УДК 004.021:004.94:519.683:519.684:544.223

ПАРАЛЛЕЛЬНАЯ МОЛЕКУЛЯРНАЯ ДИНАМИКА С СУММИРОВАНИЕМ ЭВАЛЬДА И ИНТЕГРИРОВАНИЕМ НА ГРАФИЧЕСКИХ ПРОЦЕССОРАХ

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

Рассмотрена задача молекулярно-динамического моделирования при периодических граничных условиях. Обсуждаются эффективные алгоритмы для метода суммирования Эвальда. Предложены параллельные реализации на основе технологии NVIDIA CUDA. Исследован вопрос интегрирования уравнений движения на графических процессорах с одинарной и двойной точностью плавающей арифметики. На видеокарте NVIDIA GeForce GTX 280 достигнуто удельное (на одну систему) ускорение до ≈ 890 раз по сравнению со скалярным расчетом на процессоре Intel Core2 Quad Q9550.

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

1. Введение. Метод динамики частиц (молекулярной динамики, или МД) — один из самых широко используемых в современной вычислительной науке [1]. Существуют две похожие задачи: гравитационная и электромагнитная динамика, где дальнодействующие (убывающие пропорционально квадрату расстояния) межчастичные силы считаются суперпозицией всех парных взаимодействий (закон Ньютона, закон Кулона), а движение системы частиц моделируется интегрированием уравнений Ньютона [2]. Задачи МД-моделирования условно можно разбить на 2 класса:

1) моделирование больших открытых систем из 10^4–10^8 частиц при нулевых граничных условиях (НГУ); методы расчета — прямое суммирование и иерархические мультипольные методы [3];

2) моделирование бесконечных систем при периодических граничных условиях (ПГУ) транслированием области из 10^2–10^4 частиц; методы расчета — суммирование Эвальда и "частица-сетка" (использующие переход к обратному пространству типа преобразования Фурье) [3].

В настоящее время из доступных на рынке вычислительных устройств графические процессоры (GPU) обладают наибольшей пиковой теоретической производительностью и являются оптимальной по соотношению цена–производительность параллельной архитектурой с общей памятью. Например, GPU AMD Radeon HD4870x2 с тактовой частотой 0.75 ГГц, доступный на рынке по цене около 15000 руб., имеет пиковую теоретическую производительность 2400 GFLOPS (миллиардов операций плавающей арифметики в секунду) [4], тогда как последний центральный процессор (CPU) 4-ядерный Intel Core i7 965 Extreme Edition с частотой 3.2 ГГц — всего 102 GFLOPS при цене около 35000 руб. Несколько лучшим соотношением обладает процессор STI Cell BE [5], использующийся в игровой приставке Sony Playstation3 и обеспечивающий 154 GFLOPS при частоте 3.2 ГГц и цене 15000 руб. за приставку.

При МД-моделировании больших систем основная часть времени уходит на расчет парных взаимодействий: число пар частиц равно $N(N-1)/2$, тогда как число операций для интегрирования пропорционально N. Поэтому при решении задач первого класса выгодно рассчитывать взаимодействия с максимальной скоростью на GPU, а интегрировать динамику с максимальной точностью на CPU. Однако при моделировании малых систем наибольшая часть времени может уходить на операции, не связанные с расчетом взаимодействий (обмен данными между CPU и GPU, интегрирование, коррекции, расчет микро- и макропараметров системы). Поэтому для сокращения задержек и ускорения расчетов важно исследовать возможности реализации на GPU полного цикла вычислений одного шага МД.

Такая реализация требует решения вопроса о точности плавающей арифметики. С точки зрения скорости долю расчетов с двойной (64-битной) точностью следует минимизировать, но при длительном моделировании с одинарной (32-битной) точностью всех операций происходит существенное накопление

[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. препод., e-mail: potashnikov@gmail.com

погрешностей (которое, однако, можно скомпенсировать коррекцией законов сохранения импульса, момента импульса и энергии системы). Последнее поколение GPU аппаратно поддерживает арифметику двойной точности (стандарт IEEE 754R [6]), но с пиковой производительностью на порядок меньше, чем одинарной [7]. Поэтому представляет интерес сравнение точности и скорости GPU-реализаций МД с разными форматами плавающей арифметики.

В предыдущей работе [8] мы обсудили детали реализации параллельной МД на GPU для задач первого класса (большие открытые системы) с использованием технологии NVIDIA CUDA [9]: распараллеливание по частицам, сокращение и кэширование обращений к памяти, сокращение ветвлений, разворачивание циклов. Там ускорение нарастало с увеличением числа частиц и достигало насыщения на системах порядка 50000 частиц. При этом на системах порядка 1000 частиц ускорение было невелико, что свидетельствует о неоптимальном использовании вычислительных ресурсов.

Следовательно, задачи второго класса (малые периодические системы) требуют отдельного исследования и, вероятно, других подходов к распараллеливанию. Актуальность этой темы косвенно подтверждается дорогостоящей разработкой специализированных процессоров, таких как чип WINE2 для проекта Molecular Dynamics Machine [10]. В данной статье мы собираемся: рассмотреть детали реализации метода Эвальда, в том числе эффективные варианты цикла в обратном пространстве, сокращающие число операций плавающей арифметики (разделы 4 и 5); обсудить версию для распределенных вычислений, где вместе с распараллеливанием по частицам используется распараллеливание по системам (раздел 5.3); сравнить реализации с интегрированием на CPU и GPU при разной точности арифметики (раздел 6); оценить доли каждого этапа расчетов (в общем времени) и дальнейшие оптимизации (раздел 7).

2. Технология NVIDIA CUDA (Compute Unified Device Architecture). CUDA — это технология параллельных вычислений, позволяющая разрабатывать масштабируемые (по аппаратным ресурсам) программы для GPU на стандартном языке Си (с несколькими расширениями). Она разработана компанией NVIDIA и в настоящий момент поддерживается любыми ее GPU, начиная с поколения G80 [11]. Ее ключевые абстракции [9] — иерархия сгруппированных потоков обработки (threads), разделяемая память (shared memory) и барьерная синхронизация.

Совместимые с CUDA графические процессоры компании NVIDIA содержат от 8 до 240 скалярных процессоров (СП), каждый из которых способен выполнять до трех операций плавающей арифметики (FLOP) одинарной точности за такт (одновременное выполнение операции mad вида `x*y+z` с операцией умножения). Соответственно, пиковая теоретическая производительность двухчиповой видеокарты NVIDIA GeForce GTX 295 составляет 2 GPU$\times$240 СП$\times$3 FLOP$\times$1.242 ГГц = 1789 GFLOPS.

Скалярные процессоры объединены по 8 в SIMD-мультипроцессоры (МП), которые обладают разделяемой (между потоками в одной связке) памятью объемом 16 КБ и общим декодером инструкций. Таким образом, связка потоков (thread block) 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 в разделе A.1 General Specifications [9]). Следовательно, масштабируемая программа на CUDA должна создавать множество связок потоков, достаточное для загрузки всех МП данного GPU.

Если в нескольких параллельных потоках обработки одновременно производится запись в одну ячейку памяти, то происходит так называемый "конфликт записи"; в этом случае сериализуются (т.е. 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.

3. Молекулярная динамика при ПГУ: метод суммирования Эвальда. Рассмотрим динамику системы из N заряженных частиц, в которой каждая из них электростатически (по закону Кулона) взаимодействует со всеми остальными, а результирующие силы и энергия — это суперпозиция $N(N-1)/2$ независимых парных взаимодействий.

Для расчета такой системы мы выбрали методы прямого суммирования при НГУ и суммирования Эвальда при ПГУ [12], так как по сравнению с приближенными иерархическими и сеточными методами (Treecode, Fast Multipole Method и Particle-Mesh [3]) они являются точными и простота их алгоритмов обеспечивает высокую эффективность параллельной реализации на GPU. На системах до ста тысяч частиц точные методы также оказываются быстрее приближенных, которые имеют меньшую вычислительную асимптотику $O(N \log N)$ и $O(N)$, но проигрывают в количестве операций на частицу. Кроме того, иерархические и сеточные методы для расчета взаимодействий между ближайшими соседями также используют прямое суммирование или суммирование Эвальда.

Следующим важным моментом является выбор короткодействующих потенциалов, аппроксимирующих взаимодействие частиц на расстояниях порядка суммы радиусов внешних электронных оболочек, так как при достаточной точности суммирования парных взаимодействий и интегрирования уравнений движения все структурные и динамические свойства моделируемой системы полностью определяются этими потенциалами. Наиболее распространенные формы короткодействия — степенные потенциалы Леннарда–Джонса и экспоненциальные Борна–Майера [13]. Степенное короткодействие обладает преимуществом в скорости расчетов за счет замены целочисленных отрицательных степеней расстояния между частицами операцией обратного корня (rsqrt) и умножениями. Под экспонентой потенциалов Борна–Майера имеется положительная степень расстояния, которая требует дополнительный регистр и ресурсоемкую операцию деления.

Параметризацию межчастичных потенциалов можно проводить на основе квантово-механических расчетов (из первых принципов) или по известным экспериментальным данным (эмпирически). В работе [14] нами предложен метод восстановления эмпирических потенциалов в ионных системах по теплофизическим экспериментальным данным на основе самосогласованного МД-моделирования (при котором параметризация потенциалов и их использование проходят в одинаковых условиях).

