Нижегородский государственный университет им. Н.И. Лобачевского

Национальный исследовательский университет

Программа повышение конкурентоспособности ННГУ им. Н.И. Лобачевского

Стратегическая инициатива 7 «Достижение лидирующих позиций в области суперкомпьютерных технологий и высокопроизводительных вычислений»

Основная образовательная программа
011200 «Физика», общий профиль, квалификация (степень) бакалавр

Основная профессиональная образовательная программа аспирантуры
01.04.07 Физика конденсированного состояния

Учебно-методическая разработка по дисциплине
Параллельное программирование в физических исследованиях

Денисенко М.В., Муняев В.О., Сатанин А.М.

ПРИМЕНЕНИЕ РАСПРЕДЕЛЕННЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ И ТЕХНОЛОГИИ CUDA ДЛЯ МОДЕЛИРОВАНИЯ ФИЗИЧЕСКИХ ПРОЦЕССОВ

Нижний Новгород
2014 год

ПРИМЕНЕНИЕ РАСПРЕДЕЛЕННЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ И ТЕХНОЛОГИИ CUDA ДЛЯ МОДЕЛИРОВАНИЯ ФИЗИЧЕСКИХ ПРОЦЕССОВ

Денисенко М.В., Муняев В.О., Сатанин А.М. Электронное учебно-методическое пособие. – Нижний Новгород: Нижегородский госуниверситет, 2014. – 85 с.

Аннотация. В учебно-методическом пособии рассматриваются возможности применения технологии CUDA в научных исследованиях. Описываются принципы распараллеливания, и демонстрируется пример работающей программы – расчет диссипативной динамики многоуровневой квантовой системы, на примере, квантового нелинейного осциллятора. Эволюция системы рассчитывается на основе уравнения для матрицы плотности. Разработаны и реализованы два алгоритма решения данной задачи: методом квантовых траекторий (квантовым методом Монте-Карло) и с помощью метода некоммутативного интегрирования на группах Ли (разложение по матрицам Гелл-Манна). На базе примера показана эффективность работы графических ускорителей, масштабируемость и описывается возможность взаимодействия нескольких графических ускорителей с применением технологии MPI. Приведено описание и порядок выполнения лабораторных работ по изучению принципов гетерогенных вычислительных систем и реализации численных физических экспериментов. Пособие содержит задачи для самостоятельного решения и вопросы для анализировано результатов.

Электронное учебно-методическое пособие предназначено для аспирантов ННГУ, обучающихся по основной профессиональной образовательной программе аспирантуры 01.04.07 Физика конденсированного состояния, студентов старших курсов, изучающих курс параллельное программирование в физических исследованиях, а также для использования в УНИК «Новые многофункциональные материалы и нанотехнологии».

СОДЕРЖАНИЕ

Список используемых сокращений

GPU - графический процессор (англ.Graphics Processing Unit)

CPU - центральное обрабатывающее устройство (англ. Central Processing Unit)

CUDA - программно-аппаратная архитектура параллельных вычислений (англ. Compute Unified Device Architecture)

SIMD - одна инструкция на множество потоков данных (англ. Single Instruction Multiple Data)

MIMD - множество инструкций на множество потоков данных (англ. Multiple Instruction Multiple Data)

SIMT - одна инструкция на множество нитей (англ. Single Instruction, Multiple Thread)

ALU - арифметико-логическое устройство (англ. Arithmetic and Logic Unit)

VS - Microsoft Visual Studio

МК - квантовый метод Монте-Карло

ВВЕДЕНИЕ

Компьютерные технологии активно развиваются и ежегодно наращивают вычислительные возможности, согласно хорошо известному закону Гордона Мура, о том что количество транзисторов на интегральной схеме должно удваиваться каждые два года. Соответственно производители чипов для компьютеров в погоне за быстродействием с каждым запуском новых линеек минимизируют размеры планарных кремниевых транзисторов. На сегодняшний день технологический лидер, компания Intel, в апреле 2014 года запустил серийное производство чипов с технологией 22 нм. Однако существует ограничение ("кремниевый предел") при достижении атомных размеров (~ 10 нм), связанное с квантовыми эффектами, что запрещает дальнейшее уменьшение размеров транзистора и ведет к нарушению закона Мура. Таким образом, существует физическое ограничение, которое накладывает запрет на увеличение частоты одного процессора, поэтому производители современных вычислительных систем стремятся увеличить число процессоров и ядер. Однако, как бы быстро не работали многоядерные процессоры, существует ещё одна проблема, ограничивающая быстродействественность вычислений – это относительно невысокая скорость работы оперативной памяти, соответственно, большая энергозатратность.

Другой способ параллелизма начал развиваться в связи с развитием графических процессоров GPU (Graphics Processing Unit). В 2006 году появилась программно-аппаратный комплекс компании Nvidia - технология CUDA (от англ. *Compute Unified Device Architecture*), позволяющая программистам разрабатывать алгоритмы на известных языках программирования (C, Fortran) с расширениями, выполнимые на графических процессорах видеокарт и тем самым реализовать массово-параллельные вычисления. CUDA реализует аппаратный параллелизм, базируясь на принципах вычислений SIMD (от англ. *Single Instruction Multiple Data*), т.е.

позволяет применять одни и те же команды параллельно к множеству данных.

В рамках данной работы описываются возможности применения технологии CUDA [1] в научных исследованиях, описываются принципы распараллеливания и приводится пример работающей программы – расчет диссипативной динамики многоуровневой квантовой системы, на примере, квантового нелинейного осциллятора. Разработаны два алгоритма решения поставленной задачи: первый, основанный на стохастическом методе квантовых траекторий (квантовом методе Монте-Карло) и второй метод на основе некоммутативного интегрирования на группах Ли (разложение матрицы по Матрицам Гелмана). На базе примера демонстрируется эффективность работы графических ускорителей, масштабируемость и описывается возможность взаимодействия нескольких графических ускорителей с применением технологии MPI.

Апробация разработанных программ, данного методического пособия, проходила на оборудовании, закупленном в рамках программы развития ННГУ как национального исследовательского университета (вычислительного кластера НИФТИ ННГУ), а так же на оборудовании приобретенного в рамках программы повышения конкурентоспособности ННГУ им. Н.И. Лобачевского – вычислительного кластера "Лобачевский".

Развитая в работе техника расчета диссипативной динамики многоуровневых квантовых систем может быть естественным образом распространена на более сложные системы. Например, используемый квантовый метод Монте-Карло применим для моделирования работы многих других квантовых объектов (например, квантовых ям, квантовых точек, ионов в ловушках и т.д.) в реальных условиях взаимодействия с внешним окружением, а также для изучения работы генератора одиночных фотонов, полупроводниковых транзисторов и генераторов на основе квантовых точек и др.

1 Технологии параллельных вычислений

В данном разделе кратко обсуждаются принципы работы на графических устройствах, понимание которых невозможно без знания архитектуры графических ускорителей. В силу этого, мы кратко обсуждаем принципиальные различия архитектуры между центральными процессорами и графическими, проводим обзор по расширению языка C для CUDA и работой с памятью на устройстве, а также обсуждаем перспективность использования данной технологии в научных исследованиях.

1.1 Архитектура CPU и GPU систем и их принципы работы

Высокопроизводительные вычисления на сегодняшний день плотно вошли в различные сферы человеческой деятельности: научных исследованиях, промышленном производстве, образовании [2]. Уровень развития вычислительной техники и методов математического моделирования позволяет выполнять симуляцию и анализ природных и технологических процессов, что обеспечивает эффективное решение научных и производственных задач, предоставляя промышленному производству конкурентное преимущество, а исследователям – лидирующие позиции в современной науке. Спектр применения суперкомпьютерных технологий чрезвычайно широк: авиа-, судо- и машиностроение, строительство, нефтегазодобывающая промышленность, моделирование физических, химических и климатических процессов, медицина, биотехнологии и многое другое.

Как обсуждалось во введении, быстродействие вычислительных устройств происходит из-за размещения нескольких ядер на одном чипе, так как увеличение тактовых частот процессоров наложены ограничения: физические ("кремнивый предел") и энергетические. Современные процессоры (CPU, англ. *Central Processing Unit,* центральное

обрабатывающее устройство) наших персональных компьютеров содержат обычно до 8 ядер, однако в продаже имеются и более сложные многоядерные системы с количеством ядер порядка 100 и они обеспечивают выполнение до 10^{16} операций с плавающей точкой в секунду (10 Петафлоп) при запусках тестов Linpack (но существуют другие подходы к оценке их производительности) [3]. Достаточно ли подобной вычислительной мощности для решения современных задач? Конечно, любая имеющаяся производительность будет полностью потреблена пользователями, но даже грубая оценка необходимых объемов вычислений показывает, что для решения многих задач требования к производительности отличаются на порядки от возможностей существующих ресурсов, например:

- моделирование номинальных и переходных режимов работы ядерного реактора: для расчета процессов в реакторной установке требуется порядка 100 часов на машине производительностью 1 Эфлопс [4];

- проектирование и разработка изделий в авиастроении: при использовании метода прямого численного моделирования требуется использовать сетку размером 10^{16} элементов и выполнять не менее 10^{6} шагов моделирования по времени [4];

- квантово-химические расчеты малых систем для одного состояния методом связанных кластеров (Coupled Cluster Singles and Doubles, CCSD) для системы из нескольких десятков (~50) атомов требуется порядка 10^{14} операций; при решении оптимизационных задач требуется выполнение расчетов для десятков тысяч состояний (при большом количестве измерений – миллионов).

