16% за год).	
CPU	GPU
МТП: удвоение каждые 24 месяца (+42% за год). ПСП: удвоение каждые 26 месяцев (+38% за год).	МТП: удвоение каждые 10 месяцев (+126% за год) ПСП: удвоение каждые 19 месяцев (+55% за год).

Здесь МТП — максимальная теоретическая производительность (увеличивается с количеством скалярных вычислительных блоков и их частотой), ПСП — пропускная способность памяти (увеличивается

с шириной шины и ее частотой).

3. Особенности нашей реализации МД-метода на графических процессорах. Рассмотрим динамику системы заряженных частиц, в которой каждая частица электростатически взаимодействует со всеми остальными, а результирующие силы — это суперпозиция $N(N-1)$ независимых парных взаимодействий.

В данной работе мы рассматриваем прямой расчет сил в НГУ, а в следующей [16] — суммирование Эвальда в ПГУ, так как по сравнению с приближенными иерархическими и сеточными методами (Treecode, Fast Multipole Method и Particle–Mesh [3]) они являются точными, а простота их алгоритмов обеспечивает высокую эффективность параллельной реализации на GPU. На системах с десятками тысяч частиц точные методы также оказываются быстрее приближенных, которые хотя и имеют меньшую теоретическую вычислительную асимптотику $O(N\log(N))$ и $O(N)$, но все же проигрывают в количестве операций на частицу. Кроме того, иерархические и сеточные методы для расчета взаимодействий между ближайшими соседями также используют прямой расчет или суммирование Эвальда.

Отличия нашей задачи МД-моделирования от работ других авторов заключаются в следующем:

— система состоит из частиц нескольких типов, взаимодействие между которыми включает в себя короткодействие их электронных оболочек и определяется несколькими коэффициентами (в астрофизических задачах частицы характеризуются одним коэффициентом — массой);

— число частиц кратно 12 (это число частиц в элементарной ячейке моделируемого кристалла), а в работах других авторов тестировались системы с числом частиц, равным целой степени двойки.

В отличие от астрофизической гравитационной динамики, в которой представляют интерес траектории отдельных тел и для их более точного интегрирования дополнительно рассчитывается производная третьего порядка $\dfrac{d^3x}{dt^3}$, в МД-моделировании интерес представляют лишь макропараметры, поэтому интегрирование обычно выполняется простейшими схемами второго порядка (Верлет [20], Биман и др.). Однако если сложение парных сил и интегрирование выполняются с одинарной точностью плавающей арифметики, то для компенсации накапливающихся погрешностей приходится корректировать общие импульс, момент и энергию системы для выполнения законов сохранения.

Наши исследования точности расчетов (погрешности на одном шаге, накопления импульса, момента импульса и энергии) показали, что доступная на GPU одинарная точность достаточна для расчета сил и их суммирования, тогда как интегрирование лучше проводить с двойной точностью. К аналогичным выводам пришли разработчики серии специализированных сопроцессоров MDGRAPE, которые на разных этапах конвейера выполняют вычисления с разной точностью [15]. Тем не менее, в последнем поколении графических процессоров, доступных с июня 2008 г., имеется аппаратная поддержка вычислений с 64-битной точностью (стандарт IEEE 754R) [21].

Следующим важным моментом является выбор короткодействующих парных потенциалов (ПП), аппроксимирующих взаимодействие частиц на расстояниях порядка радиуса внешних электронных оболочек ионов, так как при молекулярно-динамическом моделировании в приближении парных взаимодействий и достаточной точности МД-интегрирования все структурные и динамические свойства моделируемой системы полностью определяются этими потенциалами. Наиболее распространенные формы короткодействующих ПП — степенные потенциалы Леннарда–Джонса и экспоненциальные Борна–Майера. Степенные ПП обладают преимуществом в скорости расчетов за счет замены целочисленных степеней R (расстояния между частицами) операциями умножения и отсутствия положительной степени R, использующейся под экспонентой потенциалов Борна–Майера, для которой требуется операция деления и дополнительный регистр. Параметризацию ПП можно проводить на основе квантово-механических расчетов или по известным экспериментальным данным (эмпирические парные потенциалы — ЭПП). Наилучших результатов удалось достичь, используя самосогласованное МД-восстановление ПП по термодинамическим экспериментальным данным в ПГУ на системах из 300–800 частиц [4].

Наши реализации МД на GPU с применением библиотеки функций DirectX [22] показали, что на системах такого размера бóльшая часть времени расчетов уходит на издержки, связанные с вызовами функций DirectX. Кроме того, наблюдалась сильная зависимость времени расчетов от формата представления данных (размеров текстур), т.е. кэш использовался неэффективно. Поэтому для ускорения расчетов было решено переписать вычислительное ядро с применением технологии NVIDIA CUDA, которая на несколько порядков уменьшает издержки при доступе к GPU и позволяет управлять доступом к кэшу. Соответственно, наибольший выигрыш от применения CUDA получается при моделировании в ПГУ с небольшим (порядка 1000 частиц) размером транслируемой области (см. подробности в работе [16]).

4. Технология Compute Unified Device Architecture (CUDA). CUDA — это технология па-

раллельных вычислений, позволяющая разрабатывать программы для GPU на стандартном языке программирования Си с несколькими расширениями, причем производительность этих программ хорошо масштабируется с числом ядер. Она разработана компанией NVIDIA и в настоящий момент работает только с ее GPU, начиная с поколения G8x (G80, G84, G86, и т.д.): серии GeForce 8x00, Quadro 4600/5600 и Tesla. Ее ключевые абстракции [19] — иерархия сгруппированных потоков обработки, программируемый кэш (разделяемая память) и барьерная синхронизация.