В этом методе расчет взаимодействий при периодических граничных условиях (на системах порядка 1000 частиц) мы проводим суммированием Эвальда в приближении "ближайших отражений" (максимальное межчастичное расстояние равно половине периода трансляции L) и конечного числа векторов обратной решетки $\boldsymbol{k}$:

$$\forall \boldsymbol{r}:\ |r_x|,\,|r_y|,\,|r_z| \leqslant L/2; \quad \forall \boldsymbol{k}:\ |\boldsymbol{k}| \leqslant K_{\max}(2\pi/L),$$

$$\boldsymbol{r}_{ij} = \boldsymbol{r}_i - \boldsymbol{r}_j; \quad r_{ij} = \sqrt{\boldsymbol{r}_{ij}\boldsymbol{r}_{ij}}; \quad w = \frac{2\pi}{L}, \quad U_{ij} = \frac{Q^2 K_e q_i q_j}{r_{ij}} \operatorname{erfc}(w r_{ij}) + A_{ij} r_{ij}^{-B_{ij}},$$

$$\nabla U_{ij} = -\frac{\boldsymbol{r}_{ij}}{r_{ij}} \left(\frac{Q^2 K_e q_i q_j}{r_{ij}^2} \left(\operatorname{erfc}(w r_{ij}) + \frac{2 w r_{ij}}{\sqrt{\pi}} \exp\bigl(-(w r_{ij})^2\bigr) \right) + A_{ij} B_{ij} r_{ij}^{-B_{ij}-1} \right),$$

$$f_{\boldsymbol{k}} = \frac{Q^2 K_e 4\pi \exp\bigl(-\boldsymbol{k}^2/4w^2\bigr)}{L^3 \boldsymbol{k}^2}; \quad E_{\boldsymbol{k}} = f_{\boldsymbol{k}} \left(\left(\sum_{i=1}^{N} q_i \cos(\boldsymbol{r}_i \boldsymbol{k}) \right)^2 + \left(\sum_{i=1}^{N} q_i \sin(\boldsymbol{r}_i \boldsymbol{k}) \right)^2 \right),$$

$$\boldsymbol{F}_i = q_i \sum_{\boldsymbol{k} \neq 0} \boldsymbol{k} f_{\boldsymbol{k}} \left(\sin(\boldsymbol{r}_i \boldsymbol{k}) \sum_{j=1}^{N} q_j \cos(\boldsymbol{r}_j \boldsymbol{k}) - \cos(\boldsymbol{r}_i \boldsymbol{k}) \sum_{j=1}^{N} q_j \sin(\boldsymbol{r}_j \boldsymbol{k}) \right) - \sum_{j=1}^{N} \nabla U_{ij},$$

$$3PV = \sum_{i=1}^{N} m_i \boldsymbol{v}_i^2 + \sum_{\boldsymbol{k} \neq 0} \left(1 - \frac{\boldsymbol{k}^2}{2w^2} \right) \frac{E_{\boldsymbol{k}}}{2} - \sum_{i=1}^{N} \sum_{j \neq i}^{N} \frac{\nabla U_{ij}}{2} \boldsymbol{r}_{ij},$$

$$E = \sum_{i=1}^{N} \frac{m_i \boldsymbol{v}_i^2}{2} + \sum_{\boldsymbol{k} \neq 0} \frac{E_{\boldsymbol{k}}}{2} + \sum_{i=1}^{N} \sum_{j \neq i}^{N} \frac{U_{ij}}{2} - \frac{2w}{\sqrt{\pi}} Q^2 K_e \sum_{i=1}^{N} q_i.$$

Здесь $\boldsymbol{r}_i$, $\boldsymbol{v}_i$, q_i, m_i — позиция, скорость, заряд и масса i-й частицы; L — линейный размер, длина ребра транслируемого куба с объемом $V = L^3$; $K_{\max}$ — радиус обрезания в обратном пространстве (в суммах по k-векторам); U_{ij} — искомый парный потенциал с параметрами степенного короткодействия A_{ij}, B_{ij} и безразмерным множителем зарядов Q; $K_e = 1.4399644$ эВ·нм — электростатическая константа Кулона, а w — параметр Эвальда (чем больше — тем быстрее затухает прямая сумма и медленнее обратная). При этом для процедуры восстановления на каждом шаге динамики помимо межчастичных сил F необходимо вычислять внутреннее давление в транслируемой области P (на основе теоремы вириала [15]), а также полную энергию системы E, если требуется расчет энтальпии или теплоемкости.

Для интегрирования уравнений движения Ньютона [16] мы применяем полуневяный (semi-implicit) метод Эйлера [17] с коррекцией суммарного импульса, термостатом Берендсена и аналогичным баростатом [18]:

$$x(t) = \left(1 + \left(\frac{T_{\text{stat}}}{T_{\text{system}}(t)} - 1 \right) \frac{1}{\tau_T} \right)^{1/2}, \quad n(t) = \left(1 + \bigl(P_{\text{system}}(t) - P_{\text{stat}} \bigr) \frac{1}{\tau_P} \right)^{1/3},$$

$$\boldsymbol{v}_i(t + \Delta t) = \left(\boldsymbol{v}_i(t) + \frac{\boldsymbol{F}_i(t)}{m_i} \Delta t - \left(\sum_{j=1}^{N} m_j \boldsymbol{v}_j(t) \Big/ \sum_{j=1}^{N} m_j \right) \right) x(t),$$

$$\boldsymbol{r}_i(t + \Delta t) = \bigl(\boldsymbol{r}_i(t) + \boldsymbol{v}_i(t + \Delta t) \Delta t \bigr) n(t), \quad L(t + \Delta t) = L(t) n(t).$$

Здесь T_{system} и P_{system} — температура и внутреннее давление системы в данный момент, T_{stat}, τ_T, x — температура, время релаксации и коэффициент термостата, P_{stat}, τ_P, n — давление, время релаксации и коэффициент баростата.

4. Реализация на CPU. Рассмотрим для начала простейший алгоритм суммирования Эвальда (на языке программирования C++), который, однако, учитывает зеркальную симметрию k-векторов (границами тройного цикла и удвоением factor) и сокращает объем вычислений в обратном пространстве в 2 раза:

```
void Ewald(double3 pos[], int type[], double charge[], double **coefs[], double L,
double3 force[], double *Stress)
{
  int i, j, kx, ky, kz, kx0, ky0;
  double Pi2_L = M_PI * 2 / L, W = Pi2_L, _4WW = 1 / (4 * W * W);
  double KePi8_LLL = Ke * M_PI * 8 / (L * L * L);
  for (*Stress = 0, kz = ky0 = 0, kx0 = 1; kz <= Kmax; kz++, ky0 = -Kmax)
    for (ky = ky0; ky <= Kmax; ky++, kx0 = -Kmax)
      for (kx = kx0; kx <= Kmax; kx++)
        if (kx * kx + ky * ky + kz * kz <= Kmax * Kmax)
        {
          double QCos = 0, QSin = 0, KK, KR, factor;
          double3 k = Pi2_L * double3(kx, ky, kz); KK = k * k;
          for (i = 0; i < N; i++)
          {
            KR = k * pos[i];
            QCos += charge[type[i]] * cos(KR); QSin += charge[type[i]] * sin(KR);
          }
          factor = KePi8_LLL * exp(-KK * _4WW) / KK;
          *Stress += (0.5 - KK * _4WW) * factor * (QCos * QCos + QSin * QSin);
          QCos *= factor; QSin *= factor;
          for (i = 0; i < N; i++)
          {
            KR = k * pos[i];
            force[i] += k * charge[type[i]] * (sin(KR) * QCos - cos(KR) * QSin);
          }
        }
  for (i = 0; i < N; i++)
    for (j = i + 1; j < N; j++)
    {
      double[] c = coefs[type[i]][type[j]];
      double3 R = pos[i] - pos[j];
      if (R.x > 0.5 * L) R.x -= L; else if (R.x < -0.5 * L) R.x += L;
      if (R.y > 0.5 * L) R.y -= L; else if (R.y < -0.5 * L) R.y += L;
      if (R.z > 0.5 * L) R.z -= L; else if (R.z < -0.5 * L) R.z += L;
      double r = sqrt(R * R), _r = 1 / r;
      double f = c[0] * dErfc(W * r) * _r * _r * _r + pow(_r * c[1], c[2]);
      *Stress += f * r * r; force[i] += f * R; force[j] += -f * R;
    }
}
double dErfc(double WR) { return Erfc(WR) + WR * exp(-WR * WR) * 2 / sqrt(M_PI); }
```

Здесь массивы pos и type содержат координаты и типы частиц, массивы charge и coefs — заряды (с учетом множителя Q) и коэффициенты парного потенциала в зависимости от типов частиц. В массив force сохраняются результирующие силы, действующие на каждую частицу, в переменную Stress — потенциальная составляющая внутреннего давления. Для функции Erfc мы использовали аппроксимацию из монографии [19], так как она обеспечила достаточную точность расчетов за минимальное время вычисления.

По сравнению с реализацией прямого суммирования [8] в методе Эвальда добавляется тройной цикл по k-векторам обратной решетки, а также ресурсоемкое вычисление функции dErfc и условия проверки

выхода за границу транслируемой области в двойном цикле по частицам. При этом учет третьего закона Ньютона (равенство действия противодействию) позволяет однократно вычислять парное взаимодействие, сокращая количество операций почти в 2 раза (такой вариант двойного цикла по частицам мы называем "треугольным", см. подробности в [8]). Вычислительная сложность расчетов в обычном пространстве остается — $O(N^2)$, а в обратном — равна $O(KN)$, где K — общее количество k-векторов.