Центральные процессоры (CPU) используют архитектуру типа MIMD (англ. *Multiple Instruction Multiple Data,* множество инструкций – множество потоков данных), то есть одновременно несколько ядер могут работать совершенно независимо. Однако позже из-за возросших требований графических приложений универсальные процессоры начали поддерживать

специализированные векторные возможности, например, разработанный компанией Intel, набор инструкций для расширению процессора SSE2 (начиная с процессоров серии Pentium 4), основываясь на архитектуре SIMD (англ. *Single Instruction Multiple Data*, одна инструкция — множество данных). Именно поэтому для определённых задач применение GPU (англ. *Graphics Processing Unit*, графическое устройство) эффективнее. Процессоры на GPU куда проще, чем ядра CPU, они могут выполнять только математические операции и не способны к самостоятельной деятельности (являются сопроцессорами к центральному процессору). GPU основаны на архитектуре SIMT (англ. *Single Instruction, Multiple Thread,* одна инструкция — множество нитей (потоков)), то есть одновременно на графическом адаптере может выполняться несколько потоков вычислений, каждый из которых работает с большим набором данных.

Рассмотрим схематическое устройство процессора. Как показано на рис. 1 (а), большую часть площади вычислительного кристалла занимает кэш-память, а вычислительные модули (ALU) занимают лишь четверть кристалла. Кроме того, каждый отдельный ALU является полноценным центральным процессором. Он способен поддерживать все аппаратные прерывания, может работать со всеми устройствами ввода/вывода, что, несомненно, полезно для его функционирования как центрального процессора, однако становится излишним для его использования как векторного вычислительного модуля. Кроме того, потоки, выполняемые на центральном процессоре, являются очень «тяжеловесными», ими управляет операционная система, поэтому их не может быть много (максимальное количество измеряется несколькими тысячами).

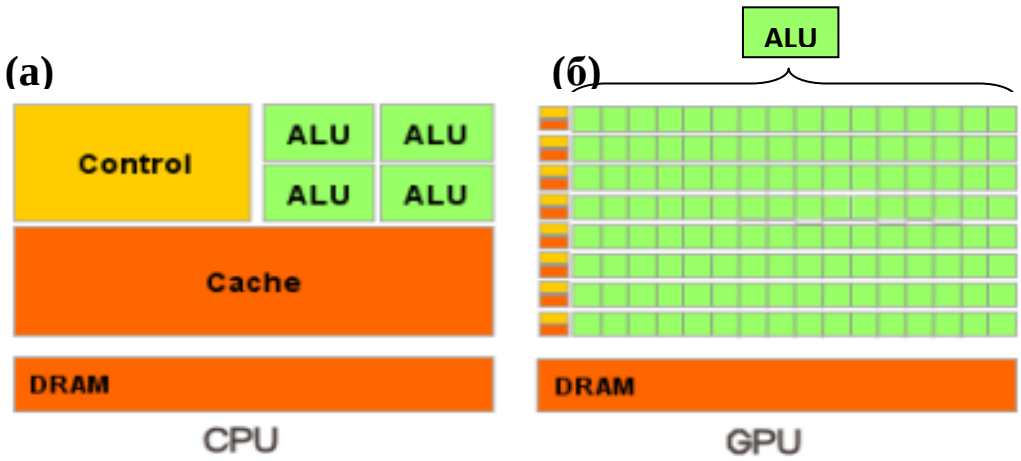


Рис. 1 Схематическое изображение архитектуры центрального процессора CPU (а) и графического адаптера GPU (б).

Теперь сравним с графическим процессором (GPU) рис. 1 (б) видно, что большая часть площади вычислительного кристалла занята вычислительными устройствами (ALU), доля кэша и устройств управления очень мала. Процессоры на GPU куда проще, чем ядра CPU, они могут выполнять только математические операции и не способны к самостоятельной деятельности (являются сопроцессорами к центральному процессору).

1.2 Общие принципы работы GPU-ускорителей

Технология CUDA – это программно-аппаратная вычислительная архитектура NVIDIA, основанная на расширении языка Си, которая даёт возможность организации доступа к набору инструкций графического ускорителя и управления его памятью при организации параллельных вычислений. CUDA помогает реализовывать алгоритмы [1, 5-7], выполнимые на графических процессорах видеоускорителей GeForce восьмого поколения и старше, а также Quadro и Tesla.

Графический процессор представляется в виде набора независимых мультипроцессоров (multiprocessors). Каждый мультипроцессор состоит из

нескольких CUDA-ядер (CUDA cores), нескольких модулей для вычисления математических функций (SFU), конвейера, а также разделяемой памяти (shared memory) и, кэша (для определенных видов памяти).

Технология CUDA позволяет определять специальные функции – ядра (kernels), которые выполняются параллельно на CPU в виде множества различных потоков (threads) (архитектура SIMT). Таким образом, ядро является аналогом потоковой функции. Каждый поток исполняется на одном CUDA-ядре, используя собственный набор инструкций и локальную память.

Отдельные потоки группируются в блоки потоков (thread block) одинакового размера, при этом каждый блок потоков выполняется на отдельном мультипроцессоре. Количество потоков в блоке ограничено (максимальные значения для конкретных устройств могут быть найдены в [7] или получены во время выполнения при помощи функций CUDA API). Потоки внутри блока потоков могут эффективно взаимодействовать между собой с помощью общих данных в разделяемой памяти и синхронизации. Кроме того, потоки могут взаимодействовать при помощи глобальной памяти и атомарных операций.

На аппаратном уровне потоки блока группируются в так называемые варпы (warps) по 32 элемента (на всех текущих устройствах), внутри которых все потоки параллельно выполняют одинаковые инструкции (по принципу SIMD). Важным моментом является то, что потоки фактически выполняют одну и ту же команды, но каждая со своими данными. Поэтому если внутри варпа происходит ветвление (например в результате выполнения оператора if), то все нити варпа выполняют все возникающие при этом ветви. По этой причине операции ветвления могут негативно сказываться на производительности – различные пути не могут выполняться параллельно (в тоже время потоки одного варпа, выполняющие один путь, работают параллельно).

В свою очередь, блоки потоков объединяются в решетки блоков потоков (grid of thread blocks). Следует отметить, что взаимодействие потоков из разных блоков во время работы ядра затруднено: отсутствуют явные инструкции синхронизации, взаимодействие возможно через глобальную память и использованием атомарных функций (другим вариантом является разбиение ядра на несколько ядер без внутреннего взаимодействия между потоками разных блоков).

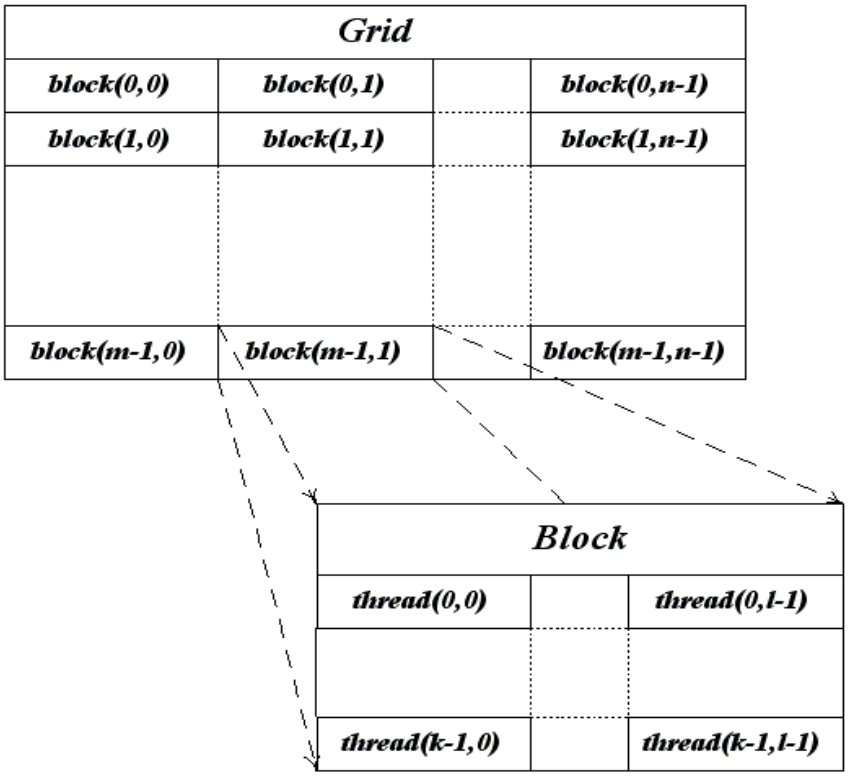


Рис. 2 Иерархия потоков CUDA

Каждый поток внутри блока потоков имеет свои координаты (одно-, двух- или трехмерные), которые доступны через встроенную переменную threadIdx. В свою очередь, координаты блока потоков (одно-, двух- или трехмерные) внутри решетки определяются встроенной переменной blockIdx. Пример иерархии потоков приведен на рис. 2. Данные встроенные переменные являются структурами с полями *.x*, *.y*, *.z*.

1.3 Работа с памятью в CUDA

Кроме иерархии потоков существует также несколько различных типов памяти. Быстродействие приложения очень сильно зависит от скорости работы с памятью. Именно поэтому в традиционных CPU большую часть кристалла занимают различные кэши, предназначенные для ускорения работы с памятью (в то время как для GPU основную часть кристалла занимают ALU).

В CUDA для GPU существует несколько различных типов памяти, доступных нитям, сильно различающихся между собой (см. табл. 1) [1].