Преимущества технологии CUDA перед использованием стандартных графических библиотек DirectX и OpenGL:

— программирование на широко известном языке Си с несколькими простыми расширениями;

— произвольная адресация при записи в память;

— доступ к программируемой разделяемой памяти;

— значительно меньшие накладные расходы на взаимодействие CPU и GPU, некоторые операции выполняются асинхронно;

— побитовые операции над целыми числами, отсутствующие в шейдерной модели SM 3.0;

— двойная точность операций с плавающей запятой [3], отсутствующая в текущей шейдерной модели SM 4.0.

Недостатки текущей реализации CUDA по сравнению с программной моделью CPU:

— примитивные структуры данных: реализована поддержка только массивов (нет стека, кучи, списков и т.д.);

— нет механизмов абстракций: средств объектно-ориентированного и функционального программирования;

— нет управления обработкой с GPU (в коде ядра нельзя запускать и останавливать ядра, инициировать обмен данными с CPU);

— обработка параллельной записи по одному адресу только для некоторых целочисленных операций;

— отсутствие поддержки рекурсивных функций;

— в настоящий момент поддерживаются только две аппаратные архитектуры: GPU от NVIDIA и x86 CPU.

5. Архитектура GPU поколения G8x. GPU компании NVIDIA поколения G8x [18] содержат от 16 до 128 скалярных процессоров, каждый имеет три скалярных арифметико-логических устройства (АЛУ) плавающей арифметики одинарной точности (для одновременного выполнения операции mad (Multiply–Add) вида $xy+z$ с операцией сложения или умножения). На GPU GeForce 8800 GTX эти процессоры работают с частотой 1.35 ГГц, поэтому его теоретическая пиковая производительность составляет 518 GFLOPS ($128 \times 3 \times 1.35$ ГГц). Однако ситуации, в которых задействуются одновременно 3 АЛУ, крайне редки, поэтому при расчетах пиковой производительности количество одновременных операций считается равным двум, как и для CPU (т.е. $128 \times 2 \times 1.35 = 346$ GFLOPS).

Скалярные процессоры объединены по 8 в SIMD-мультипроцессоры (МП), обладающие также программируемым кэшем, объем которого равен 16 КБ, — это *разделяемая* (между потоками в одной связке) *память*. Таким образом, связка не может выполняться на нескольких МП, так как у них независимая *разделяемая память*, но в некоторых случаях две связки могут одновременно выполняться на одном МП (если выполняются ограничения, указанные ниже).

GPU поколения G8x имеют следующие ограничения:

— максимум 512 параллельных потоков инструкций на связку и 768 на один МП;

— максимум 8192 32-битных регистра на все потоки, выполняемые на одном МП;

— максимум 2 миллиона ассемблерных инструкций на ядро;

— максимальный объем передаваемых параметров равен 256 байтам на ядро.

Если в нескольких параллельных потоках одновременно производится запись в одну ячейку памяти, то происходит так называемый “конфликт записи”; в этом случае сериализуются (упорядочиваются для последовательного выполнения) только специальные атомарные инструкции (в текущей версии — это лишь некоторые целочисленные), для прочих операций гарантируется только то, что как минимум одна из них произведет запись успешно. В обоих случаях порядок выполнения операций не определен.

Результаты по скорости расчетов получены на следующей программно-аппаратной конфигурации:

— AMD Athlon64 X2 4000+ (2100 МГц), 3 ГБ DDR2 (667 МГц);

[3] Аппаратная поддержка, начиная с поколения чипов ATI Radeon 4xxx и NVIDIA GeForce GTX 2xx. Отметим, что область применения CUDA выходит за рамки GPU от NVIDIA, так как существуют компиляторы CUDA-кода для CPU, а также при поддержке NVIDIA ведутся разработки компилятора для GPU от ATI, однако плавающая арифметика GPU любой точности не полностью соответствует стандарту IEEE 754 (подробности см. в документации по CUDA [19]).

— NVIDIA GeForce 8800 GTX (1350 МГц — частота скалярных процессоров), 768 MB 384-бит GDDR3 (1800 МГц);
— Microsoft Windows XP + Service Pack 2;
— CPU Driver AMD 1.3.2.0053 (09/2007);
— GPU Driver NVIDIA Forceware 177.84 (08/2008);
— Microsoft .NET Framework 3.5;
— Microsoft DirectX 9.0c (4.09.0000.0904, 08/2007);
— NVIDIA CUDA 2.0 (Toolkit + SDK);
— Microsoft 32-bit C/C++ compiler 14.0 (в составе Microsoft Visual Studio 2005).

6. Алгоритмы расчета парных взаимодействий.

6.1. Расчет сил при нулевых граничных условиях на CPU. Простейший цикл для расчета сил выглядит так:

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    force[i] = force[i] + PairInteraction(i, j);
```

Этот же расчет в случае степенного короткодействующего потенциала и с учетом третьего закона Ньютона (о равенстве действия противодействию) можно записать в следующей форме (на языке программирования C++):

```
void PairInteraction(int i, int j, double c[])
{
  double3 R = pos[i] - pos[j];
  double r = 1 / sqrt(R * R);
  double f = c[0] * r * r * r + pow(c[1] * r, c[2]); // f = |Fij|/|R|
  force[i] += f * R; force[j] -= f * R;
}
for (int i = 0; i < N; i++)
  for (int j = i + 1; j < N; j++)
    PairInteraction(i, j, coefs[type[i]][type[j]]);
```

Здесь N — число частиц, массив pos содержит координаты частиц, массив type — типы частиц, массив coefs — коэффициенты парного потенциала в зависимости от типов частиц, в массив force записываются вычисленные силы, действующие на каждую частицу.

В первом примере внешний и внутренний циклы пробегали по всем частицам, поэтому рассчитываемые парные взаимодействия можно представить в виде "квадратного массива" размера $N \times N$; такой вариант двойного цикла назовем *квадратным*. Во втором случае учет равенства действия (i, j) противодействию (j, i) позволяет рассчитывать только "треугольник" справа (или слева) от диагонали квадрата с уменьшением объема вычислений почти в два раза; такой вариант назовем *треугольным циклом*.

6.2. Расчет сил при нулевых граничных условиях на GPU. Так как положения всех тел фиксированы во время расчета сил, расчет можно выполнять параллельно для разных итераций цикла по i. В представлении потоков и ядер это выражается следующим образом:

```
Stream ions, forces
Kernel compute_force(ion_i)
{
  for each ion_j in ions
    force_i = force_i + PairInteraction(ion_i, ion_j)
  return force_i
}
forces = compute_force(positions);
```

Ядро compute_force применяется к каждому элементу потока ions и производит на его основе элемент потока forces. Отметим, что в ядре compute_force есть индексированная выборка из потока ions в цикле по j. В общем случае выборка по произвольному адресу может быть медленной, так как не будет работать механизм предвыборки. Однако в нашем случае выборка последовательная, т.е. может эффективно кэшироваться. Более того, j-цикл выполняется параллельно для многих i-элементов; поэтому даже в случае с минимальным кэшированием, когда каждое значение ion_j используется для нескольких i-элементов без многократной выборки из общей памяти, от такого подхода ожидается высокая производительность.

Однако при поточно-параллельном подходе в применении к задаче N тел нельзя эффективно использовать третий закон Ньютона (требуется медленное сохранение сил каждой пары частиц в общую память и дополнительный цикл их суммирования); поэтому для каждой пары частиц силы вычисляются дважды.

В реализации на GPU внешний i-цикл заменяется параллельным выполнением в разных потоках. Из-за ограничения на число потоков в одной связке, а также для использования всех имеющихся мультипроцессоров GPU потоки приходится разбивать на несколько связок. Поэтому индекс i вычисляется с учетом индекса потока в связке и индекса связки. Так как внутренний цикл пробегает по всем частицам, для предотвращения деления на нуль необходимо проверять равенство индексов i и j. В приведенном ниже коде используются также некоторые стандартные векторные операции, которые нетрудно определить с использованием стандартного синтаксиса C++ (отметим, что это расширение стандарта языка Си не документировано в CUDA SDK (Software Development Kit)) и модификатора `__device__`; при этом первый параметр оператора += необходимо объявить в виде ссылки.

```
__device__ float3 force_ij(float3 pos_i, float3 pos_j, float3 c)
{
  float3 R = pos_i - pos_j;
  float r = rsqrt(R * R);
  return R * (c.x * r * r * r + pow(c.y * r, c.z));
}
__global__ void kernel_NxN(float3 force[], float3 pos[], int type[], float3 coefs[],
int N, int types)
{
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  for (int j = 0; j < N; j++)
    if (i != j)
      force[i] += force_ij(pos[i], pos[j], coefs[type[i] * types + type[j]]);
}
```

Даже такая простая версия работает на 75% быстрее CPU-версии (при N = 6144). Возможны следующие оптимизации с целью повышения производительности.

1. Сокращение работы с общей (медленной) памятью.

   а) Кэширование доступа force[i], pos[i], type[i] регистрами.

   б) Кэширование доступа к pos[j], type[j] посредством разделяемой памяти (shared memory). Так как объем разделяемой памяти сильно ограничен, в каждом потоке приходится обрабатывать только часть частиц, которая помещается в разделяемую память, поэтому для обработки всех частиц ядром необходим дополнительный цикл с синхронизацией потоков между операциями записи в разделяемую память координат и типов частиц и их чтения при расчете парной силы.

   в) Передача данных посредством векторного типа float4 позволяет задействовать механизм объединенного доступа к памяти (memory access coalescing) [19].

2. Сокращение числа ветвлений.

   а) Вместо применения условного оператора `if (i != j)` можно операцию `rsqrt(R*R)` заменить на `rsqrt(max(R*R, 1e-4))`, в которой затраты на выполнение операции max значительно ниже затрат на ветвление. В этом случае при i = j (самодействие) операция rsqrt не вызовет ошибки деления на нуль, а вклад в силы будет нулевым из-за умножения на нуль-вектор межчастичного расстояния. Данный прием не влияет на точность вычислений и быстрее (примерно на 5%) по сравнению с распространенным приемом добавления небольшого числа к знаменателю.

   б) Разворачивание циклов, которое приводит к уменьшению числа проверки условия `j < N` и допускает повторное использование одинаковых параметров парных потенциалов, если частицы отсортированы по типам. Количества частиц каждого типа не должны быть взаимно простыми.

3. Оптимизация под архитектуру CUDA и конкретную аппаратную реализацию.

   а) Оптимальная загрузка имеющихся мультипроцессоров:

— Для небольших систем (когда число потоков меньше числа скалярных процессоров GPU) выгодно распределять итерации j-цикла для i-й частицы на несколько потоков, при этом необходимо устранять конфликты записи.

— Большее число итераций j-цикла в каждом потоке соответствует меньшему числу обращений к GPU, но это число ограничено объемом разделяемой памяти.

— У чипов G8x и G9x число потоков в связке равно 512, однако каждый мультипроцессор может обрабатывать до 768 потоков, поэтому если в каждой связке будет не более 384 потоков, то мультипроцессор сможет обрабатывать две связки одновременно.

— Число регистров на мультипроцессор равно 8192, регистры распределяются между потоками; если потокам не хватает регистров, то компилятор задействует медленную общую память, тогда число потоков следует уменьшить.

б) Использование более быстрых вариантов математических операций через опцию “use_fast_math” компилятора CUDA.

Таблица 3

Замедление самой быстрой версии при отключении оптимизаций (N = 6144)

Тип отключаемой оптимизации	Время на шаг, сек	Замедление, разы
Кэширование регистрами	0.01776	4.17
Кэширование разделяемой памятью	0.04085	9.59
float4 для объединенного доступа к памяти	0.00441	1.03
Исключение самодействия с помощью `max`	0.00831	1.95
Разворачивание итераций j-цикла	0.00582	1.37
Опция “use_fast_math”	0.02693	6.32
Самая быстрая версия	0.00426	1.00

Таблица 3 позволяет оценить значимость некоторых оптимизаций. Видно, что наибольшее влияние на скорость расчетов оказывает уменьшение числа операций с медленной памятью (за счет кэширования) и опция компилятора “use_fast_math”. Отметим, что при отключении опции “use_fast_math” программа потребовала больше регистров, что при измерениях для данной таблицы было скомпенсировано меньшим числом развернутых итераций j-цикла.

Моделируемые системы представляют собой кубический кристалл, состоящий из кубических элементарных ячеек, каждая ячейка — из 12-и частиц двух типов (соотношение числа частиц разных типов 2:1). К примеру, система из 6144 частиц состоит из $8 \times 8 \times 8$ элементарных ячеек.

Наименьшее ускорение (см. табл. 4) получилось на системе из 324 частиц, так как не удалось эффективно задействовать все скалярные процессоры. Небольшое снижение производительности наблюдается также на системах, число частиц в которых делится на небольшую степень двойки ($4116 = 12 \times 7^3$ и $8748 = 12 \times 9^3$), а наибольшая производительность — на системе из $49152 = 3 \times 2^{14}$ частиц. Это связано с полной загрузкой вычислительных блоков архитектуры GeForce 8800 GTX, которые сгруппированы по степеням двойки (16 мультипроцессоров, 8 скалярных процессоров в мультипроцессоре). Расчет взаимодействий на CPU был реализован на языке C++ (код приведен выше), тогда как интегрирование системы и пользовательский интерфейс реализованы на языке C#.

При расчетах на CPU часто используется прием с обрезанием короткодействующей составляющей парного потенциала на расстояниях порядка нескольких периодов решетки кристалла, сокращающий объем вычислений при увеличении размеров системы. На GPU затраты на обработку условия превосходят выигрыш от меньшего объема вычислений, поэтому обрезание не использовалось, и во всех таблицах время для CPU и GPU приведено в одинаковых условиях — без обрезания.

Ниже приведен код наиболее быстрой версии, прием с разворачиванием итераций цикла для краткости опущен.