В цикле по k-векторам тригонометрические операции $\cos(\mathrm{KR})$ и $\sin(\mathrm{KR})$ (части комплексной экспоненты) вычисляются в первом и во втором внутренних циклах по частицам, т.е. гораздо чаще, чем остальные. Эти операции очень ресурсоемкие: к примеру, на процессорах семейства Intel Core2 инструкция FSINCOS (вычисляющая сразу синус и косинус аргумента) выполняется за 140 тактов, тогда как умножение или сложение — всего за 2 (см. таблицу в [20]), т.е. две тригонометрических операции стоят 70 арифметических. Поэтому мы предлагаем кэшировать массивами qcos и qsin длины N величины `q*cos(KR)` и `q*sin(KR)`, вычисляемые для каждой частицы при фиксированном k-векторе в первом внутреннем цикле, и использовать их во втором. При этом число тригонометрических операций сокращается в 2 раза (в других участках кода они не встречаются), а арифметических — на треть (в первом цикле по частицам имеется 9 сложений и умножений плавающей арифметики, а во втором их число сокращается с 15 до 9).

Для нашей задачи восстановления межчастичных потенциалов достаточно небольшого числа частиц в транслируемой области (от 300 до 1500, см. обсуждение в работе [14]), если радиус обрезания в обратном пространстве Kmax больше или равен 6 (при этом с учетом зеркальной симметрии из $(2\times6+1)^3-1=2196$ k-векторов остается 462). В таких условиях относительная точность вычисления сил составляет 10^{-6} (см. также [21]), а общая скорость расчетов благодаря нашей модификации увеличивается на 24–41% (см. табл. 1) при почти двукратном ускорении цикла по k-векторам (т.е. время доступа к кэширующим массивам в этом случае пренебрежимо мало).

Известна другая реализация цикла в обратном пространстве (см., например, код в пакете программ GROMACS [22]), которая учитывает сложную симметрию k-векторов по трем координатным осям. В ней ресурсоемкие операции $\cos(\mathrm{KR})$ и $\sin(\mathrm{KR})$ заменяются покоординатными компонентами — $\cos(\mathrm{kx}*\mathrm{x})$, $\cos(\mathrm{ky}*\mathrm{y})$ и т.д., которые вычисляются один раз в отдельном цикле и сохраняются в трехмерном массиве размера 6N(Kmax+1). Затем в основном цикле на его основе вычисляются комбинации компонент и кэшируются четырьмя массивами длины N.

В итоге 2KN тригонометрических операций нашей версии заменяются на 6N тригонометрических операций (т.е. в K/3 раз меньше) и (18(Kmax-1)+6(2Kmax+1)(Kmax+1)+6K)N арифметических, т.е. $\dfrac{3408\mathrm{N}}{924\mathrm{N}} \approx 3.7$ арифметические операции на одну исходную тригонометрическую. При этом объем кэширующих массивов составляет `sizeof(float)*(6*(Kmax+1)+4)*N`=184N байт, что в 23 раза больше объема `sizeof(float)*2*N`=8N байт наших массивов qcos и qsin.

Без учета доступа к памяти ускорение цикла по k-векторам за счет 3-осевой симметрии по сравнению с нашей версией могло бы составить $\dfrac{(70+9+9)\mathrm{KN}}{(70\times3/\mathrm{K}+3.7\times2+4+9)\mathrm{KN}} = \dfrac{88}{20.9} \approx 4.2$ раза. Однако на практике оно составляет ≈ 2.5 раза (т.е. время доступа к кэширующим массивам в этом случае велико) и общее время расчетов сокращается всего лишь на 15–30%. Со временем локальные вычисления становятся все более выгодны в сравнении с запросами к общей памяти (см. тенденции [23]), поэтому такая версия неперспективна и ее реализацию мы не приводим (см. код в пакете GROMACS [22]).

5. Реализация на GPU. Отличия обсуждаемой в данной статье GPU-реализации МД от работ других авторов [24, 25] заключаются в следующем:

— малые системы при периодических граничных условиях, то есть заметная доля общего времени приходится на операции, не относящиеся к расчету взаимодействий;

— метод Эвальда вместо прямого суммирования парных взаимодействий;

— расчет не только межчастичных сил, но также внутреннего давления (или энергии) системы;

— 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);

— общее число частиц не является целой степенью двойки (что было бы выгодно на GPU от NVIDIA), а кратно 12 (числу частиц в элементарной ячейке моделируемых нами кристаллов диоксида урана).

5.1. Распараллеливание по потокам на одном мультипроцессоре. Простейший способ распараллеливания — замена внешних циклов метода Эвальда одновременной обработкой их итераций в разных потоках. Для одного МП возможна следующая реализация такого подхода на CUDA в случае N=324 и K=462 (при котором используется ≈ 16300 байт разделяемой памяти из 16384 доступных):

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

  __shared__ float4 pos_j[N], coefs_ij[3];
  __shared__ int type_j[N];
  if (i < 3) coefs_ij[i] = coefs[i];
  pos_j[i] = pos[i]; type_j[i] = type[i];
  __syncthreads();

  for (int j = 0; j < N; j++)
    force_i += Fij(pos_i, pos_j[j], coefs_ij[type_i + type_j[j]], L, W);
  force[i] = force_i;
}
__global__ void kernel_KxNK(float4 force[], float4 stress[], float4 pos[], float4 k[],
float Pi2_L, float KePi8_LLL, float _4WW)
{
  int i = threadIdx.x, j;
  float QCos = 0, QSin = 0, KR, factor;
  float4 k_i = k[i] * Pi2_L; k_i.w *= Pi2_L;

  __shared__ float4 pos_j[N], k_j[K]; // float * (4*N + 4*K + 2*K) = 16272 bytes
  __shared__ float2 QExp[K];
  k_j[i] = k_i; if (i < N) pos_j[i] = pos[i];
  __syncthreads();

  #pragma unroll 108
  for (j = 0; j < N; j++)
  {
    KR = dot3(k_i, pos_j[j]);
    QCos += pos_j[j].w * cosf(KR);
    QSin += pos_j[j].w * sinf(KR);
  }
  factor = KePi8_LLL * expf(-k_i.w * _4WW) / k_i.w;
  stress[i] = factor * (1 - 2 * k_i.w * _4WW) * (QCos * QCos + QSin * QSin);
  QExp[i] = make_float2(QCos * factor, QSin * factor);
  __syncthreads();

  if (i < N)
  {
    float4 pos_i = pos_j[i], force_i = { 0, 0, 0, 0 };
    #pragma unroll 77
    for (j = 0; j < K; j++)
    {
      KR = dot3(k_j[j], pos_i);
      mad3(force_i,
        k_j[j], pos_i.w * (sinf(KR) * QExp[j].x - cosf(KR) * QExp[j].y));
    }
    force[i] += force_i;
  }
}
__device__ float PBC(float x, float L)
{
  if (x > L) return x - 2 * L; else if (x < -L) return x + 2 * L; else return x;
}
```

