КОМПЬЮТЕРНЫЕ ИССЛЕДОВАНИЯ
И МОДЕЛИРОВАНИЕ 2011 Т. 3 № 1 С. 287–308

Ки&М

ЧИСЛЕННЫЕ МЕТОДЫ И ОСНОВЫ ИХ РЕАЛИЗАЦИИ

УДК: 004.942

Эффективные генераторы псевдослучайных чисел при молекулярном моделировании на видеокартах

А. А. Жмуров[1,2], В. А. Барсегов[1,2], С. В. Трифонов[1,a], Я. А. Холодов[1], А. С. Холодов[1]

[1] Московский физико-технический институт, г. Долгопрудный, Московская область, Россия, 141700
[2] Химический факультет, Массачусетский Университет, Ловелл, MA 01854

E-mail: [a] svtrifonov@gmail.com

Получено 21 июля 2011 г.

Динамика Ланжевена, метод Монте-Карло и моделирование молекулярной динамики в неявном растворителе требуют больших массивов случайных чисел на каждом шаге расчета. Мы исследовали два подхода в реализации генераторов на графических процессорах. Первый реализует последовательный алгоритм генератора на каждом потоке в отдельности. Второй основан на возможности взаимодействия между потоками и реализует общий алгоритм на всех потоках в целом. Мы покажем использование этих подходов на примере алгоритмов Ran 2, Hybrid Taus и Lagged Fibonacci. Для проверки случайности полученных чисел мы использовали разработанные генераторы при моделировании динамики Ланжевена N независимых гармонических осцилляторов в термостате. Это позволило нам оценить статистические характеристики генераторов. Мы также исследовали производительность, использование памяти и ускорение, получаемое при переносе алгоритма с центрального на графический процессор.

Ключевые слова: псевдослучайные числа, графический процессор, генератор, молекулярное моделирование

Efficient Pseudorandom number generators for biomolecular simulations on graphics processors

A. A. Zhmurov[1,2], V. A. Barsegov[1,2], S. V. Trifonov[1], Ya. A. Kholodov[1], A. S. Kholodov[1]

[1]Moscow Institute of Physics and Technology, Dolgoprudnyi, Moscow region, Russia, 141700
[2]Department of Chemistry, University of Massachusetts, Lowell, MA 01854

Abstract. — Langevin Dynamics, Monte Carlo, and all-atom Molecular Dynamics simulations in implicit solvent require a reliable source of pseudorandom numbers generated at each step of calculation. We present the two main approaches for implementation of pseudorandom number generators on a GPU. In the first approach, inherent in CPU-based calculations, one PRNG produces a stream of pseudorandom numbers in each thread of execution, whereas the second approach builds on the ability of different threads to communicate, thus, sharing random seeds across the entire device. We exemplify the use of these approaches through the development of Ran2, Hybrid Taus, and Lagged Fibonacci algorithms. As an application-based test of randomness, we carry out LD simulations of N independent harmonic oscillators coupled to a stochastic thermostat. This model allows us to assess statistical quality of pseudorandom numbers. We also profile performance of these generators in terms of the computational time, memory usage, and the speedup factor (CPU/GPU time).

Keywords: GPU, pseudorandom numbers, PRNG, biomolecules

Введение

За последние несколько лет графические процессоры (GPU) преобразовались в высоко параллельные мультипоточные расчетные устройства, так что GPU сейчас становятся альтернативной платформой для разработки высокопроизводительных научных вычислений. Поскольку в рассматриваемых нами GPU используется Single Instruction Multiple Data (SIMD) архитектура, в которой кэш-память и управление потоками данных для группы вычислительных ядер объединены в одно устройство, современные GPU хорошо подходят для вычислительных расчетов. На сегодняшний день пиковая вычислительная производительность GPU превышает 1 TFlops для одного чипа [NVIDIA CUDA Programming Guide, 2009] — величина, недостижимая для большинства современных CPU. А с появлением языка CUDA (Compute Unified Device Architecture, расширение C и C++) компании NVIDIA стало возможным использовать GPU для сложных вычислительных задач. Поскольку некоторые алгоритмы вычислений работают на GPU более в чем в сто раз быстрее, чем на CPU, с помощью GPU можно ускорять различные высокопроизводительные приложения — от гидродинамических расчетов до молекулярной динамики с астрофизикой [Stone et al., 2007; Friedrichs et al., 2009; Anderson, Schröder, 2007; Yang et al., 2007]. Расчеты на GPU происходят одновременно на многих вычислительных ядрах, арифметико-логических устройств (АЛУ), объединенных в мультипроцессоры, каждый из которых имеет собственный кэш и очередь выполнения. К примеру, в современных графических картах NVIDIA каждый мультипроцессор состоит из 8-ми 1,3 ГГц АЛУ, 14–16 Кб кэш (*Кб — константная память и 6–8 Кб — глобальный кэш текстурной памяти). Число мультипроцессоров достигает 30 на современных GPU (TESLA C1060), а следовательно, 240 АЛУ на чип.

Проектирование параллельного алгоритма для GPU имеет ряд особенностей. Во-первых, для оптимизации производительности должно быть минимизировано для каждого АЛУ, число обращений к глобальной памяти, несмотря на то что глобальная память графического процессора имеет пропускную способность, в 10 раз большую, чем центрального. Таким образом, в связи с изначально параллельной природой GPU-вычислений оптимальная производительность достигается в том случае, когда задача делится на независимые потоки, выполняющие одни и те же действия с различными данными. Во-вторых, задача должна быть расчетоемкой, чтобы основное время GPU был занят реальными расчетами, а не чтением и записью данных в память [NVIDIA CUDA Programming Guide, 2009; NVIDIA CUDA C Programming Best Practices Guide, 2009].

Вышесказанное делает задачу N тел, очень сложную, часто нерешаемую, первым кандидатом для численной реализации на GPU. Вообще, модели системы N частиц, которые используют динамику Ланжевена (ЛД), Монте-Карло (МК) и моделирование полноатомной молекулярной динамики (МД), принадлежат к классу эффективно реализуемых на GPU задач. В моделировании ЛД и МД атомные взаимодействия описываются одинаковыми для всех частиц системы потенциалами (силовыми полями). Можно сказать, что есть прямая связь между SIMD-архитектурой GPU и численными методами, используемыми для определения траектории изучаемой системы. Действительно, single instruction (расчет потенциала, случайной силы, численное интегрирование уравнений движения) исполняются на multiple data (для всех частиц). Приведем несколько примеров моделей описания межатомных потенциалов.

При моделировании МД биомолекул в неявном растворителе [Brooks et al., 1983; Haberthür, Caflisch, 2008] динамика i-й частицы описывается уравнениями для координаты и скорости частицы:

$$\frac{dR_i}{dt} = V_i,$$

$$m_i \frac{dV_i}{dt} = \xi V_i + f(R_i) + G_i(t)(V_i = V_{ix}, V_{iy}, V_{iz}),$$

где $R_i = \{R_{ix}, R_{iy}, R_{iz}\}$ — радиус-вектор частицы, m_i — масса частицы, ξ — коэффициент трения, $G_i(t) = \{G_{ix}, G_{iy}, G_{iz}\}$ — распределенная по Гауссу случайная сила с математическим ожиданием, равным 0, и корреляционной функцией $(G_i(t), G_j(t')) = 2k_B T\delta_{ij}\,\delta(t-t')$, $i, j = 1..N$ [Risken, 1989], $f(R_i) = \dfrac{dU}{dR_i}$ — молекулярная сила, действующая на i-ю частицу, где $U = U(R_1, ..., R_N)$ — потенциальная энергия частицы.

При ЛД-моделировании протеина динамика i-й C_α частицы описывается уравнением для R_i:

$$\xi \frac{dR_i}{dt} = f(R_i) + G_i t \text{ [Doi, Edwards, 1988].}$$

При моделировании это уравнение движения решается численно для каждой из N частиц на каждой итерации.

Все эти методы требуют надежных источников $3N$ псевдослучайных нормально распределенных чисел, $g_{i\alpha}(i = 1..N)$. Так, при моделировании МД в неявном растворителе и ЛД-моделировании эффект взаимодействия с молекулами воды описывается неявно с помощью случайных сил, определяемых как $G_{i\alpha} = g_{i\alpha}\sqrt{2k_B T\xi\Delta t}$, где $\alpha \in (x, y, z)$. А при моделировании методом Монте-Карло, результаты многих независимых попыток, каждая из которых определяется неким случайным процессом, комбинируются, чтобы определить некоторый средний ответ.

Для этих целей PRNG производит последовательность случайных чисел u_i, равномерно распределенных внутри интервала [0; 1), которая имитирует последовательность независимых равномерно распределенных случайных величин. Эта последовательность (u_i) преобразуется в последовательность нормально распределенных случайных величин (g_i), используя, среди прочих, метод зиггурата [Tsang, Marsaglia, 2000], полярный метод [Marsaglia, Bray, 1964] и преобразование Бокса–Миллера [Box, Miller, 1958]. Причем PRNG должен иметь большой период и одновременно быть и быстрым, и генерировать случайные числа достаточного статистического качества.

Существует обширная литература, посвященная генерации случайных чисел на CPU [Press et al., 1992; L'Ecuyer et al., 1993]. Но, в связи с фундаментальными различиями в архитектуре CPU и GPU, адаптация методов, созданных для CPU, на GPU может оказаться затруднительной. Вдобавок, даже если будет реализован отдельный PRNG на GPU, для полного использования его ресурсов в комплексах молекулярного моделирования генератор должен взаимодействовать с основной программой. Это требует минимизировать число обращений к относительно медленной глобальной памяти компьютера и генерировать случайные числа, используя в основном более быструю разделяемую память карты.

В этой статье мы продемонстрируем новую методику генерации псевдослучайных чисел на графическом процессоре «на лету», т. е. на каждом шаге моделирования. Здесь мы рассмотрим линейный конгруэнтный генератор (LCG), Ran2, Hybrid Taus и алгоритм Фибоначчи, которые будут кратко описаны в следующем разделе.

Эти алгоритмы используются в разделе 3 для описания подходов one-PRNG-per-thread и one-PRNG-for-all-threads. В первом подходе один генератор используется для каждого потока (для каждой частицы). Именно этот подход обычно используется для расчетов на CPU. Второй использует возможность межпоточного взаимодействия внутри GPU (псевдокод можно посмотреть в Приложениях).