```
__device__ float4 force_ij(float4 pos_i, float4 pos_j, float4 c)
{
```

Таблица 4

Время на шаг для систем с различным числом частиц на CPU и GPU

Число частиц	Время на шаг GPU, сек	Время на шаг CPU, сек	Время без расчета сил, сек	Время на одно парное взаимодействие, 10^{-10} сек	Ускорение GPU / CPU, разы
324	0.000048	0.0075	0.000249	4.572	157.1
768	0.000087	0.0427	0.000524	1.475	490.9
1500	0.000292	0.1620	0.000964	1.298	554.7
2592	0.000818	0.4746	0.001421	1.218	580.2
4116	0.002460	1.2040	0.002066	1.452	489.4
6144	0.004262	2.6932	0.003179	1.129	631.9
8748	0.010727	5.4465	0.004453	1.402	507.7
12000	0.017142	10.3540	0.006124	1.190	604.0
20736	0.050942	31.4460	0.010740	1.185	617.3
32928	0.127638	79.8280	0.017183	1.177	625.4
49152	0.267812	177.3300	0.026184	1.109	662.1
69984	0.577770	360.4800	0.038079	1.180	623.9
96000	1.050462	677.3800	0.051684	1.140	644.8

```
  float4 R = pos_i - pos_j; R.w = rsqrt(max(R * R, 1e-4));
  return R * (c.x * R.w * R.w * R.w + pow(c.y * R.w, c.z));
}
__global__ void kernel_NxN(float4 force[], float4 pos[], int type[], int types,
float4 coefs[])
{
  int i = threadIdx.x + blockIdx.x * blockDim.x, type_i = type[i];
  float4 pos_i = pos[i], force_i = { 0, 0, 0, 0 };

  __shared__ float4 pos_j[N_shared];
  __shared__ int type_j[N_shared];
  __shared__ float4 coefs_ij[4];
  if (threadIdx.x < 4) coefs_ij[threadIdx.x] = coefs[threadIdx.x];

  for (int J = 0; J < N; J += N_shared * BJ)
  {
    int j = threadIdx.x * threads_mem_ratio, j0 = blockIdx.y * N_shared + J;
    if (j < N_shared) for (int k = 0; k < threads_mem_ratio; k++, j++)
    {
      pos_j[j] = pos[j + j0];
      type_j[j] = type[j + j0];
    }
    __syncthreads();

    for (j = 0; j < N_shared; j++)
      force_i += force_ij(pos_i, pos_j[j], coefs_ij[type_i * types + type_j[j]]);
  }