```
__device__ float4 Fij(float4 pos_i, float4 pos_j, float4 c, float L, float W)
{
  float4 R = make_float4(PBC(pos_i.x - pos_j.x, L), PBC(pos_i.y - pos_j.y, L),
                         PBC(pos_i.z - pos_j.z, L), 0);
  R.w = dot3(R, R);
  float r = rsqrtf(max(R.w, 1e-4f));
  return R * (c.x * dErfc(W / r) * r * r * r + powf(c.y * r, c.z));
}
__device__ float dErfc(float WR)
{
  return Erfc(WR) + WR * expf(-WR * WR) * 1.128379167f; // 2/sqrt(PI) = 1.128379...
}
```

Названия CUDA-ядер содержат информацию о циклах: в ядре kernel_NxN внешний и внутренний циклы пробегают по частицам, в ядре kernel_KxNK первый внутренний цикл пробегает по частицам, а внешний и второй внутренний — по k-векторам.

Функция dot3 вычисляет скалярное произведение из первых трех компонент передаваемых ей векторов, функция mad3 — операцию вида `x += y * z`. Массивы force, pos, k имеют тип float4: во-первых, для более быстрого доступа (по сравнению с типом float3); во-вторых, четвертая компонента используется для хранения потенциальной составляющей внутреннего давления, заряда частицы и квадрата длины k-вектора соответственно. Для простоты в ядре kernel_KxNK вклады в давление по k-векторам записываются в отдельный массив stress, так как при $K > N$ для параллельной записи в массив force пришлось бы увеличивать его размер либо вручную сериализовать запись нескольких чисел в одну ячейку памяти (см. "конфликт записи" в разделе 2).

Реализация двойного цикла по частицам в ядре kernel_NxN аналогична нашей реализации прямого суммирования [8]. Расчет взаимодействия пары частиц вынесен в функцию Fij, обработка периодических граничных условий — в функцию PBC (Periodic Boundary Conditions), причем в нее для уменьшения числа операций умножения передается половина линейного размера транслируемой области (L/2). В CUDA имеется встроенная реализация функции Erfc, однако мы, как и в CPU-версии, в теле функции dErfc используем более быструю аппроксимацию из монографии [19].

Тройной (внешний) цикл по k-векторам заменяется параллельной обработкой в разных потоках ядра kernel_KxNK. Первый внутренний цикл по частицам полностью соответствует CPU-реализации из предыдущего раздела, только заряды частиц берутся из четвертой компоненты векторов pos_j. Во втором внутреннем цикле порядок вложенности меняется: в каждом из N потоков выполняется цикл по k-векторам (в случае $N < K$ остальные $K - N$ потоков простаивают) с накоплением сил по частицам в регистре force_i. При такой замене накопленные первым циклом K значений QCos и QSin необходимо сохранить в разделяемой памяти (в массиве QExp).

Версия с сохранением порядка вложенности во втором внутреннем цикле тоже была нами реализована. Но она потребовала разрешения конфликта параллельной записи из K потоков в N элементов массива force (кэшируемого разделяемой памятью). Для этого потоки разбивались на две группы: внутри каждой группы запись производилась параллельно, а синхронизация потоков позволяла не допустить одновременной записи потоками разных групп. Из-за синхронизации и условий (для выбора группы) внутри цикла, а также неэффективного доступа к разделяемой памяти (см. пункт 5.1.2.5 Shared memory [9]) такая версия выполняется в ≈ 3 раза медленнее, поэтому ее код мы не приводим.

Директива unroll в ядре kernel_KxNK указывает компилятору CUDA количество итераций цикла, которые необходимо развернуть для ускорения обработки. Разворачивать циклы в ядре kernel_NxN выгоднее вручную (см. пример кода в приложении), так как операцию выбора коэффициентов взаимодействия эффективнее выполнять один раз для нескольких итераций (при соответствующем порядке частиц в массивах). Отметим также, что из-за большего числа используемых регистров в ядре kernel_NxN выгодно меньшее число развернутых итераций цикла по сравнению с аналогичным ядром в реализации прямого суммирования [8].

Теперь обсудим кэширование результата тригонометрических операций cos(KR) и sin(KR), которое ускоряет скалярную реализацию на CPU (рассмотрим для примера тот же случай N=324, K=462). Наша скалярная CPU-версия модифицированного обратного цикла для хранения промежуточных результатов (массивы qcos и qsin) использует всего `N*sizeof(float)*2` $= 324 \times 8 = 2'592$ байта памяти. Однако для каждого k-вектора эти массивы перезаписываются, что невозможно при параллельной обработке на CUDA, где каждый k-вектор обрабатывается в отдельном потоке и для хранения всех таких величин

потребовалось бы в сумме `K*N*sizeof(float)*2` = 462×324×8 = 1'197'504 байт памяти. Это намного больше имеющихся 16 КБ разделяемой памяти или 64 КБ константной памяти. К тому же в последнюю из них нельзя производить запись из кода, выполняющегося на GPU, которая требуется внутри модифицированного цикла (см. пункт 4.2.2.4 Restrictions в [9]). Достаточный объем имеет только общая память GPU, которая не кэшируется и работает намного медленнее разделяемой, поэтому повторные вычисления выгоднее записи и чтения массивов qcos и qsin. Версия с кэшированием этих массивов может стать оптимальной на будущих параллельных архитектурах при достаточном объеме быстрой памяти. К примеру, на процессоре Cell, появившемся в продаже в конце 2006 года, каждый вычислитель SPE (Synergistic Processing Elements, аналог мультипроцессоров GPU) имеет уже 256 КБ кэша.

Параллельная версия с 3-осевой симметрией k-векторов требует существенно (в 23 раза) большего объема памяти для кэширования, кроме того, содержит дополнительные условия в циклах, поэтому ее реализация на GPU неперспективна.

5.2. Распараллеливание по связкам потоков на несколько мультипроцессоров. В версии из предыдущего раздела внешние циклы заменяются параллельной обработкой в одной связке потоков, поэтому используется только часть имеющихся ресурсов GPU — для масштабируемости требуется распределение расчетов между множеством МП (см. раздел 2).

Для этого можно разделить итерации внешних и внутренних циклов на блоки. Тогда каждой связкой потоков будут выполняться: один блок итераций внешнего цикла (параллельно по потокам) и один блок итераций внутреннего (последовательно в каждом потоке). Общее число связок будет равно произведению числа блоков, на которые разбиты циклы. В приложении мы приводим код такой реализации метода Эвальда на CUDA (для простоты опять рассматриваем случай N = 324 и K = 462, когда массивы координат и k-векторов умещаются в разделяемую память целиком).

Для реализации этого подхода в коде ядра kernel_NxN мы вводим следующие величины: BI — число блоков, на которые разбивается внешний цикл; BJ — число блоков, на которые разбивается внутренний цикл; NJ — число итераций внутреннего цикла в блоке: NJ = N/BJ, которое ограничено объемом разделяемой памяти: $\mathrm{NJ} < \dfrac{16384}{20} \approx 819$ (так как `sizeof(float4) + sizeof(int)` = 20 байт памяти на координаты и тип каждой частицы); j0 — абсолютный индекс первой из NJ частиц, записываемых в разделяемую память, j — относительный индекс частиц в разделяемой памяти.

При распределении внешнего цикла число потоков не меняется, поэтому конфликты записи не возникают. При распределении внутреннего цикла суммарное число потоков увеличивается в BJ раз, при этом доступ к каждому элементу массива force производится из нескольких связок. Так как синхронизация потоков между связками невозможна, для устранения этого конфликта записи мы увеличиваем объем массива force в BJ раз, чтобы каждая связка записывала результат в свою область памяти. По окончании расчетов выполняем редукцию такого массива ядром kernel_sum.

Распараллеливание ядра kernel_KxNK получается менее эффективным, так как:

— при распределении внешнего цикла по k-векторам в каждой связке потоков будут рассчитываться силы для каждой частицы (в каждом потоке для нескольких частиц) и для устранения конфликта записи в массив force необходимо будет кратно увеличить его размер;

— при распределении внутреннего цикла по k-векторам вычисления внутреннего цикла по частицам придется повторять в каждой дополнительной связке потоков;

— при распределении внутреннего цикла по частицам нельзя эффективно вычислять вклады k-векторов во внутреннее давление, так как они нелинейно зависят от суммарных величин QSin и QCos (квадрат суммы не выражается через сумму квадратов).

Чтобы эффективно задействовать множество МП, мы разбили ядро kernel_KxNK на два — kernel_KxN и kernel_NxK. При этом первое ядро распараллелено по k-векторам, а второе — по частицам, т.е. сохраняется измененный порядок вложенности циклов. Результаты промежуточных вычислений (вклады QSin и QCos от каждого k-вектора) мы передаем между ядрами через массив QExp, расположенный в общей памяти GPU. На одном МП такая версия работает на 2% медленнее из-за доступа к общей памяти и дополнительных издержек на вызов второго ядра, но распараллеливается более эффективно: распределение между МП итераций внешнего цикла не требует кратного увеличения массива force, а при распределении итераций внутреннего цикла вычисления не будут дублироваться. Для простоты в коде этих ядер (см. приложение) мы опустили прием с распределением внутреннего цикла, так как он дает лишь ≈5% общего ускорения (в связи с тем, что доля обоих ядер в общем времени расчетов менее 20% и уменьшается с ростом числа частиц, см. столбцы 1×324 и 1×768 в табл. 2).

В обоих ядрах заполнение разделяемой памяти реализовано в виде цикла, количество итераций которого определяется отношением числа заполняемых элементов массива к числу потоков:

— `memThreadRatio = ceiling(N * KxN_BLOCKS / K)` для ядра kernel_KxN;
— `memThreadRatio = ceiling(K * NxK_BLOCKS / N)` для ядра kernel_NxK.

Здесь константы KxN_BLOCKS и NxK_BLOCKS обозначают число МП, задействованных для параллельной обработки соответствующих ядер. Вызов функции min в условии цикла заполнения разделяемой памяти нужен для проверки выхода за границу массивов. В ядре kernel_NxN такой цикл необходим в случае BI > BJ, тогда `memThreadRatio = ceiling(BI / BJ)`.

5.3. Распараллеливание по системам и распределенные вычисления. При МД-моделировании системы из большого числа частиц рассмотренный выше прием (с распределением по связкам потоков итераций внешних и внутренних циклов) позволяет эффективно нагрузить все мультипроцессоры GPU. Если применять такой подход для малых систем, то возникают следующие проблемы (на примере кода ядра kernel_NxN в приложении): при большом значении BI недостаточное число потоков в связке не позволяет скрыть латентность памяти; при большом значении BJ узким местом становится пропускная способность памяти из-за кратного увеличения массива force и операции его редукции.

Следовательно, если количество связок потоков, равное BI×BJ, меньше числа МП, то некоторые из них будут простаивать, однако при больших значениях параметров BI и BJ время доступа к памяти начинает доминировать и ускорение от распараллеливания сменяется замедлением. Тенденции развития GPU показывают [4], что количество вычислителей на чипе увеличивается со временем значительно быстрее, чем их тактовая частота (удвоение за год против удвоения за 5 лет). Поэтому со временем (при обновлении эксплуатируемых GPU) будет простаивать все большее число МП, а ускорение будет определяться только ростом тактовой частоты (чипа и памяти). Еще один недостаток такого подхода заключается в необходимости поиска оптимального значения констант BI и BJ при любых изменениях условий: числа частиц в системе, типа потенциалов взаимодействия, характеристик эксплуатируемых GPU.

Поэтому для более эффективного использования ресурсов современных и будущих GPU в задачах второго класса (МД при периодических граничных условиях) мы предлагаем параллельное моделирование множества малых систем на всех имеющихся мультипроцессорах GPU. При реализации этого подхода на CUDA каждая связка потоков обрабатывает свою систему, а каждый поток — свою частицу. Все массивы по частицам (координаты, заряды и др.) и массив коэффициентов парных взаимодействий увеличиваются кратно числу S одновременно моделируемых систем. Параметры метода Эвальда, зависящие от линейного размера L каждой системы, также объединяются в массив длины S.

Такой параллелизм по системам применим при расчете макроскопических зависимостей (например, температурной зависимости, приведенной на рис. 2 в следующем разделе) или в задачах с множеством однотипных вычислительных экспериментов с разными начальными условиями. Например, в разработанном нами методе восстановления эмпирических межчастичных потенциалов с использованием глобальной оптимизации [26] для исследования пространства параметров задачи требуется большая плотность точек (значений целевой функции), при этом:

— метод глобальной оптимизации на каждой итерации генерирует множество вызовов целевой функции, то есть наборов коэффициентов межчастичных потенциалов, которые можно оценивать независимо (параллельно);

— каждый вызов целевой функции включает в себя несколько независимых (параллельных) МД-экспериментов, по которым оценивается среднеквадратичное отклонение вычисляемых макропараметров от известных экспериментальных данных.

Код распараллеленной по системам версии практически эквивалентен реализации для одного МП из раздела 5.1 (так как при достаточном числе систем распараллеливание по ним делает излишним прием с разбиением циклов на блоки). Отличия заключаются в следующем: количество связок потоков при вызовах ядер равно числу систем S, при доступе к массивам учитывается индекс связки. Распараллеленное по системам ядро kernel_KxNK (наиболее эффективное на одном МП) оказалось на 8% медленнее (вероятно, из-за условия перед циклом по k-векторам) версии с двумя ядрами kernel_KxN и kernel_NxK, поэтому в табл. 1 и 2 измерения скорости проведены для двуядерной реализации цикла в обратном пространстве. Кроме того, при расчетах каждой системы на одном МП во всех ядрах метода Эвальда отсутствуют конфликты записи и, следовательно, не требуются увеличение и редукция массива force.

При параллельном моделировании одной системы с фиксированным числом частиц ускорение от использования дополнительных вычислителей ограничено сверху законом Амдала [27]. Если задача допускает распараллеливание по системам, то соответствующая реализация будет масштабируемой: по закону Густафсона [28] ускорение не ограничено и будет расти вместе с числом одновременно моделируемых систем. Количество вычислителей на чипах удваивается за год, поэтому только масштабируемые задачи способны полностью задействовать ресурсы будущих GPU.

Хорошим примером таких задач являются проекты распределенных вычислений в Интернет: Folding@Home [29], SETI@Home и др. Особенностью подобных проектов является наличие огромного числа независимых подзадач, которые можно решать на разных клиентах. Если для загрузки всех вычислителей клиента не хватает одной подзадачи, то сервер всегда может предоставить дополнительные подзадачи для достижения максимальной эффективности.

Наша задача восстановления эмпирических межчастичных потенциалов становится неограниченно масштабируемой с использованием исчерпывающей глобальной оптимизации. Это свойство позволило разработать пакет программ для распределенных вычислений, объединяющий доступные нам 8 GPU с суммарной пиковой производительностью более 4 TFLOPS.

6. Реализация интегрирования и коррекций на GPU. В процессе оптимизации нашей реализации МД для малых периодических систем и эволюции графических процессоров наступил момент, когда оставшиеся не распараллеленными вычисления (интегрирование уравнений движения, коррекция импульса, термостат и баростат) стали занимать до половины общего времени расчетов (см. табл. 2), что нарушало масштабируемость программы. Параллельную реализацию этих вычислений на CPU (с использованием многоядерности и наборов команд SIMD) мы посчитали неперспективной в связи с опережающим развитием GPU (см. тенденции [8]) и необходимостью синхронизации характеристик каждой частицы в памяти CPU и GPU на каждом шаге МД. Отметим, что распараллеливание этих вычислений актуально только для малых систем, так как доля в общем времени, приходящаяся на расчет парных взаимодействий, увеличивается вместе с ростом числа частиц.

В настоящее время считается, что для интегрирования уравнений движения недостаточно одинарной (32-битной) точности плавающей арифметики (к примеру, в специализированных вычислителях GRAPE [30] расчет парных взаимодействий выполняется с одинарной точностью, а интегрирование — с двойной). В нашей задаче восстановления межчастичных потенциалов, в отличие от астрофизики, важны не траектории отдельных тел, а макропараметры, усредняемые по всей системе. При этом накопление погрешностей в ходе моделирования компенсируется обнулением смещения и вращения системы и применением термостата и баростата на каждом шаге МД, а длительность экспериментов невелика (порядка 1000–10000 шагов). Поэтому мы решили исследовать вопрос о достаточной точности арифметики для данной задачи путем экспериментальной проверки. Поскольку для решения других задач (например, астрофизических, в которых различные части системы интегрируются с разным шагом времени [30]) может потребоваться двойная точность, мы рассмотрели также версию интегрирования на GPU с 64-битной арифметикой.

На каждом шаге нашего МД-моделирования выполняются следующие действия.

1. Суммирование методом Эвальда парных сил f(t) и потенциальной компоненты внутреннего давления Stress(t) по частицам с координатами x(t).

2. Интегрирование скоростей частиц v(t+1).

3. Расчет температуры T(t+1), импульса MV(t+1) и внутреннего давления P(t+1) системы (суммированием вкладов отдельных частиц).

4. Обнуление импульса системы и применение термостата Берендсена v'(t+1).

5. Интегрирование координат частиц x(t+1).

6. Расчет коэффициента баростата Берендсена n(t+1), масштабирование линейного размера L(t+1).

7. Применение баростата Берендсена и периодических граничных условий x'(t+1).

Для краткости мы будем называть интегрированием все шаги кроме первого и введем следующие обозначения для разных реализаций: iCPU64 — версия с интегрированием на CPU, 64-битная арифметика; iGPU64 — версия с интегрированием на GPU, 64-битная арифметика; iGPU32 — версия с интегрированием на GPU, 32-битная арифметика. Код интегрирования на CPU приведен в приложении. Все шаги, кроме третьего и шестого, применяются к каждой частице независимо от остальных, поэтому их реализация на CUDA тривиальна (каждую частицу обрабатываем в отдельном потоке, а каждую систему — в отдельной связке потоков). Третий шаг — это редукция вкладов всех частиц, а шестой шаг — скалярный (рассчитываются скаляры n(t+1) и L(t+1), зависящие от скаляров P(t), T(t) и L(t)). Оба этих шага требуют синхронизации потоков, которую в нашем случае выгоднее проводить на каждом мультипроцессоре независимо (при помощи встроенной функции `__syncthreads`). Код интегрирования на CUDA при относительной простоте получился довольно громоздким, поэтому опущен нами для краткости.

При МД-моделировании с интегрированием на GPU передача данных между CPU и GPU ограничивается копированием на CPU макропараметров каждой системы: температуры, внутреннего давления и линейного размера. Копирование полного состояния (координат, сил и скоростей всех частиц) производится только в случае замены одной из моделируемых систем (например, по окончании связанного с ней

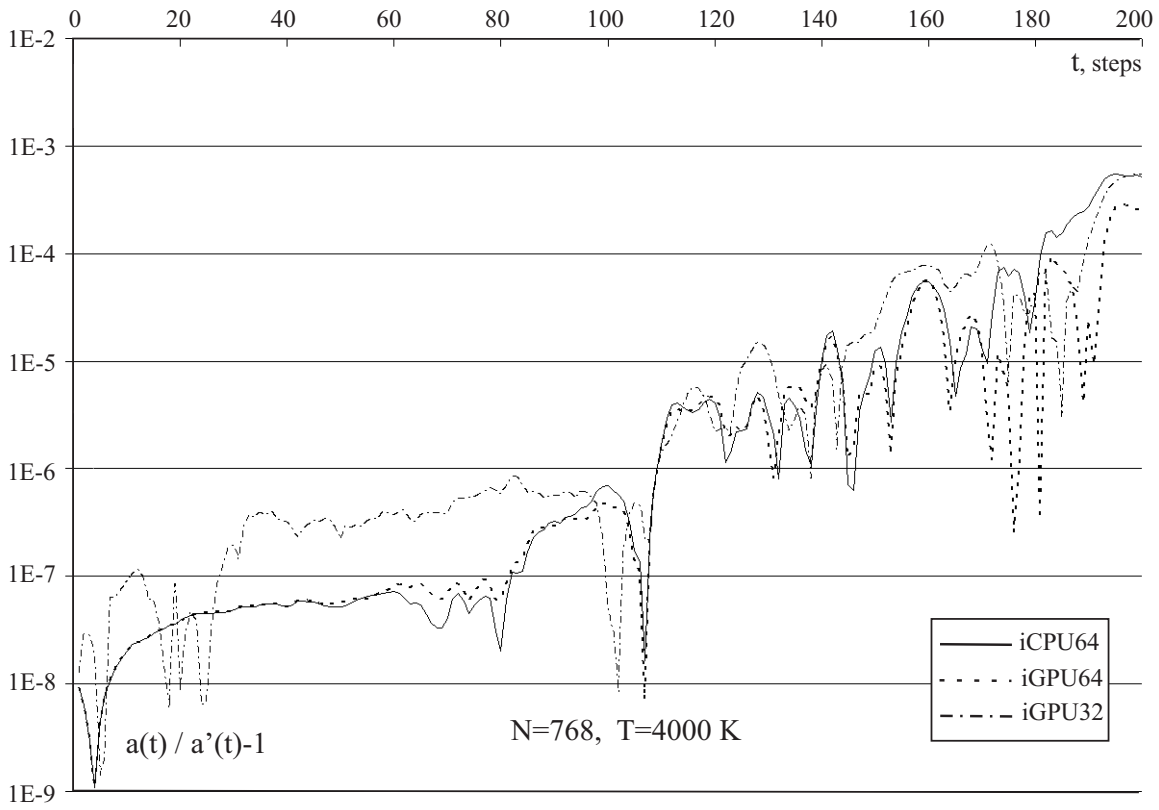

Рис. 1. Динамика погрешности CUDA-реализаций a(t) относительно расчетов на CPU a'(t)

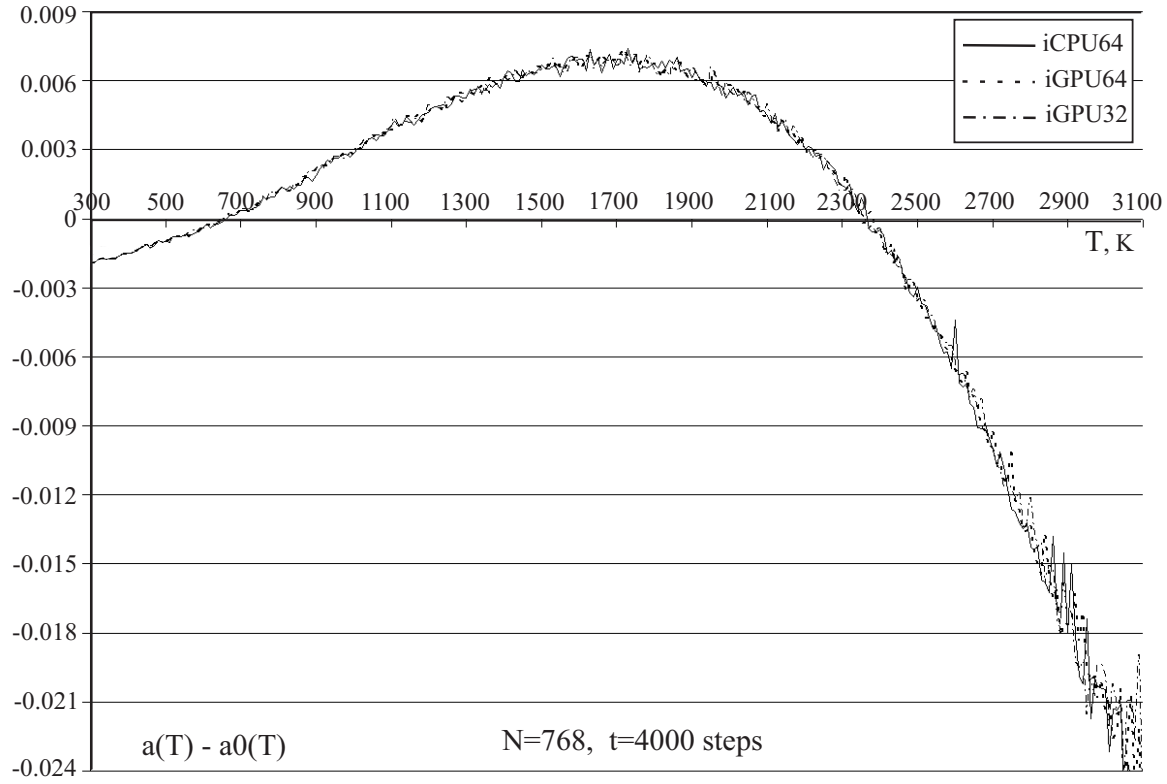

Рис. 2. Сравнение температурной зависимости отклонения периода решетки a(T) от экспериментального a0(T) для CUDA-реализаций

МД-эксперимента).

На рис. 1 и 2 приведены результаты моделирования с тремя вышеупомянутыми реализациями интегрирования. На первом приведен график относительной погрешности вычисления периода решетки от времени (a'(t) — эталонный период, полученный на CPU-версии с двойной точностью арифметики для всех операций), из которого видно, что версии iGPU64 и iCPU64 дают одинаковую погрешность, а отличия от них версии iGPU32 незначительны. На втором приведен график с температурной зависимостью периода решетки, полученный в результате ≈ 300 МД-экспериментов по 4000 шагов каждый, из которого видно, что зависимости для разных версий совпадают за исключением случайных колебаний (амплитуда отклонений на всех кривых примерно одинакова, систематических отличий нет).

В результате можно сделать вывод, что для нашей задачи восстановления межчастичных потенциалов, в которой целевая функция определяется температурной зависимостью периода решетки, версия iGPU32 имеет достаточную точность. Точность обеих версий интегрирования с 64-битной арифметикой можно считать эквивалентной.

Таблица 1

Время в секундах одного шага МД при ПГУ для разных версий

Язык Интегрирование	C++ iCPU64	C++[1] iCPU64	HLSL iCPU64	CUDA iCPU64	CUDA iGPU64	CUDA iGPU32
1×324	3.43E-02	2.43E-02	2.42E-03	4.91E-04	6.05E-04	4.89E-04
30×324	—	—	—	5.92E-05	4.24E-05	3.98E-05
60×324	—	—	—	5.79E-05	4.05E-05	3.86E-05
Ускорение, разы	1.0	1.41	14.2	592	845	888
1×768	1.24E-01	1.00E-01	3.67E-03	1.65E-03	1.80E-03	1.62E-03
30×768	—	—	—	2.05E-04	1.64E-04	1.59E-04
60×768	—	—	—	2.09E-04	1.62E-04	1.57E-04
Ускорение, разы	1.0	1.24	33.7	593	761	787

7. Сравнение быстродействия. Все измерения проведены при помощи функции QueryPerformanceCounter из Windows API (Application Programming Interface) на следующей программно-аппаратной конфигурации: CPU Intel Core2 Quad Q9550 (2833 МГц), 3 ГБ DDR3 (1333 МГц, 128-битная шина); NVIDIA GeForce GTX 280 (1296 МГц — частота скалярных процессоров), 1 ГБ GDDR3 (2214 МГц, 512-битная шина); Microsoft Windows XP Service Pack 3; NVIDIA Forceware 180.48 (11/2008); Microsoft .NET Framework 3.5; Microsoft DirectX 9.0c (4.09.0000.0904, 12/2008); NVIDIA CUDA 2.0 (Toolkit + SDK); Microsoft C/C++ compiler 14.0 (в составе Visual Studio 2005 Service Pack 1), настройки оптимизации по умолчанию, кроме значения "fast" в пункте "floating point model".

Версия с распараллеливанием по системам отмечена в табл. 1 и 2 приставкой "30×", что соответствует одновременному моделированию 30-и систем (каждая из которых обрабатывается на своем мультипроцессоре GPU GeForce GTX 280). Так как в CUDA возможно выполнение двух связок потоков на одном МП, мы также рассматриваем версию с одновременным моделированием 60-и систем (приставка "60×"). Время в табл. 1 и 2 для этих версий нормировано на одну систему. Все расчеты на CPU проводились одним процессорным ядром и без использования SSE (т.е. скалярная реализация), с двойной точностью плавающей арифметики, так как использование одинарной в нашем случае не давало прироста скорости. Отсутствие ускорения связано с особенностями компилятора от Microsoft, так как с компилятором Intel C/C++ compiler 11.0 использование арифметики одинарной точности обеспечивает прирост 25–45%.

Обсудим сначала версии с параллелизмом только по частицам (т.е. S=1). Наша предыдущая GPU-реализация [14], использующая язык шейдеров HLSL и библиотеку DirectX [31], — наиболее простая, так как вычисления автоматически распределяются драйвером видеокарты. При этом из табл. 1 видно, что SSE-реализация на 4-ядерном CPU (например, [22]) будет сравнима с ней по скорости (для оценки можно разделить время скалярной CPU-версии на 16). Однако разработка SSE-версии существенно сложнее:

[1]Реализация с нашим алгоритмом суммирования в обратном пространстве

Таблица 2

Время в секундах разных этапов расчета одного шага МД

Этап / S×N	1×324	30×324	60×324	1×768	30×768	60×768
Суммирование взаимодействий методом Эвальда на GPU, 32 бита						
kernel_N×N	3.38E-04	2.40E-05	2.34E-05	1.35E-03	1.23E-04	1.23E-04
kernel_K×N	5.95E-05	5.19E-06	5.06E-06	1.22E-04	1.20E-05	1.19E-05
kernel_N×K	3.08E-05	7.54E-06	7.45E-06	6.37E-05	1.68E-05	1.66E-05
Интегрирование на CPU, 64 бита (iCPU64)						
GPU→CPU	2.71E-05	3.31E-06	3.19E-06	3.12E-05	7.50E-06	7.35E-06
CPU Integration	2.74E-05	1.50E-05	1.51E-05	6.62E-05	3.62E-05	4.02E-05
CPU→GPU	1.92E-05	5.02E-06	4.84E-06	4.47E-05	1.55E-05	1.55E-05
Интегрирование на GPU, 64 бита (iGPU64)						
kernel_Integrate64	1.34E-04	4.17E-06	3.68E-06	1.97E-04	7.81E-06	7.14E-06
(64) GPU→CPU	2.56E-05	8.76E-07	4.41E-07	2.55E-05	8.78E-07	4.43E-07
Интегрирование на GPU, 32 бита (iGPU32)						
kernel_Integrate32	4.12E-05	1.61E-06	1.07E-06	6.41E-05	2.69E-06	1.82E-06
(32) GPU→CPU	2.55E-05	8.77E-07	4.38E-07	2.55E-05	8.77E-07	4.43E-07

ассемблерный код, ручное распараллеливание, тонкости эффективной работы с памятью и многоуровневой системой кэшей (отличающиеся для процессоров Intel и AMD). Реализация на CUDA также требует учитывать детали аппаратной архитектуры, зато дает дополнительное ускорение в 2–5 раз (которое уменьшается с ростом числа частиц). Наконец, CUDA-реализация с параллелизмом по системам (S=30) обеспечивает прирост еще в 8 раз, при этом удвоение числа связок потоков (S=60) дает лишь 2% скорости и только при N=324.

Интегрирование на GPU с 32-битной арифметикой быстрее CPU-интегрирования в 9–17 раз при S=30 и в 15–28 раз при S=60. При этом общее время расчетов уменьшается на 50% для N=324 и на 33% для N=768. Использование 64-битной арифметики замедляет GPU-интегрирование в 2.6–2.9 раза при S=30 и в 3.5–3.9 раза при S=60, однако общее время увеличивается лишь на 6% для N=324 и на 3% для N=768. При этом можно реализовать смешанный вариант интегрирования, по точности близкий к версии iGPU64, а по скорости — к iGPU32.