Таблица 1. Типы памяти в CUDA

Тип памяти	**Доступ**	**Уровень выделения**	**Скорость работы**
регистры (registers)	R/W	per-thread	высокая (on chip)
local	R/W	per-thread	низкая (DRAM)
shared	R/W	per-block	высокая (on-chip)
global	R/W	per-grid	низкая(DRAM)
constant	R/O	per-grid	высокая(on chip L1 cache)
texture	R/O	per-grid	высокая(on chip L1 cache)

Глобальная память – самый большой объём памяти, доступный для всех мультипроцессоров на видеочипе, размер составляет от 256 мегабайт до 1.5 гигабайт на текущих решениях (и до 4 Гбайт на Tesla). Обладает высокой пропускной способностью, более 100 гигабайт/с для топовых решений NVIDIA, но очень большими задержками в несколько сот тактов. Не кэшируется, поддерживает обобщённые инструкции load и store, и обычные указатели на память.

Локальная память – это небольшой объём памяти, к которому имеет доступ только один потоковый процессор. Она относительно медленная – такая же, как и глобальная.

Разделяемая память – это 16-килобайтный (в видеочипах нынешней архитектуры) блок памяти с общим доступом для всех потоковых процессоров в мультипроцессоре. Эта память весьма быстрая, такая же, как регистры. Она обеспечивает взаимодействие потоков, управляется разработчиком напрямую и имеет низкие задержки. Преимущества разделяемой памяти: использование в виде управляемого программистом кэша первого уровня, снижение задержек при доступе исполнительных блоков (ALU) к данным, сокращение количества обращений к глобальной памяти.

Память констант – область памяти объемом 64 килобайта (то же – для нынешних GPU), доступная только для чтения всеми мультипроцессорами. Она кэшируется по 8 килобайт на каждый мультипроцессор. Довольно медленная – задержка в несколько сот тактов при отсутствии нужных данных в кэше.

Текстурная память – блок памяти, доступный для чтения всеми мультипроцессорами. Выборка данных осуществляется при помощи текстурных блоков видеочипа, поэтому предоставляются возможности линейной интерполяции данных без дополнительных затрат. Кэшируется по 8 килобайт на каждый мультипроцессор. Медленная, как глобальная – сотни тактов задержки при отсутствии данных в кэше.

Естественно, что глобальная, локальная, текстурная и память констант – это физически одна и та же память, известная как локальная видеопамять видеокарты. Их отличия в различных алгоритмах кэширования и моделях доступа. При этом центральный процессор (CPU) имеет R/W доступ только к глобальной, константной и текстурной памяти (находящейся в DRAM GPU)

и только через функции копирования памяти между CPU и GPU (предоставляемые CUDA API).

1.4 Расширения языка C для работы с CUDA

Программы для CUDA (соответствующие файлы обычно имеют расширение ***.cu***) пишутся на «расширенном» C и компилируются при помощи команды **nvcc**.

Вводимые в CUDA расширения языка C состоят из

- спецификаторов функций, показывающих, где будет выполняться функция и откуда она может быть вызвана;
- спецификаторы переменных, задающие тип памяти, используемый для данной переменных;
- директива, служащая для запуска ядра, задающая как данные, так и иерархию потоков;
- встроенные переменные, содержащие информацию о текущем потоке;
- *runtime*, включающий в себя дополнительные типы данных

Таблица 2. Спецификаторы функций в CUDA

Спецификатор	Выполняется на	Может вызываться из
__device__	device	device
__global__	device	host
__host__	host	host

При этом спецификаторы **__host__** и **__device__** могут быть использованы вместе (это значит, что соответствующая функция может выполняться как на GPU, так и на CPU - соответствующий код для обеих платформ будет автоматически сгенерирован компилятором). Спецификаторы **__global__** и **__host__** не могут быть использованы вместе.

Спецификатор **__global__** обозначает ядро и соответствующая функция должна возвращать значение типа **void**.

На функции, выполняемые на GPU (**__device__** и **__global__**) накладываются следующие ограничения:

- нельзя брать их адрес (за исключением **__global__** функций);
- не поддерживается рекурсия;
- не поддерживаются **static**-переменные внутри функции;
- не поддерживается переменное число входных аргументов.

Для задания размещения в памяти GPU переменных используются следующие спецификаторы - **__device__**, **__constant__** и **__shared__**. На их использование также накладывается ряд ограничений:

- эти спецификаторы не могут быть применены к полям структуры (**struct** или **union**);
- соответствующие переменные могут использоваться только в пределах одного файла, их нельзя объявлять как **extern;**
- запись в переменные типа **__constant__** может осуществляться только CPU при помощи специальных функций;
- **__shared__** переменные не могут инициализироваться при объявлении.

Добавленные переменные

В язык добавлены следующие специальные переменные

- *gridDim* - размер *grid*'а (имеет тип **dim3**);
- *blockDim* - размер блока (имеет тип **dim3**);
- *blockIdx* - индекс текущего блока в *grid*'е (имеет тип **uint3**);
- *threadIdx* - индекс текущей нити в блоке (имеет тип **uint3**);
- *warpSize* - размер *warp*'а (имеет тип **int**).

В язык добавляются 1/2/3/4-мерные вектора из базовых типов - char1, char2, char3, char4, uchar1, uchar2, uchar3, uchar4, short1, short2, short3, s

hort4, ushort1, ushort2, ushort3, ushort4, int1,int2, int3, int4, uint1, uint2, uint3, ui nt4, long1, long2, long3, long4, ulong1, ulong2, ulong3, ulong4, float1, float2, floa t3, float2, и double2.

Обратите внимание, что для этих типов (в отличии от шейдерных языков GLSL/Cg/HLSL) не поддерживаются векторные покомпонентные операции, т.е. нельзя просто сложить два вектора при помощи оператора "+" - это необходимо явно делать для каждой компоненты.

Также для задания размерности служит тип **dim3**, основанный на типе **uint3**, но обладающий нормальным конструктором, инициализирующим все не заданные компоненты единицами.

Директива вызова ядра

Для запуска ядра на GPU используется следующая конструкция:

kernelName <<<Dg,Db,Ns,S>>> (args)

kernelName – это имя (адрес) соответствующей __global__ функции;

Dg – переменная (или значение) типа dim3, задающая размерность и размер grid'a (в блоках);

Db – переменная (или значение) типа dim3, задающая размерность и размер блока (в нитях);

Ns – переменная (или значение) типа size_t, задающая дополнительный объем shared-памяти, которая должна быть динамически выделена (к уже статически выделенной shared-памяти), S - переменная (или значение) типа cudaStream_t задает поток (CUDA stream), в котором должен произойти вызов, по умолчанию используется поток 0.

args - аргументы вызова функции kernelName.

Также в язык C добавлена функция **__syncthreads**, осуществляющая синхронизацию всех нитей блока. Управление из нее будет возвращено только тогда, когда все нити данного блока вызовут эту функцию. Т.е. когда весь код, идущий перед этим вызовом, уже выполнен (и, значит, на его

результаты можно смело рассчитывать). Эта функция очень удобная для организации бесконфликтной работы с *shared*-памятью.

Также CUDA поддерживает все математические функции из стандартной библиотеки C, однако с точки зрения быстродействия лучше использовать их *float*-аналоги (а не *double*) - например *sinf*. Кроме этого CUDA предоставляет дополнительный набор математических функций (*__sinf*, *__powf* и т.д.) обеспечивающие более низкую точность, но заметно более высокое быстродействие чем *sinf*, *powf* и т.п.

Технология CUDA поддерживает следующие стандартные библиотеки:

- CUBLAS – реализация интерфейса программирования приложений для создания библиотек, выполняющих основные операции линейной алгебры BLAS (англ. *Basic Linear Algebra Subprograms*);
- CUSPARSE – содержит набор базовых подпрограмм линейной алгебры, используемых для обработки разреженных матриц;
- CUSP – реализует параллельные алгоритмы для решения систем линейных алгебраических уравнений с разреженными матрицами;
- CUFFT – реализация библиотеки быстрого преобразования Фурье (англ. *Fast Fourier Transform*).

1.5 Установка драйвера и создание проектов CUDA под Windows

Перед началом установки необходимо проверить, поддерживает ли ваша видеокарта технологию CUDA (это можно сделать на официальном сайте Nvidia: https://developer.nvidia.com/cuda-gpus)

Для разработки и запуска приложений необходимо:

- графический процессор, поддерживающий CUDA;
- драйвер для устройства и CUDA Toolkit;

- программное обеспечение Nvidia, которое можно бесплатно загрузить с сайта https://developer.nvidia.com/cuda-downloads
- среда разработки программ (Microsoft Visual Studio, Netbeans, etc.).

В данном методическом пособии будет рассмотрена интеграция и установка CUDA (версия 6.5) с помощью Microsoft Visual Studio. Отметим, что у Вас должна иметься уже установленная среда разработки Microsoft Visual Studio 2010 (иначе необходимо пройти регистрацию на сайте http://www.dreamspark.ru/ и скачать лицензионную профессиональную версию Microsoft Visual Studio).

Этапы установки:

1) Необходимо загрузить и установить драйвер для работы CUDA, который интегрирован в последние версии Nvidia ForceWare (см. http://www.nvidia.com/drivers).

2) Скачать драйверы для разработчиков (CUDA Developer Drivers) находятся в разделе CUDA на сайте Nvidia: https://developer.nvidia.com/cuda-downloads. Данный драйвер включает в себя программный пакет CUDA Toolkit, который содержит инструменты, библиотеки, заголовочные файлы для компиляции программ с помощью Microsoft Visual Studio. GPU Computing SDK содержит в себе демонстрационные примеры, которые уже предварительно сконфигурированы для удобной работы в среде Microsoft Visual Studio, и является исключительно полезным при изучении технологии, но не требуется для создания и запуска приложений.

3) Запустите установочный пакет (для версии CUDA 6.5 cuda_6.5.14_windows_general_64.exe) и поэтапно следуйте процессу установки. Инструментарий CUDA по умолчанию устанавливается в каталог C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v6.5. В процессе установки автоматически создаются соответствующие переменные окружения (в том числе CUDA_LIB_PATH, CUDA_INC_PATH), для настройки можно использовать их.