  force[i + blockIdx.y * N] = force_i;
```

```
}
// вызов ядра:
kernel_NxN <<< dim3(BI, BJ), N / BI >>>(force, pos, type, 2, coefs);
```

BI — число блоков, на которые разбивается цикл по i для загрузки большего числа мультипроцессоров в случае малого числа частиц или для разбиения числа частиц, превышающего максимальное число потоков в связке (N / BI $<$ 512 для GPU поколения G8x).

BJ — число блоков, на которые разбивается цикл по j для увеличения общего числа потоков при наличии незагруженных мультипроцессоров (при BJ $>$ 1 необходимо устранить конфликты записи, для этого выделяем массив force размером N$\times$BJ, а не N).

N_shared — число частиц, помещающихся в разделяемую память; является делителем числа (N / BJ).

threads_mem_ratio — отношение числа N_shared к числу потоков в связке (N / BI); используется для заполнения разделяемой памяти каждым потоком.

Переменная J пробегает по всем частицам с учетом того, что одним из BJ потоков за один раз обрабатывается N_shared частиц, взаимодействующих с i-й частицей.

j0 — абсолютный индекс первой из N_shared частиц, записываемых в разделяемую память.

j — относительный индекс частиц в разделяемой памяти.

Рассмотрим для примера систему из N = 2592 частиц. Нами экспериментально выбраны следующие оптимальные значения констант: BI = 8, BJ = 2, N_shared = N/BJ/2 = 648, threads_mem_ratio = N_shared/(N/BI) = 2. При этом каждый из 16 мультипроцессоров GPU GeForce 8800 GTX выполняет связку из N/BI = 324 потоков, каждый из которых может использовать не больше 20 регистров, так как общее число регистров 8192 и количество потоков при распределении регистров округляется с шагом 64, а количество регистров — с шагом 4. Размер используемой разделяемой памяти: `4*8+sizeof(int) = 36` байт на параметры ядра (по 8 байт на массив), `(sizeof(float4)+sizeof(int))*N_Shared = 12960` байт на массивы pos_j и type_j, `sizeof(float4)*4 = 64` байта на массив coefs_ij, итого 13060 байта из 16384 доступных. Оптимальное количество развернутых итераций внутреннего цикла по j равно N_shared/6=108.

Данная версия быстрее предложенной в работе [15], в которой каждая связка состоит из 128 потоков, так как при большем числе потоков в связке уменьшается количество операций с памятью, приходящихся на один поток. К примеру, системы из 324 и 768 частиц будут рассчитаны за 4 и 6 вызовов по 128 частиц и за 1 и 2 вызова по 384, а каждый вызов включает заполнение разделяемой памяти и синхронизацию всех мультипроцессоров. Если 128 не является делителем числа частиц, то в ядро придется добавить условие для обработки выхода за границу массивов pos и force, что также замедлит выполнение (см. реализацию условий на GPU в [19]).

6.3. Комментарии к сравнению производительности в GFLOPS. В статьях по астрофизике принято считать, что расчет гравитационных сил между одной парой тел требует 38 FLOP (операций плавающей арифметики) [15], что больше чем количество реальных арифметических действий. В работе [23] можно найти пояснение: каждая операция деления и извлечения квадратного корня заменяется эффективным числом 10 FLOP; отмечается, что это число соответствует реальной вычислительной стоимости данных операций на типичных скалярных процессорах, однако не приводится никаких ссылок на происхождение этих 10 FLOP.

Если считать GFLOPS описанным выше способом, то каждое парное взаимодействие в приведенном нами коде требует 50 FLOP (ресурсоемкую функцию pow тоже будем считать за 10 FLOP). Тогда для GPU GeForce 8800 GTX на системе из 49152 частиц мы из табл. 4 получим производительность в 451 GFLOPS, что на 30% больше теоретических 346 GFLOPS, поэтому такая оценка некорректна.

В документации по SIMD-расширению процессора Intel Pentium III приведены следующие значения характеристики throughput (пропускной способности конвейера) для арифметики одинарной точности [24]: сложение и умножение выполняются за 2 такта, деление — 18–36, извлечение квадратного корня — 29–58. Однако при расчете сил можно использовать операцию rsqrt (reciprocal square root), эта SSE-команда требует всего 4 такта (как два умножения), хотя и имеет меньшую точность (которую можно улучшить, выполнив одну итерацию метода Ньютона–Рафсона [25]). В [23] при расчете парных взаимодействий используется именно rsqrt. Учитывая эти данные, с использованием операции rsqrt расчет сил между одной парой тел можно оценить в 20 FLOP (заменив 10 FLOP на деление и 10 на квадратный корень двумя на операцию rsqrt). Следовательно, все измерения производительности на CPU, приводимые в работах других авторов (например, в [23]), следует разделить на 1.9. Для программ, написанных для GPU, необходимы другие оценки.

В документации CUDA Programming Guide [19] в разделе 5.1.1 (Instruction throughput) заявлены следующие расходы на операции: сложение и умножение — 4 такта, деление — 36 тактов (имеется более

быстрая реализация __fdividef, требующая всего 20 тактов), rsqrt — 16 тактов. В нашей работе используется быстрый вариант операции возведения в степень pow, реализованный следующим образом (см. таблицу B-3 в [19]): `exp2f(y * __log2f(x))`, поэтому ее можно оценить в $32 + 4 + 16 = 52$ такта (как 13 умножений).

На GPU имеется также аппаратная реализация операции mad вида $xy + z$, стоимость которой равна стоимости одного умножения или сложения. Однако в зависимости от порядка операций в CUDA-коде и версии компилятора она может задействоваться (увеличивая плотность арифметических действий на такт) или не задействоваться. Еще труднее оценить, как часто будут выполняться три арифметических действия за такт.

Поэтому более правильно сравнивать различные МД-реализации (в том числе на разных программно-аппаратных платформах) по времени на шаг для одинаковых систем или по времени, которое тратится на одно парное взаимодействие. Так, для систем из 32768 и 65536 частиц реализация из работы [15] тратит 1.1443×10^{-10} и 1.1173×10^{-10} секунд на расчет парного взаимодействия соответственно, а наша реализация для системы из $49152 = (32768 + 65536)/2$ частиц — 1.1085×10^{-10} секунд, т.е. на 2% быстрее среднего результата $(1.1443 \times 10^{-10} + 1.1173 \times 10^{-10})/2$. Однако в нашем случае за это время выполняется больше вычислений, так как дополнительно рассчитывается степенной короткодействующий потенциал.

6.4. Реализация треугольного цикла на GPU. Рассмотрим возможные реализации треугольного цикла на CUDA, для простоты и снижения расходов на коммуникации будем использовать только один мультипроцессор.

6.4.1. Версия 1. В простейшей версии результат, полученный за один вызов Fij, записывается в два элемента квадратного массива, а суммирование производится в отдельном ядре.

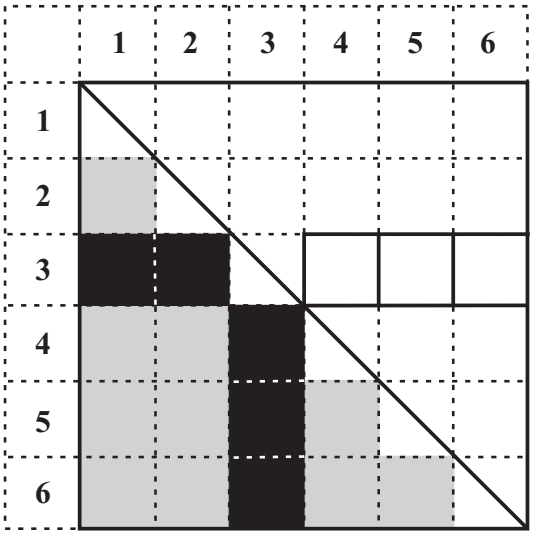
```
__global__ void kernel_NxN(float4 force[], float4 pos[], int type[], float4 coefs[],
int N, int types)
{
  int i = threadIdx.x;
  for (j = i + 1; j < N; j++)
  {
    float4 f = force_ij(pos[i], pos[j], coefs[type[i] * types + type[j]]);
    force[j * N + i] = f; force[i * N + j] = -f;
  }
  force[i * N + i] = make_float3(0, 0, 0, 0);
}
__global__ void kernel_Sum(float4 force[], int N)
{
  float4 force_sum = { 0, 0, 0, 0 };
  for (int i = threadIdx.x; i < N * N; i += N) force_sum += force[i];
  force[threadIdx.x] = force_sum;
}
```

В этой реализации (а также и в последующей) невозможно использовать разворачивание циклов, так как количество итераций в цикле вида "`j>i`" различно для разных потоков, а исключение самодействия частиц происходит автоматически. Оптимизации, связанные с кэшированием регистрами и разделяемой памятью, для наглядности опущены. Узким местом является медленный доступ к глобальной памяти, который производится в каждой итерации циклов обоих ядер.

6.4.2. Версия 2. Можно сократить количество обращений к памяти, записывая взаимодействие Fij "треугольно", т.е. только один раз — в ячейку [i][j] или в ячейку [j][i], но в этом случае необходимо суммировать взаимодействия j-й частицы с помощью двух циклов:

— сначала суммируем элементы i-й строки нижней треугольной части квадратного массива;

— затем вместо i-й строки незаполненной верхней треугольной части суммируем элементы i-го столбца нижней



Суммирование сил во второй версии с треугольным циклом

треугольной части (с противоположным знаком).

На рисунке иллюстрируется такое суммирование на системе из шести частиц: строки соответствуют индексу i, столбцы — индексу j, левая нижняя треугольная область обозначает вычисляемые парные взаимодействия. При расчете силы, действующей на третью частицу, в первом цикле суммируются пары (3,1) и (3,2), затем во втором цикле вычитаются пары (4,3), (5,3) и (6,3). На рисунке перечисленные пары обозначены черными квадратами.