Предоставлены результаты тестирования производительности обоих подходов для GPU-приложений, где также дается оценка статистического качества генерируемых чисел с помощью ЛД-моделирования N независимых броуновских частиц, основанного на гармоническом

потенциале взаимодействия. Также была протестирована производительность этих генераторов и использование памяти как функции числа частиц в системе *N*.

Генераторы псевдослучайных чисел

Обзор

Существуют 3 типа генераторов псевдослучайных чисел: Истинный или аппаратный генератор (TRNG), Квазислучайные генераторы (QRNG) и Псевдослучайные генераторы (PRNG) [Nguyen, 2008]. В данной статье мы рассмотрим PRNG — наиболее общий тип детерминированных генераторов случайных чисел.

PRNG должен удовлетворять трем основным требованиям:

- хорошие статистические свойства,
- высокая производительность,
- низкое потребление памяти устройства.

Во-первых, сгенерированная последовательность случайных чисел должна проходить эмпирические тесты на случайность, такие как «строгие статистические тесты», например, на равномерность распределения, независимость (т. е. подпоследовательность четных элементов u_{2n} должна быть независима от подпоследовательности нечетных u_{2n+1}) и другие [Tsang, Marsaglia, 2000]. Кроме этих тестов, необходимо так же исследовать качество генерируемых случайных чисел, используя их при решении тестовых задач для конкретной модели молекулярной динамики (application-based test). Действительно, использование случайных чисел плохого статистического качества в моделировании молекулярной динамики может привести к нефизическим корреляциям или даже шаблонам [Selke et al., 1993; Grassberger, 1993] и ошибочным результатам [Ferrenberg et al., 1992]. Некоторые статистические и ориентированные на приложение тесты собраны в комплексы, такие как DIEHARD, SPRNG, и в библиотеки: TESTU01 [L'Ecuyer, Simard, 2007; Marsaglia, 2010; Mascagni, Srinivasan, 2000; Soto, 1999].

Во-вторых, хороший PRNG должен быть также вычислительно эффективен. Иными словами, PRNG не должен быть «узким местом» по производительности. К примеру, при моделировании ЛД биомолекулы может требоваться получение очень длинной траектории 0.1 с на протяжении 10^{10} итераций (т. е. с шагом по времени 10 пс). Следовательно, для моделирования этой траектории системе из 10^3 частиц потребуется 10^{13} случайных чисел.

И последнее требование — низкое использование памяти — становится особенно важным при реализации PRNG на GPU. Это связано с крайне малым объемом встроенной памяти у современных графических процессоров: ~ 20 Кб на мультипроцессор, для сравнения, CPU имеет в среднем 2 МБ.

Все вышесказанное означает, что эффективный PRNG алгоритм должен использовать в работе локальную память потока, избегая медленной глобальной памяти GPU. Обычно быстрый PRNG использует простые логические переменные и немного переменных для хранения текущего состояния, но это может нанести вред статистическим свойствам генерируемых случайных чисел. С другой стороны, использование более сложного алгоритма с большим количеством арифметических операций или объединяющего несколько генераторов в гибридный позволяет нам улучшить статистические показатели, но требует больше памяти и работает медленнее. Таким образом, выбор PRNG определяется целями конкретного приложения.

Кроме этих требований использование генераторов псевдослучайных чисел в моделировании молекулярной динамики имеет еще пару особенностей.

Во-первых, поскольку генерация псевдослучайных чисел — это детерминированный процесс, определенный некоторыми математическими преобразованиями, последовательность псевдослучайных чисел всегда возвращается к начальной точке, т. е. $\exists\, p$ такое, что $u_{n+p} = u_n$ [Barreira, 2006, pp. 415–422], после которой последовательность повторяется снова. Это означает, что PRNG имеет период длиной p. К примеру, если некоторый процесс моделирования использует 10^{12} случайных чисел, то период должен быть не меньше чем 10^{12}.

Во-вторых, при моделировании МД биомолекул в неявном растворителе и ЛД используется нормально распределенная случайная сила для эмуляции случайных столкновений с молекулами воды. Для генерации случайной силы используется следующий подход: последовательность случайных чисел u_i, распределенных равномерно, преобразуется в последовательность случайных величин, распределенных нормально. Для этого используются специальные преобразования. Здесь мы рассмотрим, не слишком подробно, преобразование Бокса–Миллера [Box, Miller, 1958].

В этой статье мы рассмотрим наиболее широко применяющиеся в биофизике генераторы: линейный конгруэнтный генератор (LCG) [Press et al., 1992], Ran2 [Press et al., 1992], Hybrid Taus [Press et al., 1992; Nguyen, 2008; Tausworthe, 1965; L'Ecuyer, 1996] и алгоритм Фибоначчи [Press et al., 1992; Mascagni, Srinivasan, 2004]. Алгоритм LCG обладает очень коротким периодом, но является достаточно быстрым. Ran 2 обычно используется при моделировании МД биомолекул в неявном растворителе и ЛД протеинов на CPU в связи с его длинным периодом ($p > 2*10^{18}$), хорошими статистическими свойствами и высокой производительностью (на CPU). Однако Ran2 при реализации требует большого количества памяти, как локальной, так и глобальной, для хранения текущего состояния. Hybrid Taus — это один из примеров, как несколько простых алгоритмов могут быть объединены для улучшения статистических характеристик генерируемых случайных чисел. Этот алгоритм лучше по производительности на GPU, чем KISS — хорошо известный комбинированный генератор [Marsaglia, 1999]. А длинный период Hybrid Taus ($p > 2*10^{36}$) делает его очень удобным для биомолекулярного моделирования на GPU. Другой хороший генератор — это алгоритм Фибоначчи с запаздыванием, который работает очень просто, в то время как генерируемые им случайные числа имеют очень хорошие статистические качества [L'Ecuyer, Simard, 2007]. Этот генератор обычно используется в распределенном моделировании методом МК и также может быть применен в расчетах на GPU. В последней части этого раздела мы кратко рассмотрим LCG, Ran 2, Hybrid Taus и алгоритм Фибоначчи с запаздыванием.

Линейный конгруэнтный генератор (LCG)

Линейные конгруэнтные генераторы (LCG) относятся к классическому и наиболее популярному типу генераторов, которые используют для генерации случайного числа x_n на n-м шаге, на основе x_{n-1}, полученного ранее, следующую формулу преобразования:

$$x_n = (a * x_{n-1} + c) \mod \mathrm{m}, \qquad (1)$$

где m — максимальный период, a = 1 664 525 и c = 1 013 904 223 — некоторые константные параметры [Press et al., 1992]. Обычно берут $m = 2^{32}$, поскольку предполагается, что 2^{32} — максимальное целое число для 32-разрядной системы, а значит, и максимально возможный период.

LCG имеет известные статистические закономерности [Tsang, Marsaglia, 2000]. Если $m = 2^{32}$, можно пренебречь операцией mod m, так как делимое меньше, чем 2^{32}, но это не так для x64 системы. В предположении x32 системы получаем формулу $x_n = a * x_{n-1} + c$, которая

называется RANqd2-генератор (упрощенный LCG). Это очень быстрый генератор, использующий очень мало памяти. В связи с его скоростью он может быть использован в сравнительных тестах производительности.

Алгоритм Ran2

Ran 2 — один из наиболее популярных PRNG. Алгоритм Ran2 объединяет два LCG-генератора и использует дополнительную процедуру рандомизации [Press, Teukolsky, Vetterling, 1992]. Ran 2 имеет длинный период и очень хорошие характеристики генерируемой последовательности [Tasng, Marsaglia, 2000]. Ran 2 — один из немногих генераторов, который проходит все известные статистические тесты. Ran 2 также является достаточно быстрым, что делает его удобным в использовании на CPU. Однако есть некоторые особенности Ran2, из-за которых он менее привлекателен для расчетов на GPU. Во-первых, это использование арифметики больших целых чисел (64-битной логики). Во-вторых, он требует большого количества памяти для хранения текущего состояния, которое к тому же должно обновляться после генерации каждого следующего случайного числа. Это приводит к частым обращениям к глобальной памяти, которые могут понизить производительность работы на кэшируемом GPU устройстве.

Алгоритм Hybrid Taus

Алгоритм Hybrid Taus [L'Ecuyer et al., 1993] являетя комбинированным генератором, который использует LCG и алгоритмы Tausworthe. Алгоритм Tausworthe, или taus88, — быстрый, равномерно распределяющий по модулю 2 генератор [Tausworthe 1965; L'Ecuyer, 1996]. В нем случайные числа получаются из генерируемой с помощью линейной рекурсии по модулю 2 последовательности бит, а результат формируется с помощью битовых блоков. В пространстве бинарных векторов n-й элемент вектора определяется с использованием рекуррентной формулы

$$y_n = a_1 y_{n-1} + a_2 y_{n-2} + ... + a_k y_{n-k}, \qquad (2)$$

где a_n — постоянные коэффициенты. На n-м шаге псевдослучайное число определяется с помощью формулы $x_n = \sum_{j=1}^{L} (y_{ns+j-1} 2^{-j})$, где s — натуральное число, а $L = 32$ — разрядность машинного слова. Генерация x_n требует s рекуррентных шагов, что может быть вычислительно затратно. Быстрая реализация может быть получена для некоторого набора параметров. Если $a_k = a_q = a_0 = 1$, где $0 < 2q < k,$ и $a_n = 0$ для $0 < s \le (k-q) < k \le L$, алгоритм может быть сведен к последовательности бинарных операций [L'Ecuyer, 1996].

Статистические характеристики случайных чисел, созданных алгоритмом taus88, достаточно низкие, но сочетание его с LCG устраняет все недостатки [L'Ecuyer et al., 1993]. В результате статистические свойства композитного генератора Hybrid Taus лучше, чем любого его компонента. Похожий подход, кстати, был использован в разработке KISS, который объединяет LCG, Tausworthe и пару multiple-with-carry-генераторов [Tausworthe, 1965]. Впрочем, множество 32-битных умножений, используемых KISS, может понизить его производительность на GPU.