4) После завершения установки проверьте интеграцию CUDA v.6.5 в среду Microsoft Visual Studio 2010. Для этого откройте VS и пройдите по пути: File → New → Project и Вы должны увидеть в списке инсталированных шаблонов (Installed Templates) Nvidia → CUDA6.5 (см. рис. 3).

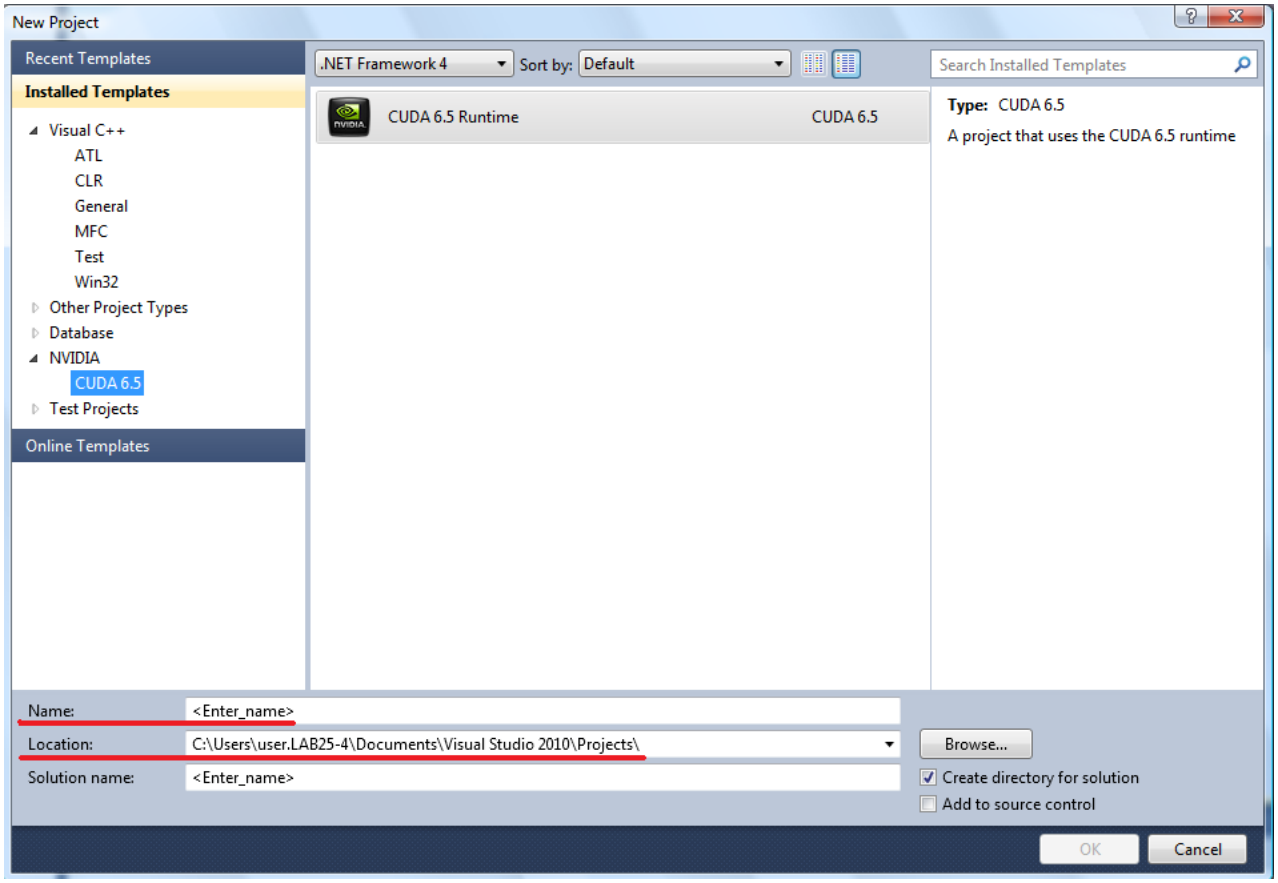


Рис. 3 Создание проектов CUDA 6.5 в среде VS 2010.

После установки драйвера CUDA и проверки интегрируемости его в среду VS можно перейти к созданию проектов, использовавших GPU ускорители:

1) В программной среде VS переходим по пути: File → New → Project выбираем шаблон Nvidia → CUDA6.5 выбираем директорию (Location), где будет хранится проект (по умолчанию Documents\Visual Studio 2010\Projects) и задаем имя проекта (Name) на рис. 3 эти поля подчёркнуты. После этого кнопка нажимаем "Ok" и появляется пустой проект с файлом kernel.cu, который является стандартным при создании проектов с CUDA и является тестом.

2) Внимательно ознакомьтесь с текстом и написанием данного примера.

3) Собираем проект выбирая вкладку Build → Build Solution в режиме отладочной версии Debug, а затем в пользовательской версии Release . После этого в папке проекта появятся папки Debug и Release.

4) Запустите находящиеся там exe-файлы и если всё выполнилось без ошибок, то CUDA установилась верно. Если тест пройден неудачно, убедитесь, что ваше устройство действительно поддерживает CUDA, и что Вы правильно установили всё необходимое программное обеспечение.

1.6 Разница между CPU и GPU в параллельных вычислениях

В заключение данного раздела резюмируем всё, что обсуждалось ранее и подчеркнём принципиальные различия между двумя технологиями распараллеливания вычислений на центральных процессорах и на графических:

1) **Многопоточность.** Ядра CPU созданы для исполнения одного потока последовательных инструкций с максимальной производительностью, а GPU проектируются для быстрого исполнения большого числа параллельно выполняемых потоков инструкций. Таким образом, обычные CPU системы могут поддерживать лишь ограниченное число одновременных потоков. На серверах, например, которые имеют четыре quad-процессора, могут работать одновременно только 16 параллельных потоков (или 32 потока, если есть поддержка HyperThreading). Например, минимальная исполняемая единица на GPU устройстве, которая называется блоком задач (warp) и состоит из 32 потоков или нитей (threads) [6]. Все графические процессоры NVIDIA могут поддерживать 768 активных потоков на каждый мультипроцессор, а некоторые более поздние версии до 1024 потоков на мультипроцессор. Устройства, у которых более 30 потоковых мультипроцессора(GeForce GTX280 например) могут создавать до 30000 активных потоков. Более того, устройства могут выполнять миллиарды задач благодаря многопоточности.

2) **Потоки.** Потоки на CPU обычно требуют много ресурсов. Операционная система должна обменивать потоки между собой, включать и выключать каналы обработки для обеспечения многопоточности. Переключение контекста (когда два потока меняются местами) слишком медленное и дорогостоящее в смысле производительности. Для сравнения, на GPU потоки в десятки, а то и сотни раз "легче" потоков CPU. В типичной системе сотни потоков стоят в очереди на выполнение в блоках задач по 32 потока в каждом. Если процессору GPU приходится ждать в одном из warp'ов, он просто переключается на выполнение другого потока. Так как регистры выделяются под активные потоки, не происходит обмена регистров и состояний между потоками GPU. Время жизни ресурсов, выделяемых на каждый поток, равно времени жизни самого потока.

3) **Доступ к памяти.** Обе системы - CPU и GPU имеют оперативную память. На CPU системе оперативная память доступна всему коду (в пределах ограничений, накладываемых операционной системой). На GPU устройстве, память разделяется виртуально и физически на различные типы [5], каждый из которых имеет свое специальное назначение и выполняет различные задачи. В результате этого видеочипам (GPU) доступна в разы большая пропускная способность памяти, что также весьма важно для параллельных расчётов, оперирующих с огромными потоками данных.

4) **Кэширование.** Универсальные центральные процессоры (CPU) используют кэш-память для увеличения производительности за счёт снижения задержек доступа к памяти, а GPU используют кэш или общую память для увеличения полосы пропускания [5]. CPU снижают задержки доступа к памяти при помощи кэш-памяти большого размера, а также предсказания ветвлений кода. Эти аппаратные части занимают большую часть площади чипа и потребляют много энергии. Видеочипы обходят проблему задержек доступа к памяти при помощи одновременного исполнения тысяч потоков — в то время, когда один из потоков ожидает

данных из памяти, видеочип может выполнять вычисления другого потока без ожидания и задержек.

В результате всех описанных выше отличий, теоретическая производительность видеочипов значительно превосходит производительность CPU. Компания NVIDIA приводит такой график роста производительности CPU и GPU за последние несколько лет (см. рис. 4).