При одновременном моделировании S систем (30-и и 60-и в нашем случае, когда на GPU GeForce 280 GTX каждый МП обрабатывает по одной или две связки потоков) увеличивается плотность вычислений по отношению к операциям ввода-вывода и декодированию инструкций, в частности, по результатам табл. 2 можно отметить следующие моменты. Ядра kernel_NxN и kernel_KxN ускоряются в 10–14 раз, а ядро kernel_NxK — в 4 раза, так как параллельная обработка множества систем позволяет задействовать все МП (с достаточным числом потоков) любого GPU. Ядро kernel_Integrate32 ускоряется в 24–39 раз, а kernel_Integrate64 в 25–36, так как одну систему наиболее выгодно интегрировать на одном МП из-за необходимости синхронизации потоков (см. раздел 6) и, следовательно, распараллеливание по системам — единственный способ ускорения этого этапа; в случае интегрирования на CPU также наблюдается ускорение на $\approx 80\%$ за счет кэширования инструкций и данных. Наибольшее ускорение достигается на этапе копирования макропараметров на CPU (после интегрирования на GPU), которое почти линейно масштабируется от числа систем (≈ 58 раз для 60 систем), так как данных передается мало и большая часть времени при S=1 уходит на ожидание, связанное с латентностью шины PCI-Express. Ускорение этапов копирования в случае интегрирования на CPU — всего лишь 3–8.5 раз из-за большего объема передаваемых данных (вместо макропараметров каждой системы — координаты и силы каждой частицы).

Доля времени, приходящаяся на этап интегрирования на CPU (с учетом копирования), растет с увеличением числа систем: от 15% при S=1 до 40% при S=30 или S=60 в случае N=324. С дальнейшим

увеличением количества мультипроцессоров на GPU и, соответственно, числа одновременно моделируемых систем эта доля станет доминирующей в общем времени расчетов, что подтверждает необходимость параллельной реализации данного этапа на GPU.

Обсудим доли разных этапов в общем времени расчетов при моделировании 60-и систем исключительно на GPU (т.е. версии с интегрированием iGPU32 и iGPU64) по результатам табл. 2 (столбцы 60×324 и 60×768). При интегрировании на GPU за каждый шаг МД передаются на CPU только макропараметры систем, что занимает лишь $\approx 1\%$ времени. Возможна также реализация, в которой они накапливаются в памяти GPU за несколько шагов МД и затем передаются на CPU пакетно. Само интегрирование на GPU с 32-битной арифметикой занимает 1–3% времени, а с 64-битной арифметикой — 4–9% при N=768 и N=324 соответственно. В случае интегрирования на GPU с 32-битной арифметикой в целом на три ядра метода Эвальда приходится $\approx 95\%$ общего времени. Наиболее длительный этап расчетов — ядро kernel_NxN, которое занимает 61% при N=324 и 78% при N=768 (его доля растет с числом частиц из-за квадратичной вычислительной сложности). При этом двойной цикл по частицам на GPU не является "треугольным", то есть выполняется в 2 раза больше вычислений, чем на CPU (см. детальное обсуждение этого вопроса в работе [8]). Суммирование в обратном пространстве (ядра kernel_KxN и kernel_NxK) занимает 32% времени при N=324 и 18% при N=768.

Таким образом, мы реализовали полный цикл вычислений одного шага МД на GPU, оставив на CPU только ввод параметров, вызов нескольких ядер CUDA и вывод результатов. При этом доля времени на операции, не связанные с методом Эвальда, сократилась с 40% до $\approx 5\%$. В принципе, возможна реализация, выполняющая последовательность из нескольких МД-шагов (возможно, весь МД-эксперимент) за один вызов CUDA. Однако в нашем случае ускорение от более редкого копирования макропараметров с GPU на CPU будет невелико (см. на два абзаца выше), а код значительно усложнится за счет объединения всех ядер в одно. Так как доля времени, приходящаяся на суммирование в обратном пространстве, — 18–32% (и уменьшается с ростом числа частиц), то его ускорение в два раза за счет кэширования тригонометрических операций обеспечит не более 10–15% прироста общего быстродействия. Поэтому мы считаем более перспективной оптимизацию ядра kernel_NxN (доля которого 61–78% и растет с числом частиц), в частности эффективную реализацию "треугольного цикла" (см. обсуждение в [8]).

8. Заключение. Задачи молекулярной динамики (МД) требуют огромных вычислительных ресурсов, которые наиболее выгодно извлекать из графических процессоров (GPU), имеющих наилучшее соотношение цена–производительность среди доступных параллельных архитектур с общей памятью (см. введение). В предыдущей статье [8] мы обсудили детали параллельной реализации прямого суммирования парных взаимодействий для МД-моделирования больших открытых систем (10^4–10^8 частиц) на GPU с поддержкой технологии NVIDIA CUDA [9]. В данной работе мы исследовали особенности моделирования малых периодических систем (10^2–10^4 частиц) и соответствующего метода расчета взаимодействий — суммирования Эвальда [12].

Рассмотрены два способа кэширования ресурсоемких тригонометрических операций на этапе суммирования в обратном пространстве (Фурье). На CPU предложенная нами реализация незначительно увеличивает объем используемой памяти (8×N байт) и дает двукратное ускорение данного этапа, тогда как более сложная реализация [22] ускоряет его еще в 2.5 раза, но требует намного большего объема памяти для дополнительных массивов (184×N байт). Параллельные реализации на GPU обоих вариантов кэширования оказались неэффективными из-за недостаточного объема быстрой (разделяемой) памяти, однако наш вариант может стать актуальным в ближайшей перспективе (см. обсуждение в разделе 5.1).

Параллельный расчет взаимодействий одной малой периодической системы (по частям) не масштабируется, и при большом количестве вычислителей (которое на GPU удваивается за год [4]) аппаратные ресурсы будут задействованы неэффективно. Напротив, предложенный в данной работе подход с одновременным расчетом множества таких систем масштабируется линейно (по закону Густафсона [28]). В нашем случае он обеспечивает ускорение почти на порядок (в 8 раз) по сравнению с наиболее быстрой реализацией первого подхода. Хорошим примером практической задачи, где параллелизм по системам приводит к наибольшему эффекту, является задача глобальной оптимизации, в которой ресурсоемкая целевая функция вычисляется на GPU.

В любых параллельных реализациях МД в первую очередь стремятся распределить расчет парных взаимодействий из-за его квадратичной $O(N^2)$ вычислительной сложности, а линейными $O(N)$ этапами (интегрированием динамики, коррекциями, расчетом микро- и макропараметров и др.), которые на системах порядка 10000 частиц и выше занимают менее 1% общего времени, часто пренебрегают. Тем не менее, на системах порядка 1000 частиц они занимают до 40% времени (см. раздел 7) и, оставаясь нераспределенными, нарушают масштабируемость программы (по закону Амдала [27]). Их распараллеливание

на CPU мы считаем неперспективным в связи с коммуникационными издержками на передачу характеристик всех частиц на каждом шаге МД и опережающим развитием GPU [8], поэтому реализовали полный цикл вычислений одного шага МД на CUDA. При этом доля линейных этапов в общем времени расчетов уменьшилась с 40% до $\approx 5\%$, а ускорение по сравнению с версией, в которой на CUDA считаются только парные взаимодействия, составило 40–60%, и на будущих GPU оно будет только расти.

На современных GPU пиковое быстродействие для плавающей арифметики двойной (64-битной) точности медленнее одинарной (32-битной) не в два раза, а на порядок [7]. Поэтому, по возможности, ее следует избегать. Тем не менее, эксперименты показали, что для нашей задачи (МД-восстановления эмпирических межчастичных потенциалов) одинарной точности достаточно для всех вычислений, а в случае использования 64-битной арифметики на линейных этапах общее время увеличивается лишь на 3–6%. При этом обе реализации с 64-битной арифметикой (CPU и GPU) обеспечили эквивалентную точность расчетов.

В итоге предлагаемая CUDA-реализация МД для периодических граничных условий (с суммированием Эвальда и интегрированием на GPU) при параллельном моделировании 60 систем из 324 частиц на видеокарте NVIDIA GeForce GTX 280 в ≈ 60 раз быстрее нашей предыдущей GPU-реализации [14], которая использует библиотеку DirectX и язык HLSL [31], и в ≈ 890 раз быстрее аналогичной скалярной CPU-реализации (ее код на языке C++ приведен в статье), выполняющейся на процессоре Intel Core2 Quad Q9550.

Достигнутую масштабируемость мы использовали при разработке пакета программ для распределенных вычислений, который решает задачу МД-восстановления эмпирических межчастичных потенциалов при помощи методов глобальной оптимизации. На данный момент он объединяет ресурсы 8-и GPU с суммарной пиковой производительностью более 4 TFLOPS. В результате нам удалось получить набор потенциалов для моделирования оксидного ядерного топлива, который воспроизводит экспериментальные данные лучше, чем любые предложенные ранее [14, 26]. В перспективе мы планируем открыть доступ к этому проекту через Интернет, чтобы любой желающий при наличии GPU с поддержкой CUDA мог участвовать в распределенных вычислениях (аналогично проекту Folding@Home [29]).

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, J. Grotendorst, D. Marx, A. Muramatsu (Eds.), NIC Series, 2002. Vol. 10. 467–506 (http://www.fz-juelich.de/nic-series/volume10/gibbon.pdf).
4. 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).
5. http://en.wikipedia.org/wiki/Cell_(microprocessor)
6. NVIDIA GeForce GTX 200 GPU Datasheet (http://www.nvidia.com/docs/IO/55506/ GeForce_GTX_GPU_Datasheet.pdf).
7. NVIDIA Tesla personal supercomputer (http://www.nvidia.com/docs/IO/43395/ NV_DS_Tesla_PSC_US_Dec08_LowRes.pdf).
8. *Боярченков А.С., Поташников С.И.* Использование графических процессоров и технологии CUDA для задач молекулярной динамики // Вычислительные методы и программирование. 2009. **10**. 9–23 (http://num-meth.srcc.msu.ru/zhurnal/tom_2009/v10r102.html).
9. NVIDIA CUDA Documentation (http://www.nvidia.com/object/cuda_develop.html).
10. MD-GRAPE Technical Report (http://www.research.ibm.com/grape/grape_mdgrape2.htm).
11. http://en.wikipedia.org/wiki/CUDA#Supported_GPUs
12. http://en.wikipedia.org/wiki/Ewald_summation
13. http://en.wikipedia.org/wiki/Molecular_dynamics#Empirical_potentials
14. *Поташников С.И., Боярченков А.С., Некрасов К.А., Купряжкин А.Я.* Молекулярно-динамическое восстановление межчастичных потенциалов в диоксиде урана по тепловому расширению // Международный научный журнал "Альтернативная энергетика и экология". 2007. **8**. 43–52 (http://isjaee.hydrogen.ru/pdf/AEE0807/AEE08-07_Potashnikov.pdf).
15. http://en.wikipedia.org/wiki/Virial_theorem
16. http://en.wikipedia.org/wiki/Equations_of_motion

17. http://en.wikipedia.org/wiki/Semi-implicit_Euler_method
18. *Berendsen H., Postma J., van Gunsteren W., Dinola A., Haak J.* Molecular dynamics with coupling to an external bath // Journal of Chemical Physics. 1984. **81 (8)**. 3684–3690.
19. *Press W.H., Teukolsky S.A., Vetterling W.T., Flannery B.P.* Numerical recipes in C. The art of scientific computing (http://www.numerical-recipes.com/).
20. Intel 64 and IA-32 Architectures Optimization Reference Manual (http://download.intel.com/design/processor/manuals/248966.pdf).
21. *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).
22. GROMACS: Fast, Free and Flexible MD (http://www.gromacs.org).
23. *Owens J.* Streaming architectures and technology trends // GPU Gems 2. 2005 (http://download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch29.pdf).
24. *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).
25. *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).
26. *Поташников С.И., Боярченков А.С.* Атомарные межчастичные потенциалы для моделирования смешанного оксидного ядерного топлива (готовится в печать).
27. http://en.wikipedia.org/wiki/Amdahl's_law
28. http://en.wikipedia.org/wiki/Gustafson's_Law
29. Проект распределенных вычислений Folding@Home (http://fah-web.stanford.edu/cgi-bin/main.py?qtype=osstats).
30. *Makino J., Fukushige T., Koga M., Namura K.* GRAPE-6: The massively-parallel special-purpose computer for astrophysical particle simulations // Publications of the Astronomical Society of Japan. 2003. **55**, N 6 (http://arxiv.org/abs/astro-ph/0310702v1).
31. DirectX Developer center (http://msdn.microsoft.com/en-us/directx/default.aspx).

Приложение.