Период Hybrid Taus — это наименьшее общее кратное периодов трех генераторов Tausworthe и одного LCG. Мы выбрали параметры таким образом, чтобы периоды были $p_1 = 2^{31} - 1, p_2 = 2^{30} - 1, p_3 = 2^{28} - 1$ для Tausworthe генераторов и $p_4 = 2^{32}$ — для LCG, что делает период композитного генератора равным $\approx 2^{121} > 10^{36}$. Генератор Hybrid Taus очень экономичен в использовании памяти. Ему требуется только 4 переменных типа integer для хранения текущего состояния.

Алгоритм Фибоначчи с запаздыванием

Алгоритм Фибоначчи с запаздыванием определяется рекуррентным соотношением

$$x_n = f(x_{n-sl}, x_{n-ll}) \bmod m, \tag{3}$$

где sl и ll — короткое и длинное запаздывание соответственно ($ll > sl$), m определяет максимальный период генератора, f — функция, которая принимает два целых числа x_{n-sl}, x_{n-ll}, для того чтобы определить x_n. Чаще всего используется произведение или сумма: $f(x_{n-sl}, x_{n-ll}) = x_{n-sl} * x_{n-ll}$ или $f(x_{n-sl}, x_{n-ll}) = x_{n-sl} + x_{n-ll}$ (аддитивный алгоритм Фибоначчи с запаздыванием). Для работы алгоритма требуется начальное множество ll случайных чисел.

Максимальный период алгоритма Фибоначчи с запаздыванием $\approx 2^{ll-1} * m$. Чтобы достичь максимального периода длинное запаздывание ll должно быть выбрано как основание Мерсеновской экспоненты, а короткое запаздывание должно быть выбрано так, чтобы характеристический полином $x^{ll} + x^{sl} + 1$ был прост. Также sl не должно быть слишком мало и не быть слишком приближенно к ll. Рекомендуется $sl \approx p * ll$, где $p \approx 0.618$ [Mascagni, Srinivasan, 2004]. Также следует отметить, что sl и ll влияют на статистические характеристики получаемых псевдослучайных чисел: чем больше sl и ll, тем характеристики лучше.

В случае арифметики одинарной точности операцией mod m можно пренебречь, считая $m = 2^{32}$ (больше о выборе параметров можно найти в [Mascagni, Srinivasan, 2004; Brent, 1992]).

В этой статье мы использовали аддитивный алгоритм Фибоначчи с запаздыванием. Одно из наиболее ценных его свойств — непосредственная генерация вещественных чисел.

Реализация LCG, Ran2, Hybrid Taus и алгоритма Фибоначчи с запаздыванием на GPU

Основные идеи

Эффективность расчетов на GPU определяется главным образом их многопоточной архитектурой. Т. е. вычисления на GPU производятся на нескольких параллельных потоках с помощью алгоритма, в котором одни и те же инструкции применяются к различным наборам данных. Поэтому, чтобы полностью использовать преимущества параллельных вычислений на GPU в молекулярном моделировании N тел, PRNG должен генерировать независимые случайные числа для всех частиц одновременно.

При этом можно хранить в CPU или GPU заранее сгенерированные случайные числа и затем использовать их в вычислениях. Но в этом случае для PRNG требуется выделить большой объем памяти. Например, для системы из 10^4 частиц в трехмерном пространстве требуется $3*10^4$ случайных чисел на каждом шаге по времени. Если генерировать эти числа заранее, то для 100–1 000 шагов по времени в GPU придется хранить $3*10^6 - 3*10^7$ чисел, что занимает 12–120 MB памяти. Такой объем требуемой памяти существенен для графических процессоров с ограниченной памятью, таких как в серии GeForce (NVIDIA) с объемом памяти ≈ 1 ГБ. Таким образом, PRNG должен быть включен в главное вычислительное ядро при моделировании N тел на GPU. Наибольшей производительности PRNG на GPU можно достичь, максимизируя количество вычислений на GPU при уменьшении числа обращений к глобальной памяти GPU (операции ввода/вывода), которая должна быть доступна через текстурные ссылки или в режиме, допускающем одновременное обращение к ячейке памяти. В дополнение, возможности GPU будут полностью использованы, если общее число параллельных потоков

будет в ~ 10 раз больше числа вычислительных ядер в устройстве GPU, поскольку в этом случае ни одному из ядер не придется ждать окончания вычислений на других.

Чтобы получить параллельные реализации некоторых PRNG, мы используем круговое разделение [L'Ecuyer, 1996]. Основная идея заключается в распределении одной последовательности PRNG, которая может быть рассмотрена как периодический цикл случайных чисел между множеством потоков, одновременно выполняющихся на GPU, каждый из которых производит поток случайных чисел. Поскольку большинство PRNG основаны на последовательной трансформации текущего состояния с помощью генераторов LCG, Ran 2 и Hybrid Taus, наиболее естественным способом разделения PRNG последовательности представляется сообщение потокам различных начальных значений (random seeds). При этом последовательность распределяется между потоками таким образом, чтобы исключить внутрипоточные корреляции. Это — основа one-PRHG-per-thread-подхода, описанного ниже (рис. 1).

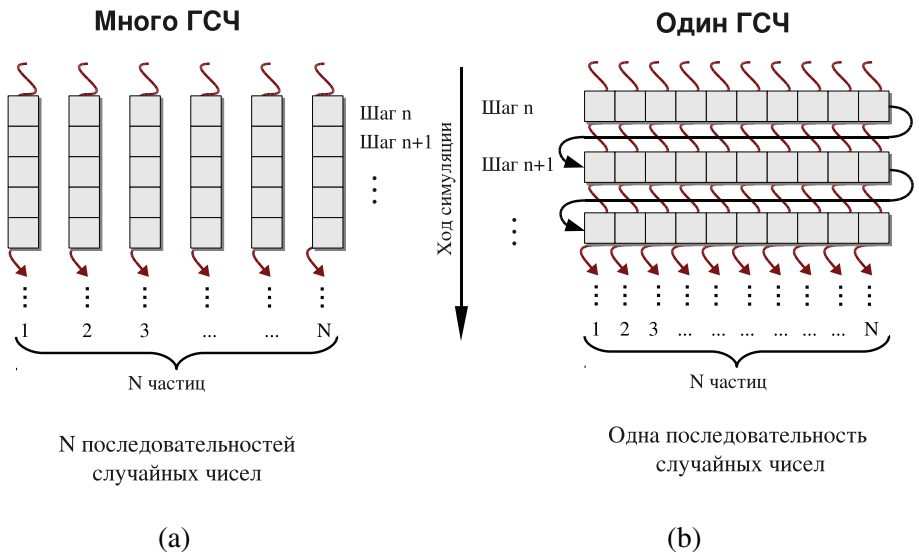


Рис. 1. Блок-схема one-PRNG-per-thread-подхода (a) и one-PRNG-for-all-threads-подхода (b). В подходе one-PRNG-per-thread N независимых PRNG (для N частиц) работают одновременно на N потоках GPU-устройства. Таким образом генерируются псевдослучайные числа из одной последовательности, но начинающиеся с разных наборов начальных состояний. В one-PRNG-for-all-threads-подходе единственный PRNG используется для N потоков, работающих параллельно на GPU и разделяющих один набор начальных состояний. Таким образом получают N подпоследовательностей одной последовательности псевдослучайных чисел. Последовательность вычислений для молекулярного моделирования указана стрелками. n, $n + 1$, … — шаги вычислений

Существуют PRNG, такие как твистер Менсенна и алгоритм Фибоначчи с запаздыванием, в которых можно перескочить по последовательности и вычислить $(n+1)$-е случайное число, не вычисляя перед этим n-го [Tausworthe, 1965; Matsumoto, Nishimura, 1998]. Длина скачка, которая обычно зависит от параметров PRNG, может быть скоррелирована с числом потоков (которое пропорционально числу частиц). Тогда все N чисел могут быть получены одновременно, т. е. j-я нить вычисляет j-е, $(j+N)$-е, $(j+2N)$-е и т. д. числа. Следует отметить, что в конце каждого шага по времени все потоки должны быть синхронизированы, чтобы обновить текущее состояние PRNG. Следует отметить, одно состояние PRNG используется для всех потоков, каждый из которых обновляет только один элемент этого состояния. Мы будем называть такой подход one-PRNG-per-all-threads (рис. 1). Ниже оба подхода будут рассмотрены более подробно.

One-PRNG-per-thread подход

Идея заключается в использовании одного алгоритма PRNG на нескольких потоках, которые генерируют подпоследовательности одной последовательности случайных чисел, используя один и тот же алгоритм, но с разными начальными значениями.

Сначала CPU генерирует N наборов начальных значений и отсылает их в глобальную память GPU (рис. 2). Чтобы избежать корреляций, эти последовательности должны быть взяты из независимой последовательности случайных чисел или получены с помощью других PRNG алгоритмов на CPU. Во время выполнения каждый поток считывает свой набор из глобальной памяти GPU и копирует его в локальную (для потока) или разделяемую (для блока потоков) память GPU. Тогда PRNG на каждом потоке может генерировать столько случайных чисел, сколько требуется, не используя медленную глобальную память GPU. В конце каждого шага по времени каждый PRNG сохраняет свое текущее значение в глобальную память и освобождает локальную. Следует отметить, что, поскольку каждый поток имеет свой собственный PRNG, нет нужды синхронизировать потоки при моделировании независимых частиц. Но при моделировании взаимодействующих частиц нити должны быть синхронизированы.