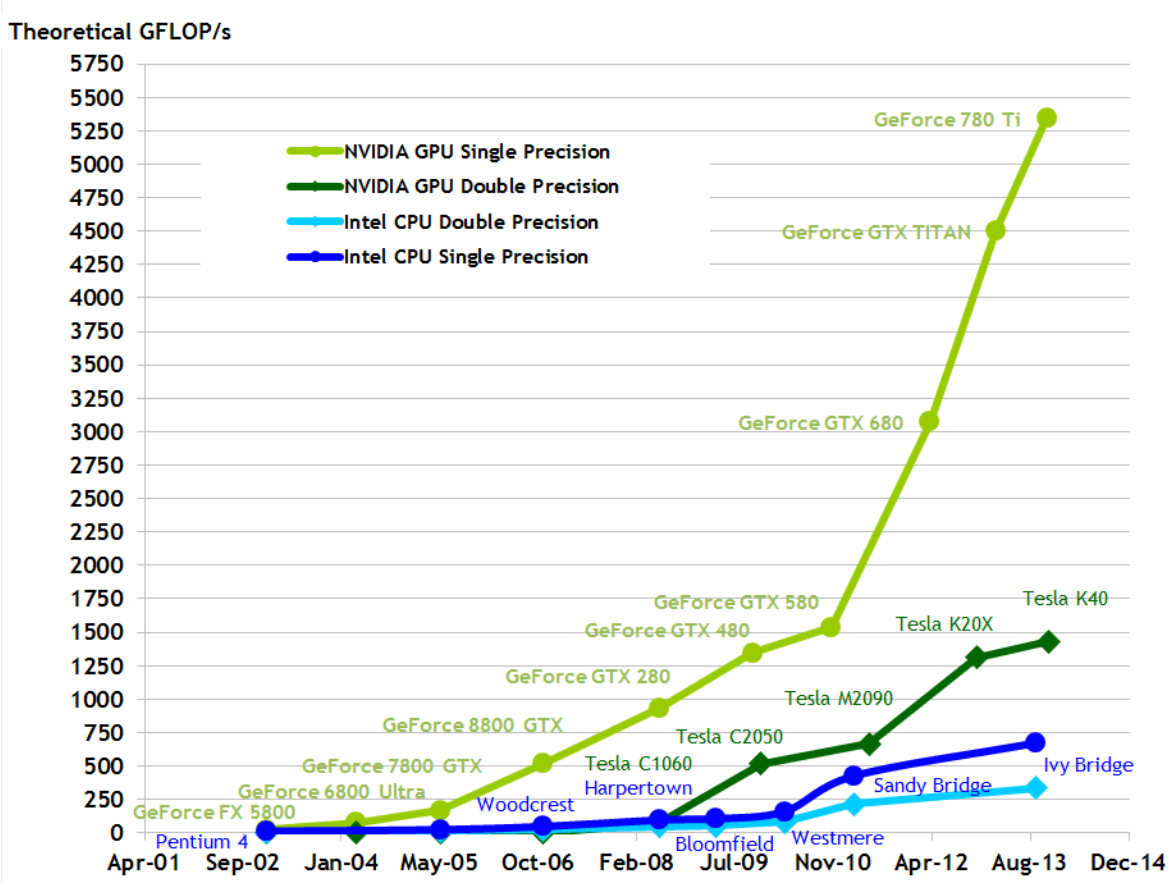


Рис. 4 Количество операций с плавающей запятой в секунду для CPU и GPU.

Однако не следует забывать, что данная теоретическая производительность может быть достигнута только при учете эффективного использования мощности GPU и распараллеливания алгоритма вычислительной задачи на сотни исполняемых блоков.

Например, в научных исследованиях можно привести примеры реализованных задач (рис. 5), которые показали значительное ускорение при использовании синтетического кода на GPU против SSE-векторизованного кода на CPU (по данным NVIDIA):

- флуоресцентная микроскопия: в 12 раз по сравнению с расчетом на CPU (12x);

- молекулярная динамика (non-bonded force calc): 8-16x;

- электростатика (прямое и многоуровневое суммирование Кулона): 40-120x и 7x.

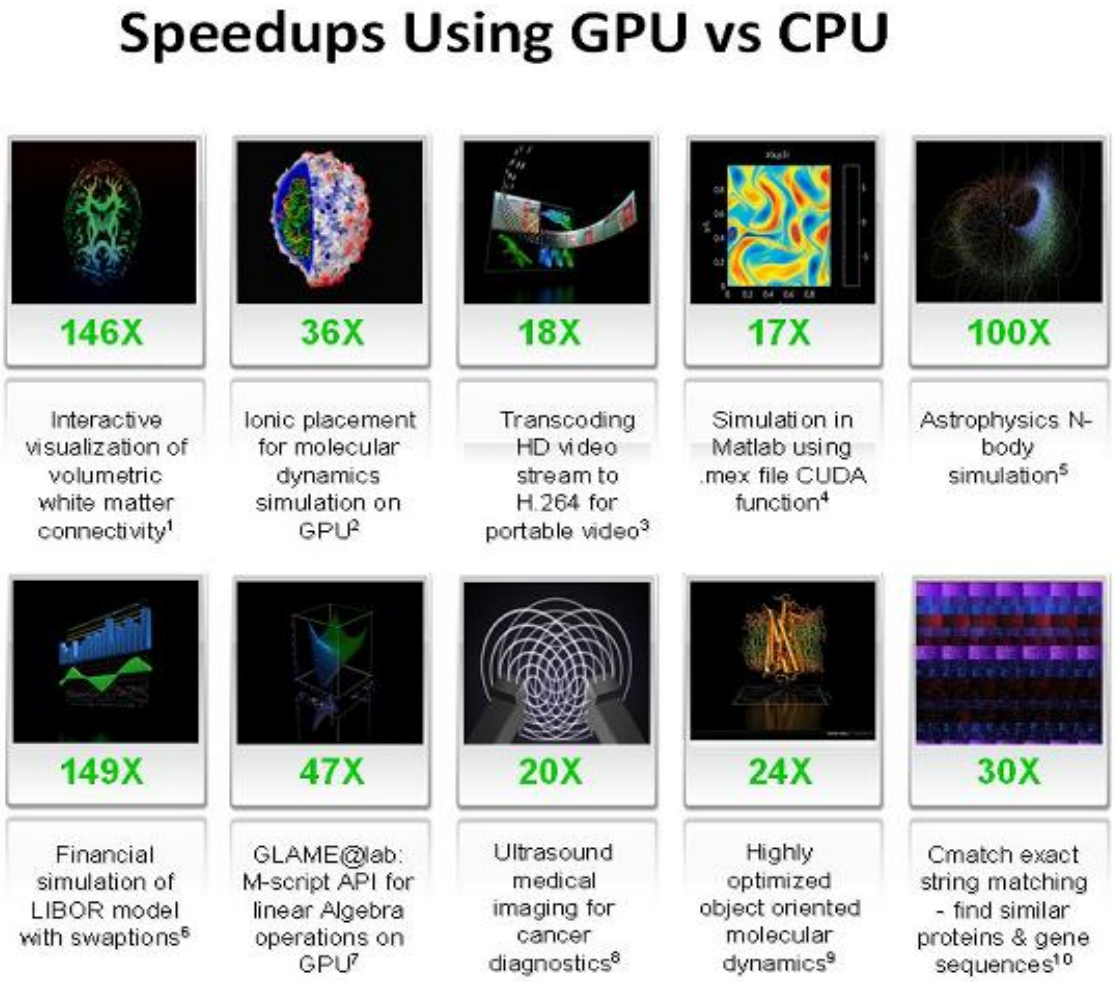


Рис. 5 Примеры использования GPU-ускорителей в научных исследованиях и полученные ускорения по сравнению с CPU. Данные взяты с официального сайта NVidia 2008 г.

2 Применение технологии CUDA для расчета диссипативной динамики многоуровневой квантовой системы

В рамках данной работы описываются возможности применения технологии CUDA в научных исследованиях, описываются принципы распараллеливания и демонстрируется пример работающей программы – расчет диссипативной динамики многоуровневой квантовой системы, на примере нелинейного квантового осциллятора. Разработаны и реализованы два алгоритма решения данной задачи: методом квантовых траекторий (квантовым методом Монте-Карло) и с помощью метода некоммутативного интегрирования на группах Ли (разложение матрицы по матрицам Гелл-Манна).

Квантовые эффекты играют ключевую роль в современных электронных устройствах: квантовых точках, квантовых нитях, сверхпроводниковых джозефсоновских кольцах с встроенными связями и т.п. В таких системах размер волновой функции квазичастиц или сверхпроводящего конденсата сравнивается с размером системы, а спектр возбуждений становится дискретным. Если энергия теплового движения меньше расстояния между уровнями, то эволюция систем описывается уравнениями квантовой механики. Вместе с тем, подобные наноструктуры могут состоять из макроскопического числа атомов, они также могут взаимодействовать с окружением, которое приводит к случайному сбою фаз волновой и разрушению квантовой интерференции. Указанные эффекты особенно важны при работе связанных кубитов - двухуровневых квантовых систем, - которые являются базисным элементом устройств квантовой логики [8-11]. Кроме того, измерение над системой или считывание информации, реализуется как проектирование состояний кубитов на состояние измерительного устройства ("метра")[12-15], роль которого обычно играет осциллятор, также подверженный влиянию окружающей среды.

В данной работе рассматривается простой пример многоуровневой квантовой системы: квантовый нелинейный осциллятор, который используется для проективных измерений состояний кубита. Влияние окружения на осциллятор имитируется его взаимодействием с бозонным термостатом. При условии слабого обратного воздействия на термостат и локального по времени возмущения (марковское приближение), динамика системы может быть описана уравнением для матрицы плотности. Поскольку размерность гильбертова пространства состояний осциллятора неограниченно, практически мы вынуждены ограничится конечномерной аппроксимацией, вводя вектор состояний размерности N. В то же время, уравнение для матрицы плотности приводит к необходимости интегрирования $N \times N$ дифференциальных уравнений. Новым вычислительным аспектом данной задачи является необходимость вовлечения в динамику и процессы релаксации огромного числа уровней N ($N > 1000$) квантовой системы, что позволит наблюдать качественно новые эффекты квантово-классического соответствия, которая играет важную роль для интерпретации неразрушающих измерений. В свою очередь, рассмотрение эволюционной задачи большой размерности предполагает применение суперкомпьютерных технологий для решения квантового кинетического уравнения. Основная идея решения состоит в предварительном разложении матрицы плотности по полному ортогональному набору матриц специальной унитарной группы $SU(N)$ (матрицам Гелл-Манна). Это позволяет отобразить матрицу плотности на вектор размерности $N^2 - 1$, который эволюционирует согласно матричному уравнению. В работе описаны принципы распараллеливания и численного моделирования динамического уравнения, а также программный комплекс, позволяющий осуществлять высокопроизводительные параллельные расчеты на GPU.