```
// Код ядер метода Эвальда на CUDA для раздела 5.2.
// Константы: N = 324, K = 462, BI = BJ = NxK_BLOCKS = KxN_BLOCKS = 6, NJ = 54.
__global__ void kernel_NxN(float4 force[], float4 pos[], int type[], float4 coefs[],
float L, float W)
{
  int i = threadIdx.x + blockIdx.x * blockDim.x, type_i = type[i];
  int j = threadIdx.x, j0 = blockIdx.y * NJ;
  float4 pos_i = pos[i], force_i = { 0, 0, 0, 0 };

  __shared__ float4 pos_j[NJ], coefs_ij[3];
  __shared__ int type_j[NJ];
  if (j < 3) coefs_ij[j] = coefs[j];
  pos_j[j] = pos[j + j0]; type_j[j] = type[j + j0];
  __syncthreads();

  for (j = 0; j < NJ; ) // вручную разворачиваем 9 итераций цикла
  {
    float4 c = coefs_ij[type_i + type_j[j]];
    force_i += Fij(pos_i, pos_j[j], c, L, W); j++; // 1-я итерация
    ...
    force_i += Fij(pos_i, pos_j[j], c, L, W); j++; // 9-я итерация
  }
  force[i + blockIdx.y * N] = force_i;
}
__global__ void kernel_KxN(float4 force[], float2 QExp[], float4 k[], float4 pos[],
float Pi2_L, float KePi8_LLL, float _4WW)
{
  const int memThreadRatio = 5; // ceiling(324 * 6 / 462) = 5
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int j = threadIdx.x * memThreadRatio;
  float QCos = 0, QSin = 0, KR, factor;
  float4 k_i = k[i] * Pi2_L; k_i.w *= Pi2_L;
```