```
__global__ void kernel_NxN(float4 force[], float4 pos[], int type[], float4 coefs[],
int N, int types)
{
  int i = threadIdx.x;
  for (j = i + 1; j < N; j++)
    force[j * N + i] = force_ij(pos[i], pos[j], coefs[type[i] * types + type[j]]);
}
__global__ void kernel_Sum(float4 force[], int N)
{
  float4 force_sum = { 0, 0, 0, 0 };
  for (int i = threadIdx.x + N * (threadIdx.x + 1); i < N * N; i += N)
    force_sum += force[i];
  for (int i = 0; i < threadIdx.x; i++)
    force_sum -= force[N * threadIdx.x + i];
  force[threadIdx.x] = force_sum;
}
```

Первый цикл суммирования можно выполнять сразу при расчете парных взаимодействий, что позволяет уменьшить число обращений к медленной общей памяти. Второе суммирование можно выполнить в первом ядре после синхронизации потоков, повторно используя переменные i и force_i.

```
__global__ void kernel_NxN(float4 force[], float4 pos[], int type[],
float4 coefs[], int types)
{
  int i = threadIdx.x, j = threadIdx.x * BI, type_i = type[i];
  float4 pos_i = pos[i], force_i = make_float4(0, 0, 0, 0);

  __shared__ float4 pos_j[N];
  __shared__ int type_j[N];
  __shared__ float4 coefs_ij[4];
  if (i < 4) coefs_ij[i] = coefs[i];
  pos_j[i] = pos[i];
  type_j[i] = type[i];
  __syncthreads();

  for (j = i + 1; j < N; j++)
  {
    float4 f = force_ij(pos_i, pos_j[j], coefs_ij[type_i * types + type_j[j]]);
    force[j * N + i] = make_float4(f.x, f.y, f.z, 0);
    force_i += f;
  }
  __syncthreads();
  for (j = 0; j < i; j++) force_i -= force[i * N + j];
  force[i] = force_i;
}
```

6.4.3. Версия 3. С появлением в поколении GPU NVIDIA G8x программируемой разделяемой памяти стал возможен еще один вариант реализации треугольного цикла. В нем каждый i-й поток суммирует парные взаимодействия i-й частицы и записывает их в разделяемую память, а в конце сохраняет одну из накопленных сумм в линейный массив force (каждой частице соответствует только один элемент), расположенный в глобальной памяти. Это позволяет минимизировать количество обращений к медленной общей памяти.

В треугольном цикле каждый поток на каждой итерации записывает результат парного взаимодействия в свой элемент массива force_j и конфликта записи не возникает при условии, что потоки производят запись одновременно. Для этого на каждой итерации цикла необходима синхронизация потоков.

К данной реализации, в отличие от первых двух, применим прием с разворачиванием итераций цикла, причем синхронизацию потоков можно проводить не на каждой итерации, а на каждой второй. Более редкая синхронизация в наших экспериментах уже не помогает избежать конфликта записи.