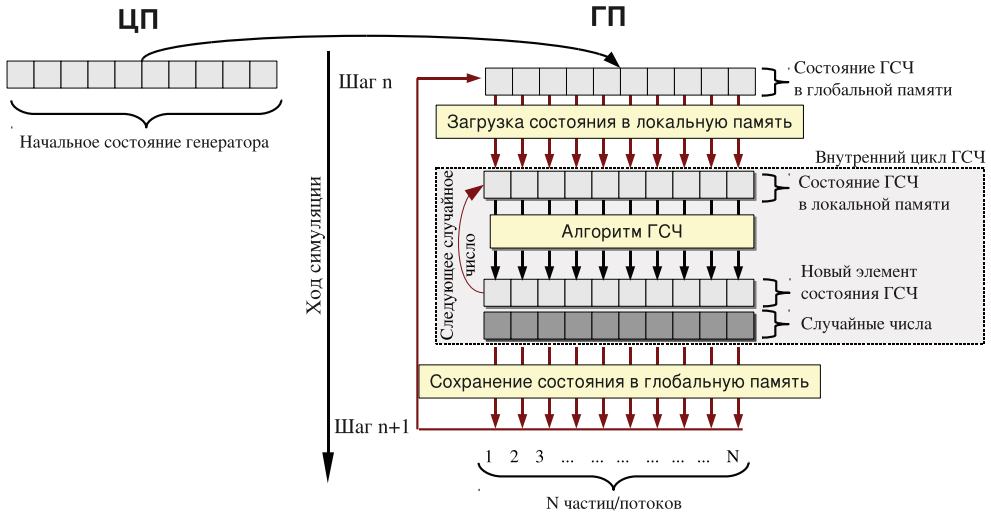


Рис. 2. Схематичное представление реализации one-PRNG-per-thread подхода на GPU. Стрелки указывают последовательность вычислений и перемещение данных. Перед тем как запустить PRNG на GPU, на CPU генерируют (и копируют в глобальную память GPU) N наборов начальных состояний (один набор для каждой частицы). Чтобы получить случайные величины, GPU читает соответствующий набор из глобальной памяти и генерирует столько случайных чисел, сколько требуется на текущем шаге вычислений. Когда все случайные числа получены, каждый поток сохраняет свое текущее состояние PRNG в глобальной памяти GPU, так что это состояние может быть использовано на следующем шаге

При молекулярном моделировании N частиц на каждом шаге моделирования требуется $4N$ равномерно распределенных случайных величин, а массивы начальных значений и текущих состояний должны быть доступны для чтения в режиме, допускающем одновременное обращение к ячейке памяти, чтобы ускорить работу с глобальной памятью. Сами PRNG должны требовать очень мало памяти, поскольку обращения каждого отдельного потока к большим объемам данных могут сильно тормозить работу GPU. Кроме того, малый размер встроенной (on-chip) памяти может быть недостаточным, чтобы хранить текущее состояние PRNG на основе сложного алгоритма. Такие ограничения затрудняют использование изощренных алгоритмов, особенно основанных на статистических свойствах случайных чисел.

В подходе one-PRNG-per-thread количество памяти, требуемой для хранения текущего состояния генератора, пропорционально числу потоков (числу частиц). Следовательно, чтобы выполнить расчет большой системы, для каждого PRNG должен быть выделен значительный объем памяти. Например, LCG-генератор требует для хранения текущего состояния одно целое значение, т. е. 4 байта для одной нити или ~ 4 MB для 10^6 частиц (нитей). Алгоритм Hybrid Taus использует уже четыре целых значения, т. е. ~ 16 MB для 10^6 частиц. Это приемлемые величины при объеме памяти GPU в сотни мегабайт. Для сравнения, Ran2 требует 35 длинных целых величин, т. е. 280 байт для нити или ~ 280 MB для 10^6 нитей. Это приводит к некоторым ограничениям. Во-первых, не все начальные значения могут храниться во встроенной (локальной или разделяемой) памяти (~ 16 MB), и PRNG приходится обращаться к глобальной памяти GPU, чтобы обновить или прочитать текущее состояние. Во-вторых, меньше памяти становится доступно другим процедурам. Это может препятствовать использованию алгоритма Ran2 при моделированию больших систем на некоторых графических процессорах, таких как GeForce GTX280, GTX295 (NVIDIA), с объемом глобальной памяти 768 MB. Однако для таких процессоров, как Tesla C1060, с объемом глобальной памяти 4 Гб, подобных проблем не стоит.

Мы использовали подход one-PRNG-per-thread для реализации алгоритмов Hybrid Taus и Ran2 на GPU. Псевдокод представлен в приложении А.

One-PRNG-for-all-threads подход

Здесь разработчик может использовать единственный PRNG, предоставив доступ к текущему состоянию генератора для всех потоков. Этот метод подходит для PRNG, основанных на следующих преобразованиях:

$$x_n = f(y_{n-r}, y_{n-r+1}, ..., y_{n-k}),$$

где r и $k > r$ — степень рекурсии и постоянный параметр соответственно. Такое преобразование позволяет получить случайное число на n-м шаге с помощью переменных состояния, полученных на предыдущих шагах $n - r$, $n - r + 1$, ... $n - k$.

Если последовательность случайных чисел должна быть одновременно получена с использованием N потоков, каждый из которых генерирует одно случайное значение на каждом шаге, то все N случайных чисел будут получены за один шаг. Кроме того, при условии $k > N$ во время вычислений все элементы для преобразования уже получены на предыдущих шагах и доступны без синхронизации потоков.

Одним из алгоритмов, который может быть выполнен на GPU с использованием one-PRNG-for-all-threads, является алгоритм Фибоначчи с запаздыванием [Mascagni, Srinivasan, 2004] (рис. 3). Предположим, что каждый поток вычисляет одно случайное значение и $sl > N$, $ll - sl > N$. Тогда N случайных чисел могут быть одновременно получены N потоками на GPU без синхронизации.

Чтобы инициализировать алгоритм Фибоначчи с запаздыванием, на CPU с помощью начальных значений должны быть созданы ll целых величин. Каждый поток читает два целых значения из этой последовательности, соответствующие длинному запаздыванию ll и короткому запаздыванию sl, генерирует результат, который соответствует длинному запаздыванию, и сохраняет его в глобальной памяти GPU. Условия $sl > N$, $ll - sl > N$ гарантируют, что ни один элемент массива целых (переменных текущего состояния) не будет использован двумя потоками одновременно. В таком случае только что полученная случайная величина может быть записана в этот массив, который соответствует длинному запаздыванию.

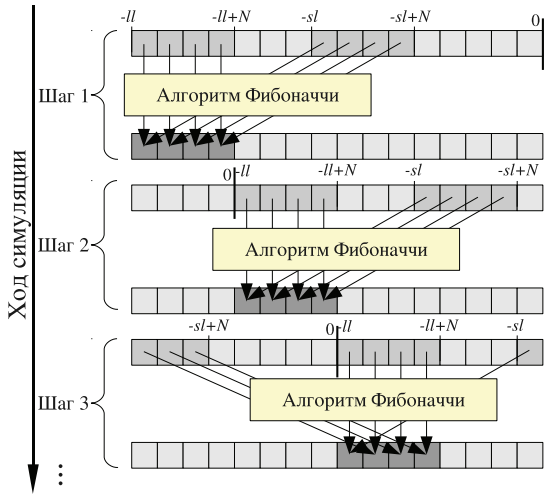


Рис. 3. Схематичное представление реализации подхода one-PRNG-for-all-threads на GPU и параллельное выполнение алгоритма Фибоначчи с запаздыванием, используя принцип разделения «круг». Состояние генератора представлено в виде набора *ll* целых значений. Начальные значения остояния получены на CPU и скопированы в глобальную память GPU. Генерация *N* случайных чисел на каждом шаге происходит одновременно на *N* потоках, используя (2) (показано стрелками). Полученные случайные значения сохраняются, чтобы обновить состояние PRNG для будущего использования. По ходу вычислений сетка потоков движется вдоль последовательности случайных чисел, каждый раз переписывая *N* переменных состояния, которые появляются в последовательности на *ll* позиций раньше. Эта процедура повторяется многократно. Темные серые квадраты на рисунке соответствуют переменным состояния, которые использует PRNG, и обновленной на данном шаге части состояния PRNG. Черная нулевая линия соответствует текущему положению первого потока. Эта линия сдвигается на *N* позиций вперед на каждом шаге

Движущееся окно *N* случайных величин, обновляемое *N* потоками на каждом шаге, вращается вокруг массива переменных состояния, перескакивая на каждом шаге на *N* позиций вперед. Важно, что период алгоритма Фибоначчи $p \approx 2^{ll+31}$ должен удовлетворять $p >> N * S$, где *S* — число вычислительных шагов. Этого можно добиться, увеличивая *ll* и *sl*. Выбор *ll* и *sl* не влияет на время выполнения, но изменяет размер массива переменных состояния, поскольку число целых значений, хранящихся в глобальной памяти GPU, пропорционально *ll*. Впрочем, при $ll \approx 10^6$ требуется только 4 Мб памяти на GPU. Но заметим, что для реализации алгоритма Фибоначчи на GPU с помощью подхода one-PRNG-for-all-threads требуется хранить *N* независимых состояний PRNG размера *ll*, т. е. потребуется в *N* раз больше памяти. Псевдокод приведен в приложении B.

Тест на случайность: процесс Орнштайна–Уленбека

Чтобы исследовать статистическую и вычислительную производительность LCG, Ran2, Hybrid Taus и алгоритма Фибоначчи с запаздыванием, мы провели (целиком на GPU) ЛД-моделирование *N* независимых одномерных гармонических осцилляторов в стохастическом термостате. Мы использовали аналитически решаемую в статистической физике модель, чтобы можно было сравнить результаты моделирования с теоретическими, полученными на действительно случайных числах. Для тестового моделирования мы использовали видеопроцессор NVIDIA GeForce GTX295 с двумя процессорными блоками (GPU), каждый из которых содержит 30 мультипроцессоров (т. е. всего 240 ALU) [NVIDIA CUDA Programming Guide, 2009], и с глобальной памятью 768 Мб.