2.1 Физическая модель и основные уравнения

Рассматриваемая система представляет собой джозефсоновский переход который можно представить эквивалентной схемой (см. Рис.1), состоящей из последовательно соединенной емкости C, слабой связи с критическим током I_c и генератора переменного тока $I(t)$. Согласно уравнениям Джозефсона [14] через диэлектрическую прослойку, разрывающую сверхпроводящую проволоку (слабую связь), может протекать ток $I = I_c \sin\theta$, где θ - разность фаз волновых функций сверхпроводящих электронов по разную сторону от прослойки. При этом напряжение V на переходе зависит от скорости изменения разности

фаз: $\frac{\partial\theta}{\partial t} = \frac{\hbar}{2e}V$ (здесь e - заряд электрона, $\hbar$ - постоянная Планка). С учетом уравнений

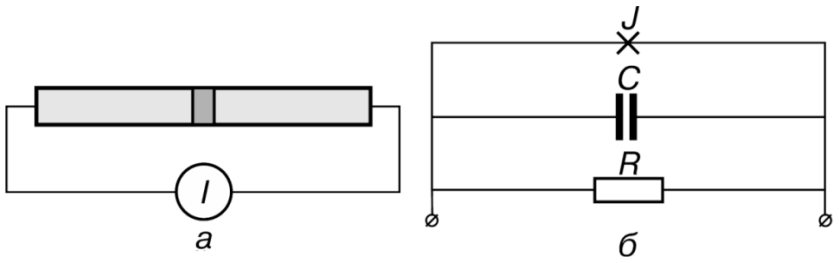


Рис. 1. Джозефсоновский туннельный контакт, подключенный к источнику тока (а), и его эквивалентная схема (б).

Джозефсона, энергия перехода может быть записана в виде:

$$H_J = \frac{4E_c}{\hbar^2} p^2 + E_J\left(1-\cos\theta\right) - \frac{\hbar}{2e} I(t)\theta \ , \qquad (1)$$

где $E_c = \frac{e^2}{2C}$ и $E_J = \frac{\hbar}{2e} I_c$ - соответственно электростатическая и джозефсоновская энергия перехода, $p = \frac{d\theta}{dt} \equiv \dot{\theta}$ - импульс, сопряженный фазе θ ("координате" осциллятора). Из (1) видно, что гамильтониан перехода аналогичен гамильтониану классического маятника, к которому приложен момент, зависящий от времени. Переход к квантовой теории осуществляется заменой классического импульса оператором импульса $\hat{p} = -i\hbar\frac{\partial}{\partial\theta}$ и требованием выполнения коммутационных соотношений: $\left[\hat{\theta}, \hat{p}\right] = i\hbar$. Таким образом гамильтониан квантового джозефсоновского осциллятора имеет вид:

$$\hat{H}_J = \frac{4E_c}{\hbar_2} \hat{p}^2 + E_J(1-\cos\theta) - \frac{\hbar}{2e} I(t)\theta \, . \qquad (2)$$

Квантовая динамика замкнутой системы описывается волновой функцией (вектором) $|\psi(t)\rangle$, который подчиняется уравнению Шредингера:

$$i\hbar\frac{\partial}{\partial t}|\psi(t)\rangle = \left(\frac{4E_c}{\hbar_2}\hat{p}^2 + E_J(1-\cos\theta) - \frac{\hbar}{2e} I(t)\theta\right)|\psi(t)\rangle \quad . \qquad (3)$$

При не слишком сильном возбуждении осциллятора, когда амплитуда тока мала по сравнением с критическим током, можно также считать угол отклонения малым, удержав в разложении по углу слагаемое $\sim\theta^4$:

$$\hat{H}_J = \frac{4E_c}{\hbar_2}\hat{p}^2 + \frac{E_J\theta^2}{2} - \frac{E_J\theta^4}{4!} - \frac{\hbar}{2e}I(t)\theta \qquad (4)$$

Вводя операторы рождения $\hat{a}^+$ и уничтожения $\hat{a}$ для параметризации координаты $\hat{\theta} = \left(\frac{2E_c}{E_J}\right)^{1/4}(\hat{a} + \hat{a}^+)$ и импульса осциллятора $\hat{p} = i\hbar\left(\frac{E_C}{2E_J}\right)^{-1/4}(\hat{a} - \hat{a}^+)$ с учетом коммутационных соотношений, можно получить эффективный гамильтониан в терминах вторичного квантования:

$$\hat{H}_J = \hbar\omega_0\hat{a}^+\hat{a} - \mu(\hat{a}^+ + \hat{a})^4 + f_0\cos(\omega t)(\hat{a} + \hat{a}^+), \qquad (5)$$

где собственная частота осциллятора $\omega_0 = \sqrt{\frac{2eI_c}{\hbar C}}$, параметр нелинейности $\mu = \frac{E_c}{12}$, амплитуда $f_0 = \frac{\hbar}{2e}\left(\frac{2E_c}{E_J}\right)^{1/4}$ и частота ω внешнего возмущения.

2.2 Модель релаксации

При рассмотрении динамики квантовой системы существенное влияние оказывают процессы декогерентности. Для джозефсоновского осциллятора, таковыми процессами являются: флуктуации заряда на джозефсоновских контактах, квазичастицы на островках сверхпроводимости, взаимодействие с ядерными спинами в подложке, затухание при взаимодействии с

электромагнитным управляющим полем и измерительным прибором. Процессы релаксации можно рассматривать как взаимодействие системы с бозонным резервуаром с большим числом степеней свободы. Отметим, что в экспериментах, когда джозефсоновский переход работает в чисто квантовом режиме, проводятся при криогенных температурах, поэтому вклад шума фермионных возбуждений мал и не учитывается в данной работе. Гамильтониан, соответствующий вкладу бозонных возбуждений имеет вид:

$$\hat{H}_{noise} = \hat{F}(t)\hat{a}^{+} + \hat{F}^{+}(t)\hat{a}, \tag{6}$$

где оператор термостата $\hat{F}(t)$ отвечает за релаксацию нелинейного джозефсоновского осциллятора. Ясно. что если система незамкнутая, то мы не можем использовать волновую функцию, подчиняющуюся уравнению (3).

Таким образом, полный гамильтониан, описывающий релаксационную динамику связанной системы имеет вид:

$$\hat{H} = \hat{H}_J + \hat{H}_{noise}, \tag{7}$$

где H_J определен формулой (3).

Исключая переменные термостата $\hat{F}(t)$ получим в борн-марковском приближении [16] для матрицы (оператора) плотности $\hat{\rho}(t)$ уравнение:

$$\frac{\partial\hat{\rho}}{\partial t} = \frac{1}{i\hbar}[\hat{H},\hat{\rho}] + \frac{\gamma}{2}(2\hat{a}\hat{\rho}\hat{a}^{+} - \hat{a}^{+}\hat{a}\hat{\rho} - \hat{\rho}\hat{a}^{+}\hat{a}), \tag{8}$$

где γ - параметр релаксации осциллятора. Отметим, что данные параметры выражаются стандартным образом через корреляционными функциями бозонного термостата $\langle R(t)R^{+}\rangle$ (см. вывод в [16]).

2.3 Краткий обзор методов решения уравнения для матрицы плотности

Аналитические решения уравнения могут быть получены только в простейших случаях. Например, когда переменный ток отсутствует

($I(t)=0$), уравнение (7) легко решить. В этом случае первоначально возбужденная система релаксирует за время $\sim\gamma^{-1}$ в равновесное состояние. Однако, даже для линейного осциллятора, возбуждаемого периодическим током, замкнутое выражение для матрицы плотности отсутствует.

Существует несколько методов численного решения уравнения для матрицы плотности. Прежде всего, для параметризации матрицы плотности используем базис, который образуют состояния линейного гармонического осциллятора $|n\rangle$ (так называемый фоковский базис), определяемый как собственный вектор оператора чисел заполнения: $\hat{n}|n\rangle = n|n\rangle, \hat{n} = a^{+}a$. Записывая уравнение (8) в фоковском базисе, получаем:

$$\begin{aligned}
&\frac{\partial \rho_{n,m}}{\partial t} = \frac{1}{i\hbar}\left(\hbar\omega_0(n-m) - \mu(n^2-m^2)\right)\rho_{n,m} + \\
&\frac{1}{i\hbar}\left(\sqrt{n}\rho_{n-1,m} + \sqrt{n+1}\rho_{n+1,m} - \sqrt{m+1}\rho_{n,m+1} - \sqrt{m}\rho_{n,m-1}\right) - \\
&\frac{\gamma}{2}\left((n+m)\rho_{n,m} - 2\sqrt{(n+1)(m+1)}\rho_{n+1,m+1}\right),
\end{aligned} \tag{9}$$

где матричные элементы $\hat{\rho}$ определены выражением $\rho_{n,m} = \langle n|\hat{\rho}|m\rangle$, $n,m = 0,\cdots,N-1$ Матричное уравнение имеет размерность $N\times N$, где N - максимальная длина вектора, которая допустима с точки зрения вычислений (конечномерная аппроксимацией фоковского вектора).

Простейший метод численного решения (сведение к уравнению Лиувилля) состоит в следующем. Распишем систему уравнений (9) по строкам или по столбцам для вектора

$$r = \{\rho_{0,0},...,\rho_{0,N-1},\rho_{1,0},...,\rho_{1,N-1},...,\rho_{N-1,0},...,\rho_{N-1,N-1}\}. \tag{10}$$

Справа формируется блочная матрица - оператор Лиувилля $\mathbb{L}$. Таким образом, задача сводится к решению уравнения Лиувилля для вектора **r:** $\dot{r} = \mathbb{L}r$**.** Эффективно мы увеличили размерность системы до $N \times N$, но свели задачу к численному решению системы дифференциальных линейных уравнений первого порядка, которые можно решить, например, методом Рунге-Кутты 4-го порядка. Однако данный метод не является эффективным при проведении параллельных вычислений, так как все $N \times N$ уравнений являются связанными друг с другом, а матрица блочно заполненной, поэтому для каждого последующего шага необходимо знать состояние на предыдущем шаге, следовательно распараллеливание оказывается затруднительным.