```
__global__ void kernel_NxN(float4 force[], float4 pos[], int type[], float4 coefs[])
{
  int i = threadIdx.x, j, type_i = type[i];
  float4 pos_i = pos[i], force_i = { 0, 0, 0, 0 };

  __shared__ float4 pos_j[N];
  __shared__ int type_j[N];
  __shared__ float4 coefs_ij[4];
  __shared__ float4 force_j[N];
  if (threadIdx.x < 4) coefs_ij[i] = coefs[i];
  pos_j[i] = pos[i]; type_j[i] = type[i];
  force_j[j] = make_float4(0, 0, 0, 0);
  __syncthreads();

  for (j = i; j < N + i; j++)
  {
    if (j < N)
    {
      float4 Fij = force_ij(pos_i, pos_j[j], coefs_ij[type_i + type_j[j]]);
      force_i += Fij; force_j[j] -= Fij;
    }
    __syncthreads();
  }

  force[i] = force_i + force_j[i];
}
```

Таким образом, при достаточно благоприятных условиях (система с небольшим числом частиц и расчет на одном мультипроцессоре, тогда расходы на коммуникации минимальны) лучшая из реализаций треугольного цикла быстрее на 39% (табл. 5). Двукратное ускорение от треугольного цикла (получаемое в программах на CPU) недостижимо из-за наличия условия внутри цикла и дополнительной синхронизации потоков.

Таблица 5

Время для различных реализаций треугольного цикла по сравнению с квадратным циклом (на одном мультипроцессоре), N = 324

Вариант реализации треугольного цикла	Время на шаг, сек	Ускорение, разы
1: запись всех Fij, затем суммирование	0.001213	0.82
2: запись половины из всех Fij	0.000829	1.21
3: суммирование сил в разделяемой памяти	0.000721	1.39

7. Заключение. Задачи N тел, в частности, самосогласованное МД-восстановление межчастичных потенциалов [4] и МД-моделирование наносистем (размеры — десятки нанометров, времена — десятки-сотни наносекунд) [11], требуют огромных вычислительных ресурсов.

В настоящее время из доступных на рынке вычислительных устройств наибольшей пиковой теоретической производительностью обладают графические процессоры (в частности [18], 622 GFLOPS у видеокарты NVIDIA GeForce GTX 280 и 2400 GFLOPS у AMD Radeon 4870x2), при этом их цена находится в диапазоне 10–20 тысяч рублей. Появившаяся в прошлом году программная платформа NVIDIA

CUDA позволила по некоторым оценкам [15] достичь 98% теоретической производительности на задаче N тел для системы из более 130 тысяч частиц (без учета расчетов на CPU и обмена данными между CPU и GPU), тогда как на CPU Athlon64 2 ГГц с использованием SIMD-оптимизаций и при расчетах с двойной точностью удалось получить только около 4 GFLOPS (50% от пиковой производительности) [23].

В настоящей работе мы обсуждаем неадекватность распространенного способа оценки производительности прикладных задач в GFLOPS (как в работах [15] и [23]), связанную с различной реализацией операций плавающей арифметики на разных аппаратных архитектурах. Мы предлагаем сравнивать производительность архитектур и реализаций в задаче N тел через количество парных взаимодействий, вычисляемых в единицу времени (или, что эквивалентно, через время вычисления одного парного взаимодействия) при одинаковой форме межчастичного потенциала.

Задача МД-моделирования, рассматриваемая в данной работе, имеет некоторые отличия от задачи N тел в работах других авторов, в частности, наличие разных типов частиц и короткодействующих потенциалов (для отталкивания и притяжения электронных оболочек); кроме того, число частиц не произвольно, а выбирается кратно их числу в элементарной ячейке. Предложены следующие приемы для увеличения скорости расчетов:

— опираемся на делители числа частиц для сокращения коммуникаций и эффективной загрузки каждого мультипроцессора, а не на степени двойки, являющейся основой аппаратной реализации GPU от NVIDIA;

— заменяем условный оператор "`if (i != j)`" функцией `max` для корректной обработки самодействия частиц, при этом, в отличие от распространенного приема добавления небольшого числа в знаменатель, точность расчетов не ухудшается, а скорость заметно выше;

— уменьшаем количество операций выбора по типам частиц коэффициентов их парного взаимодействия за счет приема разворачивания цикла.

Как видно из табл. 4, при МД-моделировании наносистем при нулевых граничных условиях достигнуто ускорение до 660 раз по сравнению с приведенным скалярным кодом на CPU. Сравнение времени обработки одной пары частиц на одинаковых GPU с работой [15] показало преимущество нашей CUDA-версии, причем в отличие от астрофизической задачи наше межчастичное взаимодействие дополнительно включает короткодействующий степенной потенциал.

Кроме того, в настоящей работе впервые обсуждаются параллельные реализации на GPU треугольного цикла расчета парных взаимодействий, учитывающего равенство действия противодействию, применяющегося во всех расчетах на CPU, поскольку такой подход позволяет сократить число операций в два раза. Нам удалось предложить параллельную реализацию треугольного цикла при нулевых граничных условиях, которая при определенных ограничениях быстрее наилучшей реализации с квадратным циклом на 39%.