Итак, рассмотрим систему, в которой каждая частица создает гармонический потенциал $VR_i = \frac{k_{sp}R_i^2}{2}$, где k_{sp} — коэффициент упругости. Уравнения движения Ланжевена в сильно демпфированном пределе,

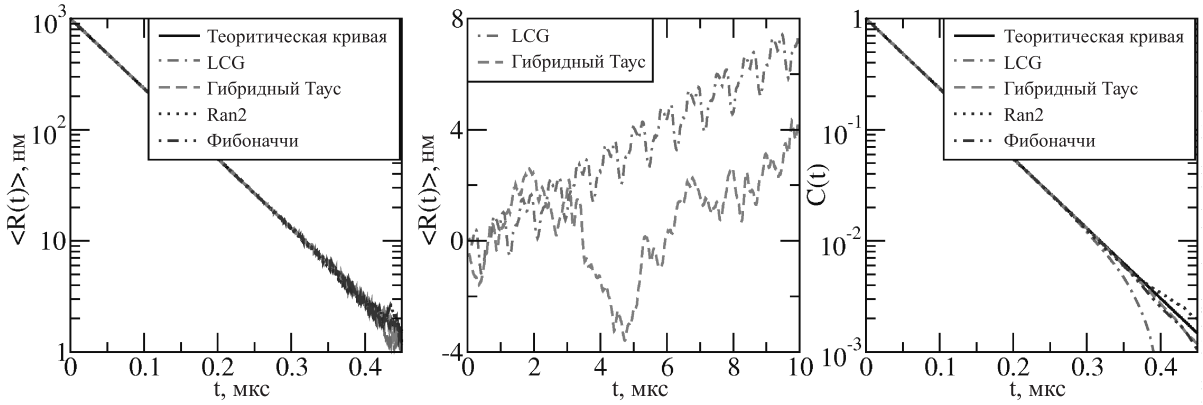
$$\xi \frac{dR_i}{dt} = -\frac{\delta V(R_1, R_2, ..., R_N)}{\delta R_i} + G_i(t), \tag{4}$$

были численно проинтегрированы с помощью интеграционной схемы первого порядка (алгоритм Ермака–МкКаммона) [Matsumoto, Nishimura, 1998]:

$$R_i(t + \Delta t) = R_i(t) + fracf(R_i(t)))\Delta t\xi + g_i(t)\sqrt{2k_{BT}\xi\Delta t}\,, \tag{5}$$

где $f(t) = -\frac{\delta V(R_1, R_2, ..., R_N)}{\delta R_i}$ — это сила, действующая на *i*-ю частицу, а ξ — коэффициент трения, T — температура [Ermak, McCammon, 1978; Hummer, Szabo, 2003; Barsegov et al., 2006]. В уравнениях (4–5) $G_i(t) = g_i(t)\sqrt{2k_{BT}\xi\Delta t}$ — это случайные силы, а g_i — случайные значения, имеющие гауссовское распределение с нулевым средним и единичной дисперсией, полученные с помощью LCG, Ran2, Hybrid Taus, алгоритма Фибоначчи с запаздыванием. Такая модель широко используется в ЛД-моделировании биомолекул. Значения постоянных параметров для LCG, Ran2, Hybrid Taus, алгоритма Фибоначчи можно найти соответственно в [Tsang, Marsaglia, 2000, раздел II], [Press et al., 1992], приложении А и табл. 2.

Мы использовали метод one-PRNG-per-thread для разработки LCG, Ran2 и Hybrid Taus алгоритмов на GPU. Для алгоритма Фибоначчи использовался метод one-PRNG-for-all-threads. Созданные PRNG были использованы в программе ЛД-моделирования, написанной на CUDA. Численные алгоритмы ЛД-моделирования на GPU биомолекул, т. е. преобразования потенциальной энергии, вычисление сил и интегрирование уравнений Ланжевена, представлены в [Zhmurov et al., 2010]. В нашей реализации каждый поток рассчитывает траекторию своей частицы. Мы используем по 64 потока в каждом блоке. Численный расчет для $N = 10^4$ частиц был выполнен с шагом по времени $\Delta t = 1$ пс, начиная от равновесного положения $R_0 = 10$ нм, при значениях параметров $k_{sp} = 0.01pN/nmT = 300KD = 0.25nm^2/ns$. Поскольку GPU может выполнять миллионы операций в секунду, мы использовали мягкий коэффициент упругости ($0.01pN/nm$), чтобы получить длинные траектории за 1 мс (10^9 шагов по времени). Мы анализировали среднее местоположение частиц $\langle R(t)\rangle$ и корреляционную функцию $\langle R(t)R(0)\rangle$, полученные при моделировании с использованием LCG, Ran2 и Hybrid Taus и алгоритма Фибоначчи, и сравнивали их с точными значениями [Doi, Edwards, 1988; Risken, 1989] — $\langle R(t)\rangle = R_i(0)exp[-t/\tau]$ и $\langle R(t)R(0)\rangle = (k_BT/k_{sp})exp[-t/\tau]$, где $\tau = \xi/k_{sp}$. Полученные результаты показывают, что все PRNG, кроме LCG, хорошо моделируют броуновское движение (см. рис. 4). И $\langle R(t)\rangle$, и $\langle R(t)R(0)\rangle$, полученные с помощью Ran2, и Hybrid Taus, и алгоритма Фибоначчи, практически полностью совпадают с теоретическими значениями. LCG, напротив, приводит к повторяющемуся шуму в $\langle R(t)\rangle$ и нефизической корреляции в $\langle R(t)R(0)\rangle$. Во всех случаях на больших временах численные значения $\langle R(t)\rangle$ и $\langle R(t)R(0)\rangle$ слегка отклоняются от теоретических кривых из-за мягкой гармонической упругости и незначительной дискретизации.



(a) (b) (c)

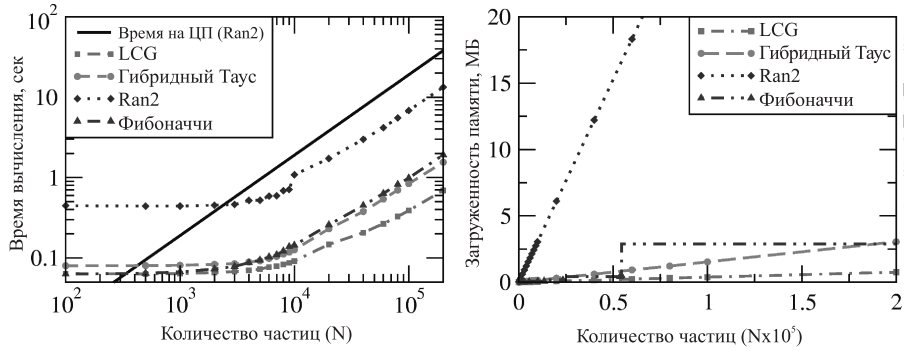
Рис. 4. Полулогарифмические графики среднего по ансамблю положения частиц $\langle X(t)\rangle$ (a) и (b) и двухчастичная корреляционная функция $C(t)=\langle X(t)X(0)\rangle$ (c) для системы из N одномерных гармонических осцилляторов в стохастическом термостате. Теоретические кривые $\langle X(t)\rangle$ и $C(t)$ сравниваются с результатами ланжевеновского моделирования с использованием LCG, Hybrid Taus, Ran 2 и Lagged Fibonacci алгоритмов. Равновесные флуктуации $\langle X(t)\rangle$ выделены на рисунке (b), где можно видеть повторяющиеся изменения $\langle X(t)\rangle$ на больших временах, полученные на LCG генераторе. Эти изменения связаны с корреляциями случайных чисел, полученных на разных потоках. Эти корреляции не наблюдаются при использовании Hybrid Taus PRNG

Мы также сравнили различные PRNG по числу обращений к глобальной памяти за один шаг вычислений. В биомолекулярном моделировании на GPU требуется выделять большие объемы памяти — для параметров силового поля, списков Верле, расстояний между частицами и т. д. А размер требуемой памяти растет как квадрат числа частиц. При этом в современных видеопроцессорах объем памяти мал и составляет 2 * 768 Мб в GeForce GTX 295 и 4 Мб в Tesla C1060, а каждое обращение к глобальной памяти требует около 300 тактов. Хотя сами по себе алгоритмы PRNG работают быстро, их скорость в основном определяется числом обращений к глобальной памяти. Так что большое число обращений может значительно замедлить работу PRNG (текущее состояние PRNG должно быть загружено и обновлено каждый раз, когда генерируется случайное число). Число обращений к памяти прямо пропорционально числу генерируемых случайных чисел. LCG, Ran 2, Hybrid Taus и алгоритм Фибоначчи используют соответственно 1, 40, 4 и ~ 3 (зависит от *ll* и *sl*) целых значений состояния для каждой нити. В нашей реализации все PRNG используют 4–16 байт для каждой нити, что вполне приемлемо даже для больших ($N=10^6$) систем. Хотя Ran 2 использует 280 байт для каждого потока, что уже существенно для больших систем (см. табл. 1). Поэтому для хранения и обновления текущего состояния Ran 2 невыгодно использовать локальную или разделяемую память GPU. Ran 2 использует длинные, 64-битные переменные, что удваивает объем данных и требует 4 обращения на запись и 4 — на чтение (для генерации 4 случайных чисел требуется 7 обращений на запись и 7 — на чтение). Hybrid Taus PRNG обращается к глобальной памяти только при инициализации и сохранении текущего состояния. Поскольку он использует 4 переменные состояния, то для каждого потока требуется 4 обращения на запись и 4 — на чтение на каждом шаге (табл. 1). Поскольку в нашей реализации алгоритма Фибоначчи состояния генераторов распределены между потоками, то они хранятся и загружаются из глобальной памяти GPU. Таким образом, поскольку необходимо получить 2 состояния, для генерации одного случайного числа необходимо 2 обращения на чтение и 1 — на запись.