```
  __shared__ float4 pos_j[N];
  for (int m = 0; m < min(memThreadRatio, N - j); m++) pos_j[j + m] = pos[j + m];
  __syncthreads();

  #pragma unroll 108
  for (int j = 0; j < N; j++)
  {
    KR = dot3(k_i, pos_j[j]);
    QCos += pos_j[j].w * cosf(KR);
    QSin += pos_j[j].w * sinf(KR);
  }
  factor = KePi8_LLL * expf(-k_i.w * _4WW) / k_i.w;
  force[i].w += factor * (1 - 2 * k_i.w * _4WW) * (QCos * QCos + QSin * QSin);
  QExp[i].x = factor * QCos;
  QExp[i].y = factor * QSin;
}
__global__ void kernel_NxK(float4 force[], float2 QExp[], float4 k[], float4 pos[],
float Pi2_L)
{
  const int memThreadRatio = 9; // ceiling(462 * 6 / 324) = 9
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int j = threadIdx.x * memThreadRatio;
  float4 pos_i = pos[i], force_i = { 0, 0, 0, 0 };

  __shared__ float3 k_j[K];
  __shared__ float2 QExp_j[K];
  for (int m = 0; m < min(memThreadRatio, K - j); m++)
  {
    k_j[j + m] = make_float3(k[j + m].x, k[j + m].y, k[j + m].z) * Pi2_L;
    QExp_j[j + m] = QExp[j + m];
  }
  __syncthreads();

  #pragma unroll 77
  for (j = 0; j < K; j++)
  {
    float KR = dot3(k_j[j], pos_i);
    mad3(force_i,
      k_j[j], pos_i.w * (sinf(KR) * QExp_j[j].x - cosf(KR) * QExp_j[j].y));
  }
  force[i] += force_i;
}
__global__ void kernel_sum(float4 force[])
{
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  float4 sum = { 0, 0, 0, 0 };
  for (int j = 0; j < BJ; j++) sum += force[i + j * N];
  force[i] = sum;
}
// Код расчета взаимодействий в ПГУ методом Эвальда с вызовом CUDA-ядер.
void Ewald(double3 pos[], int type[], double charge[], double **coefs[], double L,
double3 force[], double *Stress)
{
  float Pi2_L = (float)(M_PI * 2 / L), W = Pi2_L, _4WW = 1 / (4 * W * W);
  float KePi8_LLL = (float)(Ke * M_PI * 8 / (L * L * L)); // Ke - константа Кулона
```

```
  CopyToGPU(pos, type, charge, coefs);

  // Вычисления в прямом пространстве.
  kernel_NxN<<< dim3(BI, BJ), N / BI >>>(force_gpu, pos_gpu, type_gpu, coefs_gpu, L/2, W);
  // Вычисления в обратном пространстве.
  kernel_KxN<<< KxN_BLOCKS, K / KxN_BLOCKS >>>(force_gpu, QExp_gpu, k_gpu,
  pos_gpu, Pi2_L, KePi8_LLL, _4WW);
  kernel_NxK<<< NxK_BLOCKS, N / NxK_BLOCKS >>>(force_gpu, QExp_gpu, k_gpu,
  pos_gpu, Pi2_L);
  // Редукция элементов массива force_gpu
  kernel_sum<<< BI, N / BI >>>(force_gpu);

  *Stress = 0.5 * CopyFromGPU(force); // Stress удвоенный, в отличие от CPU.
}
// Код МД-шага в ПГУ с коррекцией импульса, термостатом и баростатом Берендсена.
void UpdatePBC()
{
  Ewald(pos, type, charge, coefs, L, force, &Stress); // Stress = 3*P*V - 2*Ekin
  for (int i = 0; i < N; i++) vel[i] += force[i] * (dt / mass[type[i]]);

  // Вычисляем массу, температуру, удвоенную кин. энергию и импульс системы.
  double Mass = 0, T_system = 0, Ekin2 = 0;
  double3 Impulse = double3::Empty;
  for (int i = 0; i < N; i++)
  {
    double mi = mass[type[i]];
    Mass += mi; Impulse += mi * vel[i]; Ekin2 += mi * (vel[i] * vel[i]);
  }
  Impulse /= Mass; T_system = Ekin2 / 3 * Kb * N; // Kb - константа Больцмана.

  for (int i = 0; i < N; i++) vel[i] -= Impulse; // Корректируем импульс.

  // Применяем термостат, интегрируем позиции, применяем баростат и ПГУ.
  double P_system = (Stress + Ekin2) / (L * L * L * 3);
  double n = pow(1 + (P_system - P_stat) / tau_in_steps, 1 / 3.0);
  L *= n;
  for (int i = 0; i < N; i++)
  {
    vel[i] *= sqrt(1 + (T_stat / T_system - 1) / tau_in_steps);
    pos[i] = (pos[i] + vel[i] * dt) * n;
    if (pos[i].x > L/2) pos[i].x -= L; else if (pos[i].x < -L/2) pos[i].x += L;
    if (pos[i].y > L/2) pos[i].y -= L; else if (pos[i].y < -L/2) pos[i].y += L;
    if (pos[i].z > L/2) pos[i].z -= L; else if (pos[i].z < -L/2) pos[i].z += L;
  }
}
```

Поступила в редакцию
20.02.2009