2.4 Метод разложения по матрицам Гелл-Манна

Более эффективный алгоритм можно реализовать, если использовать разложение матрицы плотности по ортогональному набору матриц специального вида - матрицам Гелл-Манна [18]. Эти матрицы содержат данные о симметрии системы, в соответствии с теории некоммутативных групп Ли. Обобщенными матрицами Гелл-Манна называют $N^2 - 1$ матриц, представляющие собой генераторы группы $SU(N)$, $N \geq 2$. Они являются обобщением хорошо известных матриц Паули, генераторов группы $SU(2)$ и матриц Гелл-Манна, генераторов группы $SU(3)$,.

Алгоритм построения обобщенных матриц Гелл-Манна состоит в следующем. Введем вспомогательных матриц $E_{jk} = |i\rangle\langle k|$, матричные элементы которых равны

$$\left(E_{jk}\right)_{lm} = \delta_{jl}\delta_{km}, \tag{11}$$

где $\delta_{i,j}$ – символ Кронекера. Используя (11), построим три набора матриц: симметричные, антисимметричные и диагональные. Симметричный набор матриц набор λ_{jk}^{s} определен выражением:

$$\lambda_{jk}^{s} = E_{jk} + E_{kj}, \tag{12}$$

где $1 \le j < k \le N$. Второй набор образуют антисимметричные матрицы λ_{jk}^{a}:

$$\lambda_{jk}^{a} = -i\left(E_{jk} - E_{kj}\right), \tag{13}$$

где i – мнимая единица, а $1 \le j < k \le N$. Наконец, набор диагональных матриц λ_{l}^{d} определяется выражением:

$$\lambda_{l} = \sqrt{\frac{2}{l\left(l+1\right)}}\left(\sum_{j=1}^{l} E_{jj} - lE_{l+1,l+1}\right), \tag{14}$$

$1 \le l \le N-1$. Нетрудно проверить, что полное число матриц равно $\frac{N\left(N-1\right)}{2} + \frac{N\left(N-1\right)}{2} + \left(N-1\right) = N^2 - 1$. Чтобы иметь полный набор матриц, к перечисленным необходимо добавить любую матрицу, пропорциональную единичной матрице I. Один из вариантов заключается в следующем. Полагаем, что $E_{N+1,N+1} = 0$, а ряд диагональных матриц λ_{l} продолжается вплоть до $l \le N$, то есть $\lambda_{N}^{d} = \sqrt{\frac{2}{N\left(N+1\right)}} I$. Обобщенные матрицы Гелл-Манна вместе с единичной матрицей образуют полный набор, по которому можно разложить любую матрицу.

Для программной реализации элементы построенного базиса удобно расположить в следующей последовательности:

$$\Lambda_1 = \Lambda_{11},\ \Lambda_2 = \Lambda_{12},\ \ldots,\ \Lambda_N = \Lambda_{1N}, \tag{15}$$

$$\Lambda_{N+1} = \Lambda_{21},\ \Lambda_{N+2} = \Lambda_{22},\ ...,\ \Lambda_{2N} = \Lambda_{2N},$$

…

$$\Lambda_{N(N-1)+1} = \Lambda_{N1},\ \ldots,\ \Lambda_{N^2} = \Lambda_{NN}$$

где

$$\Lambda_{jk} = \begin{cases} \lambda_{jk}^{s}, & j < k \\ \lambda_{kj}^{a}, & j > k \\ \lambda_{j}, & j = k \end{cases} \tag{16}$$

Матрицы Λ_α обладают свойством ортогональности по следу, которое может быть использовано для разложения произвольной эрмитовой матрицы по базису Λ_α:

$$Tr\left(\Lambda_\alpha \Lambda_\beta\right) = \begin{cases} 2\delta_{\alpha\beta}, & \left(\alpha \neq N\right) \wedge \left(\beta \neq N\right) \\ \dfrac{2}{N+1}\delta_{\alpha\beta}, & \left(\alpha = N\right) \vee \left(\beta = N\right) \end{cases} \tag{17}$$

Разложим матрицу плотности ρ по полному базису Λ_α:

$$\rho(t) = \sum_{\alpha=1}^{N^2} \rho_\alpha(t)\Lambda_\alpha \tag{18}$$

Таким образом, задача сводится к расчету динамики вектора для коэффициентов разложения $\left\{\rho_1, \rho_2, ..., \rho_{N^2}\right\}$. Использование базиса матриц Гелл-Манна дает ряд преимуществ, главное из которых заключается в том, что явно учитывается физическая специфика решаемой задачи – автоматически сохраняет эрмитовость искомой матрицы плотности ρ. Кроме того, коэффициенты разложения по базису Λ_α являются действительными.

Как будет показано ниже, аналогичное разложение гамильтониана $\hat{H}$ по базису Λ_k и действительность всех коэффициентов разложения позволяют полностью отказаться от использования комплексных чисел в программной реализации, что в свою очередь приводит к ее упрощению и экономии используемых ресурсов.

Выполнив разложения $\hat{\rho}$, $\hat{H}$ и диссипативного слагаемого уравнения (9) по базису Гелл-Манна Λ_α приводит к следующей системе уравнений для вектора коэффициентов разложения $\{\rho_1, \rho_2, ..., \rho_{N^2}\}$:

$$\frac{\partial \rho_\alpha}{\partial t} = \sum_{\mu,\nu} f_{\alpha\mu\nu} H_\mu \rho_\nu + \sum_{\nu} \Gamma_{\alpha\nu} \rho_\nu, \ 1 \le \alpha, \mu, \nu \le N^2 \tag{19}$$

Здесь структурные константы $f_{\alpha\mu\nu}$ и $\Gamma_{\alpha\nu}$ определяются следующими выражениями

$$f_{\alpha\mu\nu} = \frac{1}{2i} Tr\left(\Lambda_\alpha \left[\Lambda_\mu \Lambda_\nu\right]\right) \tag{20}$$

$$\Gamma_{\alpha\nu} = \frac{\gamma}{4} Tr\left(\Lambda_\nu a^+ \left[\Lambda_\alpha a\right] - a\Lambda_\nu \left[\Lambda_\alpha a^+\right]\right) \tag{21}$$

Проводя элементарные преобразования, можно переписать выражение для структурных констант (20) в виде

$$f_{\alpha\mu\nu} = \mathrm{Im} Tr\left(\Lambda_\alpha \Lambda_\mu \Lambda_\nu\right) \tag{22}$$

2.5 Алгоритм распараллеливания метода разложения по матрицам Гелл-Манна

Основные затраты вычислительных ресурсов в описанном методе

сопряжены с вычислением двойной суммы $\sum_{\mu,\nu}...$ правой части системы (19). Для вычисления каждой из таких сумм требуется $\sim 3N^4$ операций сложения и умножения. Общее число операций необходимых для вычисления двойных сумм, относящихся к различным строкам системы составляет $\sim 3N^6$. Число операций, требуемых для вычисления всех N^2 сумм, связанных с диссипативной частью, на два порядка меньше и составляет $\sim 2N^4$.

Однако благодаря особым свойствам структурных констант $f_{\alpha\mu\nu}$ (22), многие из них оказываются равными нулю, что позволяет исключить из суммирования $\sum_{\mu,\nu}...$ подавляющее большинство слагаемых. К упомянутым выше свойствам структурных констант $f_{\alpha\mu\nu}$ относятся их антисимметричность при перестановке любой пары индексов и действительность их значений (см. (22)). Аналитические расчеты показывают, что при фиксированном значении индекса α число структурных констант $f_{\alpha\mu\nu}$ отличных от нуля меняется в пределах от 0 до $\frac{1}{2}(N-1)(N+3)$ для нечетных N или до $\frac{1}{2}\left(N^2+2N-4\right)$ для четных значений N $(N \geq 8)$. Таким образом, число ненулевых слагаемых в сумме $\sum_{\mu,\nu}...$ является величиной порядка $\sim N^2/2$, и для ее вычислений требуется $\sim 3N^2/2$, что на два порядка меньше, чем требуется при обычном суммировании. Общее число независимых ненулевых структурных констант $f_{\alpha\mu\nu}$ составляет $\frac{1}{6}(N-1)\left(5N^2-4N-6\right)$.

Использование описанных фактов способно увеличить производительность вычислительного процесса и явилось основой для разработанного приложения.

Один из возможных и наиболее простых способов реализации суммирования $\sum_{\mu,\nu}...$ с исключением нулевых слагаемых заключается в том, что в соответствие каждой строке α системы ставится таблица, называемая таблицей индексов умножения, имеющая структуру таблицы 1.

Таблица 1. Вид таблицы индексов умножения

μ	ν	$f_{\alpha\mu\nu}$
μ_1	ν_1	$f_{\alpha\mu_1\nu_1}$
μ_2	ν_2	$f_{\alpha\mu_2\nu_2}$
μ_3	…	…

k–ая строка такой таблицы указывает, что в сумму должно войти слагаемое $f_{\alpha\mu_k\nu_k}H_{\mu_k}\rho_{\nu_k}$. Построение таблиц индексов умножения происходит в момент запуска приложения. Для вычисления значений $f_{\alpha\mu\nu}$ используется алгоритм, применяющий аналитические выражения и не требующий работы с матрицами, что делает процесс инициализация таблиц индексов умножения очень быстрым. Сами таблицы индексов умножения исполнены в виде массива структур MultiplicationIndex, описание которых приводится в листинге 1.