Табл. 1. Оценки используемой памяти (байт/поток) и число обращений к глобальной памяти GPU (число операций ввода/вывода, приходящееся на одну M_1 и четыре (M_2) случайные величины), требуемые для генерации псевдослучайных чисел на GPU на каждом шаге вычислений для LCG, Hybrid Taus, Ran 2, и Lagged Fibonacci алгоритмов. В молекулярном моделировании требуется 4 псевдослучайных величины для каждой частицы, чтобы получить три компоненты (x, y z) действующей на них случайной силы

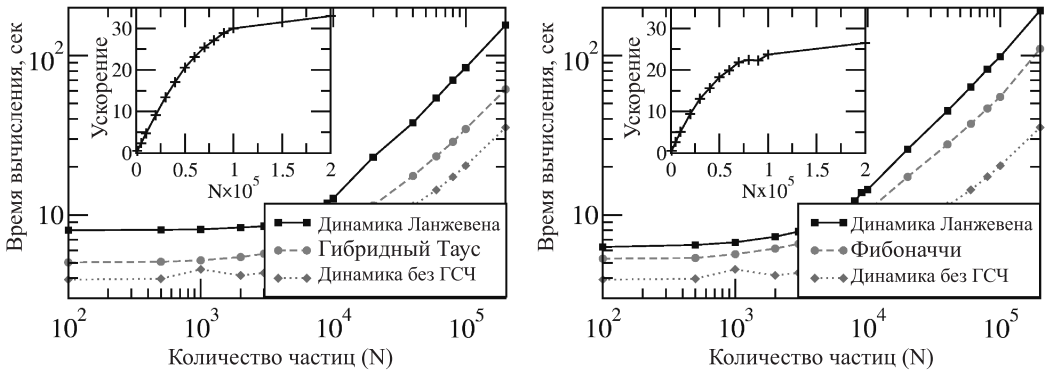
Параметр	LCG	Hybrid Taus	Ran 2	Lagged Fibonacci
байт/нить	4	16	280	12
M1	1/1	4/4	4/4	3/1
M2	1/1	4/4	7/7	12/4

Чтобы сравнить вычислительную эффективность разных PRNG, мы провели моделирования ланжевеновской системы из N трехмерных осцилляторов в стохастическом термостате при разных значениях N. В реализации на GPU i-я нить расчитывает силу и потенциальную энергию i-й частицы. Для каждого размера системы были смоделированы 10^3 шагов по времени. В конце каждого шага все потоки были синхронизированы. Время выполнения и требуемая память для каждого генератора, в зависимости от N, показаны на рис. 5. Ran 2 требует больше всего и времени, и памяти. Например, использование Ran 2 в ланжевеновском моделировании системы из 10^4 частиц в течение 10^9 шагов по времени требует ~ 264 часа (на видеопроцессорах NVIDIA GeForce GTX 295). Требование по памяти у Ran 2 так же большое — около 250 Мб для 10^6 частиц. К тому же реализация Ran 2 на GPU не дает большого выигрыша в скорости по сравнению с реализацией на CPU (рис. 5, 6). Таким образом, использование Ran 2 в молекулярном моделировании неэффективно и даже невозможно для больших систем. LCG, Hybrid Taus и алгоритм Фибоначчи имеют хорошую производительность и требуют не слишком много памяти. Но, принимая во внимание статистическое качество каждого генератора, можно сказать, что Hybrid Taus и алгоритм Фибоначчи лучше всего подходят для молекулярного моделирования на GPU. Оба этих генератора требуют < 15–20 MB памяти даже для больших систем (данные не представлены).



(a) (b)

Рис. 5. Вычислительная производительность реализаций на GPU различных алгоритмов — LCG, Ran 2, Hybrid Taus и Lagged Fibonacci — в ланжевеновском моделировании N трехмерных гармонических осцилляторов. (a) Время выполнения 10^3 шагов как функция размера системы N в логарифмическом масштабе. Потоки были синхронизированы на CPU в конце каждого шага моделирования, чтобы сымитировать LD-моделирование биомолекулы (в котором требуется такая синхронизация). Так же показано, для сравнения, время расчета аналогичной модели с использованием Ran 2 на CPU. (b) Показан объем памяти, требуемый для хранения текущего состояния PRNG, как функция N. Ступенчатое увеличение требуемой памяти (для Lagged Fibonacci PRNG) при $N \sim 0.6 * 10^5$ соответствует изменению значений постоянных параметров (см. табл. 2)



(a) (b)

Рис. 6. Время вычисления 10^3 шагов, полученное при сквозном LD-моделировании N трехмерных гармонических осцилляторов, полностью выполненное на GPU с использованием Hybrid Taus (a) и Lagged Fibonacci PRNG (b), как функция от размера системы N. Полное время вычислений по LD-алгоритму (Langevin Dynamics) сравнивается с временем вычисления случайных чисел (временем работы PRNG), а также со временем вычисления динамики аналогичной, но детерминированной системы (без PRNG). Отношение времени, требуемого для вычислений на CPU, ко времени аналогичных вычислений на GPU как функция от N показано на вставках

Выводы и Заключение

Увеличение производительности отдельных ядер CPU становится все более важной задачей для производителей. С увеличением частоты работы CPU увеличивается и их нагрев, что препятствует дальнейшему увеличению производительности. По этой причине современные CPU оснащены несколькими (6–8) ядрами и большой кэш-памятью, а не состоят из единственного и быстрого ядра. Таким образом, графические процессоры получили свое развитие как альтернативный тип расчетных устройств, основанных на увеличении числа процессоров, а не их частоты. Параллельная архитектура GPU дает разработчику альтернативную расчетную платформу со множеством ALU на одном графическом процессоре. Хотя это и приводит к уменьшению кэш-памяти и ограничениям в управлении потоками. Таким образом, чтобы использовать вычислительную мощность GPU в конкретном приложении, приходится переделывать программы, которые уже давно используются на CPU. В контексте молекулярной динамики необходимо разделить всю задачу (генерация случайных чисел, вычисление положений частиц и сил, действующих на них) на множество независимых подзадач. И, кроме того, чтобы полностью оптимизировать работу GPU, требуется аккуратно организовать работу с памятью, синхронизацию и коммуникации между нитями.

В этой статье были рассмотрены one-PRNG-per-thread и one-PRNG-for-all-threads подходы к генерации случайных чисел на GPU (рис. 1), которые мы применили к LCG, Ran 2, Hybrid Taus и алгоритму Фибоначчи с запаздыванием. Эти PRNG были исследованы с помощью ланжевеновской модели N независимых броуновских частиц, создающих гармонический потенциал, описанный в программе на CUDA. LCG, Hybrid Taus и Ran 2 были реализованы на GPU как независимые PRNG, параллельно работающие на всех нитях сразу и получающие одновременно множество разных потоков случайных величин из одной начальной последовательности (one-PRNG-per-thread, рис. 2). Алгоритм Фибоначчи реализован с помощью подхода one-PRNG-for-all-threads, когда все нити генерируют один поток случайных чисел (рис. 3).

Hybrid Taus и алгоритм Фибоначчи с запаздыванием имеют хорошее статистическое качество [Tsang, Marsaglia, 2000; Nguyen, 2008; Brent, 1992] (рис. 4) при той же скорости, что LCG, и небольшом требовании памяти (рис. 5). Кроме того, их длинные периоды позволяют описывать стохастическую динамику очень больших систем ($N > 10^6$) на больших временных

масштабах (число шагов по времени $>10^9$). Все это делает Hybrid Taus и алгоритм Фибоначчи очень удобными для LD-моделирования биомолекул [Barsegov et al., 2006; Mickler et al., 2007; Clementi, 2008] и MD-моделирования в неявных растворителях [Brooks et al., 1983; Haberthür, Caflisch, 2008].

Ran 2 является хорошо исследованным генератором, дающим случайные числа высокого статистического качества (рис. 4) [Press et al., 1992]. Он мог бы быть лучшим генератором для молекулярного моделирования, но работает как минимум в 10 раз медленнее и требует слишком много памяти (рис. 5). Но в молекулярном моделировании описание дальнодействующих и электростатических взаимодействий вызывает больше всего затруднений, так что Ran 2 можно использовать для малых систем ($N \le 10^3$). Использование Ran2 в молекулярном моделировании больших систем на GPU может сильно понизить скорость вычислений, так что в этом случае следует использовать Hybrid Taus и алгоритм Фибоначчи с запаздыванием.

Статистические характеристики случайных чисел, полученных подходом one-PRNG-per-thread, не улучшаются с увеличением системы. В этом подходе, когда каждой нити соответствует свой генератор со своими состоянием, увеличение числа частиц (потоков) увеличивает число генераторов, но не улучшает статистических качеств генерируемых чисел. Напротив, в подходе one-PRNG-for-all-threads с увеличением числа потоков увеличивается число переменных состояния генератора и, значит, увеличивается статистическое качество чисел. Следует отметить, что это — главное свойство генераторов, реализованных с подходом one-PRNG-for-all-threads [Tsang, Marsaglia, 2000; Brent, 1992]. По этой причине мы рекомендуем алгоритм Фибоначчи с запаздыванием для MD-моделирования неявных растворителей и LD-моделирования биомолекул. Также, поскольку в подходе oner-PRNG-for-all-threads генерируется только одна последовательность случайных чисел, можно непосредственно сравнить результаты моделирования на GPU и CPU. Это может быть использовано в сравнительных тестах для определения ошибок, связанных с первым порядком точности в операциях с плавающей точкой, ошибок округления, а также ошибок обращения к памяти GPU.

Мы исследовали вычислительную производительность алгоритма Фибоначчи и Hybrid Taus в зависимости от размера моделируемой системы. Наши исследования показывают, что время работы обоих генераторов пропорционально числу моделируемых частиц в случае $N < 5 \cdot 10^3$, в связи с малой параллелизацией, но время начинает расти значительно быстрее при больших значениях N, когда все ALU на GPU загружены (рис. 5).

Качественное сравнение времени работы PRNG и расчета детерминированной динамики системы (т. е. без случайных гауссовых сил) показывает, что время расчета PRNG немного превышает время расчета собственно системы. Это отражает тот факт, что в использованной нами модели не применяются дальнодействующие потенциалы.

В дополнение нужно отметить, что использование Hybrid Taus и алгоритма Фибоначчи на GPU уменьшает время вычислений в 25–35 раз по сравнению с временем решения аналогичной задачи на CPU. И хотя использование Hybrid Taus дает больший выигрыш во времени, чем алгоритм Фибоначчи, использовать последний было бы предпочтительнее, поскольку он допускает реализацию на новейших видеопроцессорах, реализующих архитектуру Multiple Instruction Multiple Data (MIMD) [NVIDIA's Next generation CUDA Compute Architecture: Fermi, 2009].

Мы также использовали строгие тесты на случайность, чтобы исследовать статистические свойства чисел, генерируемых PRNG на основе алгоритма Фибоначчи с запаздыванием. Оказалось, что этот генератор с коротким запаздыванием $sl = 1252$ проходит тесты DIEHARD [Marsaglia, 1996] и набор тестов BigCrash из TestUO1 package [Tsang, Marsaglia, 2000].