Авторы благодарны профессору кафедры молекулярной физики Уральского государственного технического университета д.ф.-м.н. А. Я. Купряжкину за предоставленное оборудование и поддержку.

список литературы

1. http://en.wikipedia.org/wiki/Computer_simulation
2. http://en.wikipedia.org/wiki/Molecular_dynamics
3. *Gibbon P., Sutmann G.* Long-range interactions in many-particle simulation // Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms. Lecture Notes / Grotendorst J., Marx D., Muramatsu A. (Eds.). Neumann Institute for Computing Series. Vol. 10. Jülich, 2002. 467–506.
4. *Поташников С.И., Боярченков А.С., Некрасов К.А., Купряжкин А.Я.* Молекулярно-динамическое восстановление межчастичных потенциалов в диоксиде урана по тепловому расширению // Альтернативная энергетика и экология. 2007. **8**. 43–52 (http://isjaee.hydrogen.ru/pdf/AEE0807/AEE08-07_Potashnikov.pdf).
5. *Боярченков А.С., Поташников С.И.* Класс методов ветвления с неявной оценкой константы Липшица и бисекцией для решения задач ограниченной глобальной оптимизации (готовится в печать).
6. SIGGRAPH 2005 GPGPU course (http://www.gpgpu.org/s2005/).
7. *Buck I.* Molecular dynamics on graphics hardware // IEEE Visualization 2004 TUTORIAL (http://www.gpgpu.org/vis2004/I.buck.vis04.raymolc.pdf).
8. *Buck I., Rangasayee V., Darve E., Pande V., and Hanrahan P.* Accelerating molecular dynamics with GPUs // Proceedings of ACM Workshop on General Purpose Computing on Graphics Processors, 2004 (http://www.cs.unc.edu/Events/Conferences/GP2/proc.pdf).
9. *Поташников С.И., Некрасов К.А., Купряжкин А.Я., Боярченков А.С., Рисованный В.Д., Голованов В.Н.* Высокоскоростное моделирование диффузии ионов урана и кислорода в UO2 // Тр. Всероссийского Семинара

"Вопросы создания новых методик исследований и испытаний, сличительных экспериментов, аттестации и аккредитации". Димитровград, 2005.

10. *Hummer G.* The numerical accuracy of truncated Ewald sums for periodic systems with long-range Coulomb interactions. 1995. (http://arxiv.org/pdf/chem-ph/9502004v1).
11. *Поташников С.И., Боярченков А.С., Некрасов К.А., Купряжкин А.Я.* Моделирование массопереноса в диоксиде урана методом молекулярной динамики с использованием графических процессоров // Альтернативная энергетика и экология. 2007. **5**. 86–93 (http://isjaee.hydrogen.ru/pdf/AEE0507/ISJAEE05-07_Potashnikov.pdf).
12. *Elsen E., Houston M., Vishal V., Darve E., Hanrahan P., and Pande V.* N-Body Simulation on GPUs. 2006 (http://arxiv.org/pdf/0706.3060).
13. *Боярченков А.С., Поташников С.И., Некрасов К.А., Купряжкин А.Я.* Возможности графических процессоров для высокоскоростных параллельных вычислений и физического моделирования // Тр. Всероссийского Семинара "Физическое моделирование изменения свойств реакторных материалов в номинальных и аварийных условиях". Димитровград, 2008.
14. *Hamada T., Iitaka T.* The chamomile scheme: an optimized algorithm for N-body simulations on programmable graphics processing units. 2007 (http://arxiv.org/pdf/astro-ph/0703100v1).
15. *Belleman R.G., Bedorf J., Zwart S.P.* High performance direct gravitational N-body simulations on graphics processing units - II: An implementation in CUDA. 2007 (http://arxiv.org/pdf/0707.0438v2).
16. *Боярченков А.С., Поташников С.И.* Использование графических процессоров и распределенных вычислений для восстановления межчастичных потенциалов (готовится в печать).
17. http://en.wikipedia.org/wiki/GPGPU
18. Сравнительные характеристики процессоров компаний Intel, ATI, NVIDIA (http://en.wikipedia.org/wiki/List_of_Intel_microprocessors, http://en.wikipedia.org/wiki/Comparison_of_NVIDIA_Graphics_Processing_Units, http://en.wikipedia.org/wiki/Comparison_of_ATI_Graphics_Processing_Units).
19. CUDA Documentation (http://www.nvidia.com/object/cuda_develop.html).
20. Verlet integration (http://en.wikipedia.org/wiki/Verlet_integration).
21. NVIDIA GeForce GTX 200 GPU Datasheet (http://www.nvidia.com/docs/IO/55506/ GeForce_GTX_GPU_Datasheet.pdf).
22. DirectX Developer Center (http://msdn.microsoft.com/en-us/directx/default.aspx).
23. *Nitadori K., Makino J., and Hut P.* Performance tuning of N-body codes on modern microprocessors: I. Direct integration with a Hermite scheme on x86 64 Architecture // New Astronomy. 2006. **12**. 169–181.
24. *Diefendorff K.* Pentium III = Pentium II + SSE // Microprocessor Report. March 1999. **13** (http://studies.ac.upc.edu/ETSETB/SEGPAR/microprocessors/pentium3%20(mpr).pdf).
25. Increasing the accuracy of the results from the reciprocal and reciprocal square root instructions using the Newton–Raphson method. Intel Corporation, 1999 (http://cache-www.intel.com/cd/00/00/04/10/41007_nrmethod.pdf).

Поступила в редакцию
30.10.2008