В заключение отметим, что развитие архитектуры Fermi (NVIDIA) и Larrabee (Intel), каждая из которых содержит 512 ALU, является крайне важным шагом в развитии GPU-вычислений. Это новое поколение процессоров будет поддерживать MIMD протокол,

который позволит разработчику использовать множество ALU в независимых вычислениях так, чтобы разные ядра могли одновременно выполнять разные вычислительные процедуры на разных блоках данных. Кроме того, высокая скорость внутренней сети позволит организовать быстрый обмен сообщениями между потоками. Эти достижения в архитектуре компьютеров позволят программисту более эффективно распределить задачи между ядрами GPU. В контексте MD-моделирования неявных растворителей и LD-моделирования это позволит вычислять псевдослучайные числа и силы, используя меньше синхронизаций потоков внутри одного графического процессора. Это делает подход one-PRNG-for-all-threads, использующий обмен между потоками, еще более важным, поскольку в нем можно достичь дополнительного ускорения за счет быстрых сообщений между потоками. Реализация алгоритма Фибоначчи на GPU на новых графических процессорах может быть достигнута минимальными изменениями уже имеющейся. Подход one-PRNG-for-all-threads может быть также использован для реализации на GPU Твистера Мерсенна [Matsumoto, Nishimura, 1998; Matsumoto, Kurita, 1994; Zhmurov et al., 2010], а так же нескольких других генераторов, таких как мультирекурсивный (MRG) или linear/generalized shift feedback register (LSFR/GSFR), например 4-lag алгоритм Фибоначчи с запаздыванием [Tsang, Marsaglia, 2000; Marsaglia, 1999].

Приложения

A. Подход one-PRNG-per-thread: Hybrid Taus и Ran 2 алгоритмы

Приведенный псевдокод описывает реализацию Hybrid Taus и Ran 2 PRNG на GPU. Здесь верхний индекс h означает принадлежность к памяти CPU, а индекс d означает, что данные хранятся в глобальной памяти GPU. Отметим, что код как для GPU, так и для CPU, описаны в одном листинге. В приложениях PRNG алгоритмов на CUDA код для GPU реализован на различных ядрах.

Алгоритм Hybrid Taus — комбинация генераторов Tausworthe taus 88 и LCG — может быть использован в молекулярном моделировании на GPU с помощью следующего псевдокода.

Алгоритм 1: Hybrid Taus алгоритм.

Требуется: выделенные в памяти CPU массивы $y_1^h[N]$, $y_2^h[N]$, $y_3^h[N]$ и $y_4^h[N]$.

Требуется: выделенные в памяти GPU массивы $y_1^d[N]$, $y_2^d[N]$, $y_3^d[N]$ и $y_4^d[N]$.

1. $y_1^h[1\ldots N]$ to $y_4^h[1\ldots N] \leftarrow$ инициализация начального состояния.
2. $y_1^h[1\ldots N]$ to $y_4^h[1\ldots N] \rightarrow y_1^d[1\ldots N]$ to $y_4^d[1\ldots N]$ копирование состояний на GPU.

[Начало секции GPU кода]

3. $j_{th} \leftarrow$ индекс потока
4. y_1, y_2, y_3 и $y_4 \leftarrow y_1^d[j_{th}]$, $y_2^d[j_{th}]$, $y_3^d[j_{th}]$ и $y_4^d[j_{th}]$ {загрузка состояний генераторов}
5. **for** $i = 1$ to 4; $i{+}{+}$ **do** {генерация 4 случайных чисел}
6. $b \leftarrow (((y_1 \ll c_{11}) \text{ XOR } y_1) \gg c_{21})$
7. $y_1 \leftarrow (((y_1 \text{ AND } c_1) \ll c_{31}) \text{ XOR } b$
8. $b \leftarrow (((y_2 \ll c_{12}) \text{ XOR } y_2) \gg c_{22})$
9. $y_2 \leftarrow (((y_2 \text{ AND } c_2) \ll c_{32}) \text{ XOR } b$
10. $b \leftarrow (((y_3 \ll c_{13}) \text{ XOR } y_3) \gg c_{23})$
11. $y_3 \leftarrow (((y_3 \text{ AND } c_3) \ll c_{33}) \text{ XOR } b$
12. $y_4 \leftarrow ay_4 + c$
13. Output $mult \times (\text{ XOR } y_1 \text{ XOR } y_2 \text{ XOR } y_3 \text{ XOR } y_4)$
14. **endfor**

15. y_1, y_2, y_3 and $y_4 \to y_1^d[j_{th}]$, $y_2^d[j_{th}]$, $y_3^d[j_{th}]$ and $y_4^d[j_{th}]$ {сохранение текущего состояния генератора}

[Конец секции GPU кода]

В приведенном листинге:

b — временная целочисленная беззнаковая переменная,

y_1, y_2, y_3, y_4 — беззнаковые целые числа для хранения состояний трех генераторов Tausworthe (строки 6-11) и одного LCG (строка 12),

XOR — бинарное исключающее ИЛИ,

``$\gg$'' и ``$\ll$'' — бинарный сдвиг направо и налево соответственно,

$mult = 2.3283064365387 \times 10^{-10}$ — мультипликатор, который преобразует целое число в число с плавающей точкой от 0 до 1,

$c_{11} = 13$, $c_{21} = 19$, $c_{31} = 12$, $c_{21} = 2$, $c_{22} = 25$, $c_{23} = 4$, $c_{31} = 3$, $c_{32} = 11$, $c_{33} = 17$, $c_1 = 4294967294$, $c_2 = 4294967288$, $c_3 = 4294967280$ — постоянные параметры Tausworthe генераторов,

$a = 1664525$ и $c = 1013904223$ постоянные параметры LCG.

Алгоритм $\mathrm{Ran}2$ может быть использован в молекулярном моделировании на GPU с помощью следующего псевдокода.

Алгоритм 2: $\mathrm{Ran}2$ алгоритм.

Требуется: выделенные в памяти CPU массивы $idum^h[N]$, $idum2^h[N]$, $iy^h[N]$ и $iv^h[N * NTAB]$.

Требуется: выделенные в памяти GPU массивы $idum^d[N]$, $idum2^d[N]$, $iy^d[N]$ и $iv^d[N * NTAB]$.

1. $idum^h[1 \ldots N] \leftarrow$ инициализация начального состояния
2. **for** $i = 0$ to $N - 1$; $i{+}{+}$ **do** {загрузка всех N генераторов}
3. $idum2^h[i] \leftarrow idum^h[i]$
4. **for** $j = NTAB + 7$ to 0; $j{-}{-}$ **do**
5. $k \leftarrow idum^h[i] / IQ1$
6. $idum^h[i] \leftarrow IA * (idum^h[i] - k * IQ1) - k * IR1$
7. **if** $idum^h[i] < 0$ **then**
8. $idum^h[i] = idum^h[i] + IM1$
9. **endif**
10. **if** $j < NTAB$ **then**
11. $iv^h[i * NTAB + j] = idum^h[i]$
12. **endif**
13. **endfor**
14. **endfor** {все N генераторов инициализированы}
15. $idum^h \to idum^d$; $idum2^h \to idum2^d$; $iy^h \to iy^d$; $iv^h \to iv^d$ {копирование на GPU}
16. **for** $t = 0$ to S; $t{+}{+}$ **do** {начало симуляции длинной в S шагов}

[Начало секции GPU кода]

17. $j_{th} \leftarrow$ индекс потока
18. $idum \leftarrow idum^d[j_{th}]$; $idum2 \leftarrow idum2^d[j_{th}]$; $iy \leftarrow iy^d[j_{th}]$ {копирование в локальную память GPU}
19. $x[4]$ {вектор для вывода случайных чисел}
20. **for** $i = 0$ to 4; $i{+}{+}$ **do** {генерация четырех случайных чисел}

21. $k \leftarrow idum / IQ1;\ idum \leftarrow IA1 * (idum - k * IQ1) - k * IR1$
22. **if** $idum < 0$ **then**
23. $idum = idum + IM1$
24. **endif**
25. $k \leftarrow idum2 / IQ2;\ idum2 \leftarrow IA2 * (idum2 - k * iQ2) - k * IR2$
26. **if** $idum2 < 0$ **then**
27. $idum2 = idum2 + IM2$
28. **endif**
29. $j \leftarrow iy / NDIV$
30. $iv \leftarrow iv^d[j_{th} * NTAB + j]$ {часть состояния в глобальной памяти GPU}
31. $iy = iv - idum2;\ idum \rightarrow iv^d[j_{th} * NTAB + j]$
32. **if** $iy < 1$ **then**
33. $iy \leftarrow iy + IMM1$
34. **endif**
35. $tempran \leftarrow AM * iy$
36. **if** $tempran > RNMX$ **then**
37. $x[i] \leftarrow RNMX$
38. **else**
39. $x[i] \leftarrow tempran$
40. **endif**
41. **endfor**
42. $idum \rightarrow idum^d[j_{th}];\ \ idum2 \rightarrow idum2^d[j_{th}];\ \ iy \rightarrow iy^d[j_{th}]$ {сохранениее в глобальную память GPU}
43. Output: x

[Конец секции GPU кода]

44. Следующий шаг симуляции

Как только N генераторов инициализированы на CPU (строки 1–14), начальные значения для всех генераторов копируются в глобальную память GPU (строка 15). Вычисления на GPU начинаются с 17 строки. Каждый поток на GPU хранит значения состояния генератора в глобальной памяти GPU, используя индекс потока, а также копирует значения переменных $idum$, $idum2$ и iy в локальную память GPU (строка 18). Поскольку массив iv слишком большой, чтобы хранить его в локальной памяти, доступ к нему организован с помощью непосредственных обращений к глобальной памяти GPU (строки 30 и 31). Каждый поток генерирует четыре случайных величины (цикл, начинающийся на строке 20) и сохраняет их в массиве $x[4]$. После этого переменные текущего состояния PRNG обновляются в глобальной памяти GPU (строка 42).

B. Подход one-PRNG-for-all-threads: Lagged Fibonacci алгоритм

Реализация Lagged Fibonacci алгоритма на GPU приведена в следующем псевдокоде.

Алгоритм 3: аддитивный Lagged Fibonacci алгоритм.

Требуется: выделенный на GPU массив $x^d[N]$.

1. $x^d[1 \ldots ll] \leftarrow$ инициалицация начального состояния
2. **For** $t = 0$ to S **do** {начало симуляции}

[Начало секции GPU кода]

3. $j_{th} \leftarrow$ индекс потока
4. $shift_0 \leftarrow (j_{th} + N * t) * RNS$

5. **for** $shift = shift_0$ to $shift_0 + RNS - 1$ **do**
6. $x_{ll} \leftarrow x^d[shift \bmod ll]$
7. $x_{sl} \leftarrow x^d[(shift + sl - ll) \bmod ll]$
8. $x \leftarrow (x_{ll}\ op\ x_{sl}) \bmod m$
9. Output x
10. $x \rightarrow x^d[shift \bmod ll]$
11. **endfor**

[Конец секции GPU кода]

12. **endfor**

Чтобы инициализировать генератор, CPU помещает ll целых, описывающих начальное состояние в массиве x^d и копирует их на GPU (строка 1). На GPU каждый поток вычисляет положение ($shift0$) целого числа, которое соответствует положению первой случайной величины, которую нужно сгенерировать. Это проделывается при использовании текущего шага вычислений (t), номера потока (j_{th}), полного числа потоков (N) и числа случайных величин, требующихся на каждом шаге (RNS). Строки 6–10 повторяются до тех пор, пока не будет сгенерировано RNS случайных чисел (цикл начинается со строки 5), используя оператор op на строке 8. Для вычисления каждого случайного числа необходимо считать два целых из состояния PRNG (строки 6 и 7). Эти целые соответствуют параметрам генератора ll (long lag) и sl (short lag). Индексация в массиве целых всегда выполняется по модулю ll, что соответствует циклическому обходу (по достижении конца массива обход начинается сначала). Полученное целое x сохраняется для следующих шагов (строка 10). Если требуется RNS случайных чисел в каждом потоке и на каждом шаге, то должно быть выполнено $sl > RNS * N$ и $ll - sl > RNS * N$, поскольку на каждом шаге обновляется $RNS * N$ целых чисел.

Таблица. 2. Постоянные параметры — short lag sl и the long lag ll — для Lagged Fibonacci PRNG в молекулярном моделировании системы размера N

N	<1252	<3004	<5502	<10095	<12470	<23463	<54454	<279695	<288477	<1010202
sl	1 252	3 004	5 502	10 095	12 470	23 463	54 454	279 695	288 477	1 010 202
ll	2 281	4 423	9 689	19 937	23 209	44 497	132 049	756 839	859 433	3 021 377

Список литературы

Anderson A. G., Goddard III W. A., Schröder P. Quantum Monte Carlo on graphical processing units. // Comput. Phys. Commun. — 177 (2007). — Pp. 298–306.

Anderson J. A., Lorentz C. D., Travesset A. General purpose molecular dynamics simulations fully implemented on graphics processing units. // J. Comput. Phys. — 227 (2008). — Pp. 5342–5359.

Barreira L. Poincar´e recurrence: old and new. // XIVth International Congress on Mathematical Physics. World Scientific. — 2006. — Pp. 415–422.

Barsegov V., Klimov D., Thirumalai D. Mapping the energy landscape of biomolecules using single molecule force correlation spectroscopy: Theory and applications. // Biophys. J. — 90 (2006). — Pp. 3827–3841.

Box G. E. P., Miller M. E. A note on the generation of normal random deviates // Ann. Math. Statist. — 29 (1958). — Pp. 610–611.

Brent R. P. Uniform random number generators for supercomputers. // Proc. Fifth Australian Supercomputer Conference. — Pp. 95–104.

Brent R. P., Larvala S., Zimmermann P. A fast algorithm for testing reducibility of trinomials mod 2 and some new primitive trinomials of degree 3021377. // Math. of Comput. — 72 (2003). — Pp. 1443–1452.

Brooks B. R., Bruccoleri R. E., Olafson B. D., States D. J., Swaminathan S., Karplus M. CHARMM: A programm for macromolecular energy, minimization, and dynamics calculations. // J. Comput. Chem. — 4 (1983). — Pp. 187–217.

Clementi C. Coarse-grained models of protein folding: toy models or predictive tools? // Curr. Opin. Struct. Biol. — 18 (2008). — Pp. 10–15.

Davis J. E., Ozsoy A., Patel S., Taufer M. Towards large-scale molecular dynamics simulations on graphics processors. // BICoB'09: Proceedings of the 1st International Conference on Bioinformatics and Computational Biology. Springer-Verlag, Berlin, Heidelberg. — 2009. — Pp. 176–186.

Dima R. I., Joshi H. Probing the origin of tubulin rigidity with molecular simulations. // Proc. Natl. Acad. Sci. USA. — 105 (2008). — Pp. 15743–15748.

Doi M., Edwards S. The Theory of Polymer Dynamics. International Series of Monographs on Physics, Oxford Science Publications. — 1988.

Ermak D. L., McCammon J. A. Brownian dynamics with hydrodynamic interactions. // J. Chem. Phys. — 69 (1978). — Pp. 1352–1360.

Ferrenberg A. M., Landau D. P., Wong Y. J. Monte Carlo simulations: Hidden errors from "good" random number generators. // Phys. Rev. Lett. — 69 (1992). — Pp. 3382–3384.

Friedrichs M. S., Eastman P., Vaidyanathan V., Houston M., Legrand S., Beberg A. L., Ensign D. L., Bruins C. M., Pande V. S. Accelerating molecular dynamic simulation on graphics processing units. // J. Comput. Chem. — 30 (2009). — Pp. 864–872.

Grassberger P. On correlations in "good" random number generators. // Phys. Lett. A 181 (1993). — Pp. 43–46.

Haberthür U., Caflisch A. FACTS: Fast analytical continuum treatment of salvation. // J. Comput. Chem. — 29 (2008). — Pp. 701–715.

Harvey M. J., Fabritilis G. D., An implementation of the smooth Particle Mesh Ewald method on GPU hardware // J. Chem. Theory Comput. — 5 (2009). — Pp. 2371–2377.

Hummer G., Szabo A. Kinetics from nonequilibrium single-molecule pulling experiments. // Biophys. J. — 85 (2003). — Pp. 5–15.

Hyeon C., Dima R. I., Thirumalai D. Pathways and kinetic barriers in mechanical unfolding and refolding of RNA and proteins. // Structure. — 14 (2006). — Pp. 1633–1645.

L'Ecuyer P. Maximally equidistributed combined Tausworthe generators. // Math. Comput. — 65 (1996). — Pp. 203–213.

L'Ecuyer P., Blouin F., Couture R. A search for good multiple recursive random number generators. // ACM Trans. Model. Comput. Simul. — 3 (1993). — Pp. 87–98.

L'Ecuyer P., Simard R. TestU01: A C library for empirical testing of random number generators. // ACM Trans. Math. Softw. — 33 (2007). — P. 22.

Marsaglia G. DIEHARD: A battery of tests of randomness. 1996. `http://stat.fsu.edu/geo/diehard.html`.

Marsaglia G. Random numbers for C: The END?, 1999. Published on sci.crypt.

Marsaglia G., Bray T. A. A convenient method for generating normal variables. // SIAM Rev. — 6 (1964). — Pp. 260–264.

Mascagni M., Srinivasan A. Algorithm 806: SPRNG: A scalable library for pseudorandom number generation. // ACM Trans. Math. Softw. — 26 (2000). — Pp. 436–461.

Mascagni M., Srinivasan A. Parameterizing parallel multiplicative lagged-Fibonacci generators. // Parallel Comput. — 30 (2004). — Pp. 899–916.

Matsumoto M., Kurita Y. Twisted GFSR generators. // ACM Trans. Model. Comput. Simul. — 2 (1992). — Pp. 179–194.

Matsumoto M., Kurita Y. Twisted GFSR generators II. // ACM Trans. Model. Comput. Simul. — 4 (1994). — Pp. 254–266.

Matsumoto M., Nishimura T. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. // ACM Trans. Model. Comput. Simul. — 8 (1998). — Pp. 3–30.

van Meel J. A., Arnold A., Frenkel D., Zwart S. F. P., Belleman R. Harvesting graphics power for MD simulations. // Mol. Simul. — 34 (2008). — Pp. 259–266.

Mickler M., Dima R. I., Dietz H., Hyeon C., Thirumalai D., Rief M. Revealing the bifurcation in the unfolding pathways of GFP by using single-molecule experiments and simulations. // Proc. Natl. Acad. Sci. USA. — 104 (2007). — Pp. 20268–20273.

Nguyen H. (Ed.). GPU Gems 3. Addison-Wesley, 2008.

NVIDIA CUDA C Programming Best Practices Guide, NVIDIA, 2.3 edition, 2009.

NVIDIA CUDA Programming Guide, NVIDIA, 2.3.1 edition, 2009.

NVIDIA's Next generation CUDA Compute Architecture: Fermi, NVIDIA, 1.1 edition, 2009.

Press W. H., Teukolsky S. A., Vetterling W. T., Flannery B. P. Numerical Recipes in C, The Art of Scientific Computing. Cambridge University Press, second edition, 1992.

Risken H. The Fokker-Planck Equation. Springer-Verlag, second edition, 1989.

Selke W., Talapov A. L., Shchur L. N. Cluster-flipping Monte Carlo algorithm and correlations in "good" random number generators. // JETP Lett. — 58 (1993). — Pp. 665–668.

Soto J. Statistical testing of random number generators. 1999. Available at: http://csrc.nist.gov/rng/.

Stone J. E., Phillips J. C., Freddolino P. L., Hardy D. J., Trabuco L. G., Schulten K. Accelerating molecular modeling applications with graphical processors. // J. Comput. Chem. — 28 (2007). — Pp. 2618–2640.

Tausworthe R. C. Random numbers generated by linear recurrence modulo two. // Math. Comput. — 19 (1965). — Pp. 201–209.

Tozzini V. Coarse-grained models for proteins. // Curr. Opin. Struct. Biol. — 15 (2005). — Pp. 144–150.

Tsang W. W., Marsaglia G. The Ziggurat method for generating random variables. // J. Stat. Softw. — 5 (2000).

Veitshans T., Klimov D., Thirumalai D. Protein folding kinetics: timescales, pathways and energy landscapes in terms of sequence-dependent properties. // Fold. Des. — 2 (1997). — Pp. 1–22.

Yang J., Wang Y., Chen Y. GPU accelerated molecular dynamics simulations of thermal conductivities. // J. Comput. Phys. — 221 (2007). — Pp. 799–804.

Zhmurov A., Dima R.I., Kholodov Y.A., Barsegov V. SOP-GPU: Accelerating biomolecular simulations in the centisecond timescale using graphics processors. // Proteins: Structure, Function, and Bioinformatics. 2010. V. 78. I 14. Pp. 2984–2999.