Листинг. 1. Структура MultiplicationIndex

```
__global__ struct MultiplicationIndex
{
```

```
        int x, y;
        REAL v;
    };
```

Поля x, y и v играют соответственно роль μ, ν и $f_{\alpha\mu\nu}$. Ввиду антисимметричных свойств структурных констант $f_{\alpha\mu\nu}$ достаточно хранить лишь половины таблиц индексов таблиц умножения.

Аналогичные рассуждения можно произвести и для суммы $\sum_{\nu}$.... Каждая таблица индексов умножения теперь состоять из двух столбцов и содержать одну или две записи. Таким образом, количество операций, необходимых для вычисления одной суммы, снижается на два порядка, с $\sim 2N^2$ до ~ 2. Индексы умножения суммы $\sum_{\nu}$... также можно сохранить в таблицу 1. Так как в этом случае требуется задействовать один индекс ν, то значения индекса μ выставляем равным -1, что позволит во время счета определять, к какой из двух сумм относится слагаемое.

Дополнительное ускорение приложения достигается путем распараллеливания процесса вычисления групп строк правой части системы на графических процессорах. Количество строк в группе, должно выбираться с учетом технических характеристик графического процессора (его тактовой частоты, объема доступной памяти, максимального числа thread-ов). На стадии инициализации приложения определяется число графических процессоров в системе, их характеристики, и принимается решение о размерах групп, после чего, в соответствии с принятым решением, им рассылаются соответствующие таблицы индексов умножения. При

вычислении на GPU одной строке группы в соответствие ставится один thread GPU.

Реализация всех перечисленных выше идей для вычисления правых частей на GPU представлена в листинге 2.

Листинг 2. Вычисление группы строк на GPU

```
__constant__ REAL s_rGamma2;                    // Коэффициент диссипации
__constant__ int s_nLinesCount;                 // Число вычисляемых на GPU строк
__constant__ REAL *s_vHamiltonian;              // Гамильтониан системы

__global__ void CalculateRightPart(const REAL *d_vDensity0, REAL *d_vDensity, const REAL *d_vDensityArg, REAL *d_vDensityArgTemp, REAL h1, REAL h2)
{
    int n = blockIdx.x * blockDim.x + threadIdx.x;

    if (n < s_nLinesCount)
    {
        REAL rValue = 0;

        // Массив индексов умножения
```

```
            Index *pIndices = s_aMultiplicationIndices[n];
            Index *pIndicesNext = s_aMultiplicationIndices[n + 1];

            // Диссипативная часть
            while (pIndices->x < 0)
            {
                rValue += pIndices->v * d_vDensityArg[pIndices->y];

                ++pIndices;
            }

            rValue = rValue * s_rGamma2;

            // Двойная сумма
            while (pIndices < pIndicesNext)
            {
                rValue += pIndices->v * s_vHamiltonian[pIndices->x] *
d_vDensityArg[pIndices->y];
                rValue -= pIndices->v * s_vHamiltonian[pIndices->y] *
d_vDensityArg[pIndices->x];
```

```
                ++pIndices;
            }

            ...
        }
    }
```

Функция CalculateRightPart образована двумя циклами, вычисляющими суммы $\sum_{\mu,\nu}...$ и $\sum_{\nu}...$. Признаком второй суммы является отрицательный индекс x. Используется соответствующая строке n таблица индексов умножения aMultiplicationIndices[n], построенная при инициализации приложения.

Процесс инициализации приложения условно может быть разделен на три основных этапа:

- создание субкоммуникаторов, объединяющих процессы, исполняемые на одном узле;
- выбор каждым процессом видеокарты, используемой для вычислений;
- разделение нагрузки между видеокартами.

Для объединения в группы процессов, запущенных на одном узле, используется объект коммуникатора и функция MPI MPI_Comm_split, обеспечивающая быстрый и простой способ одновременного создания

нескольких коммуникаторов. Алгоритм формирования групп может быть описан следующим образом:

1. все процессы, кроме процесса Master (имеющего ранг 0 в коммуникаторе MPI_COMM_WORLD), “окрашиваются” в черный цвет. Используя “окраску” процессов и функцию MPI_Comm_split, создаем коммуникаторы;

2. все “неокрашенные” процессы продолжают выполнение цикла 2 – 5;

3. “неокрашенный” процесс с рангом 0, используя свой текущий коммуникатор, рассылает всем участникам группы имя узла и “окрашивается”;

4. Члены группы принимают имя узла. Если принятое имя совпадает с именем узла приемника, то процесс “окрашивается”;

5. Используя “окраску” процессов и функцию MPI_Comm_split, создаем коммуникаторы. Переходим к шагу 2.

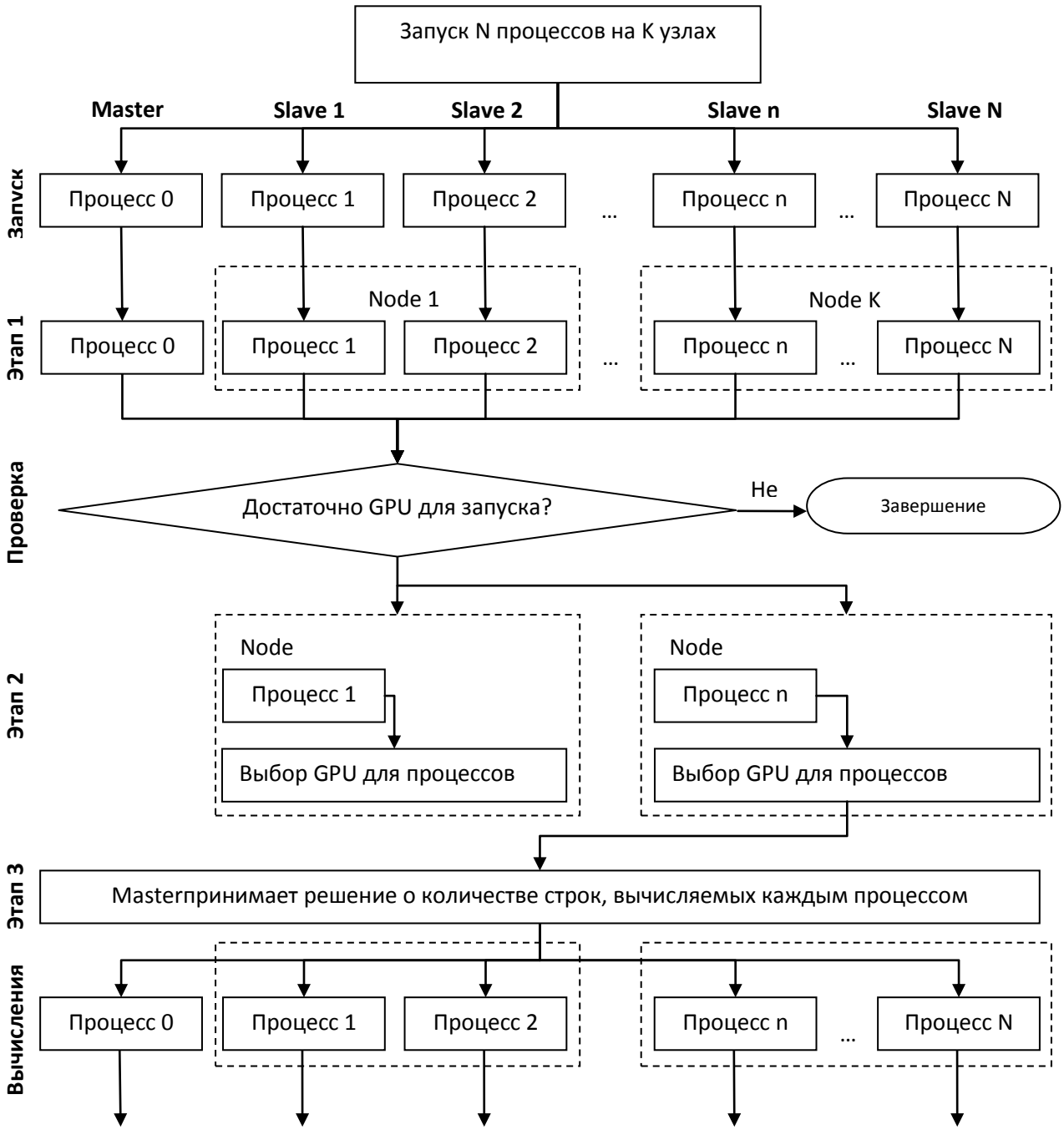


Рис. 2. Схема процесса инициализации приложения для метода разложения по матрицам Гелл-Манна

Листинг 3 содержит фрагмент исходного кода программы, реализующего описанный алгоритм формирования субкоммуникаторов.

Листинг 3. Создание субкоммуникаторов

```
// g_nProcRank – ранг процесса в MPI_COMM_WORLD
MPI_Comm commNode, commNodeTemp;

// "Цвет" процесса
int nColor = !g_nProcRank;
MPI_SAFE_CALL(MPI_Comm_split(MPI_COMM_WORLD, nColor, g_nProcRank, &commNode));

// Обрабатываем "неокрашенные" процессы
while(!nColor)
{
 char sNodeNameBuf[MPI_MAX_PROCESSOR_NAME];

 MPI_SAFE_CALL(MPI_Comm_rank(commNode, &nNodeProcRank));

 // node master
 if (nNodeProcRank == 0)
 {
        strcpy(sNodeNameBuf, sNodeName);
 }

 MPI_SAFE_CALL(MPI_Bcast(sNodeNameBuf, MPI_MAX_PROCESSOR_NAME, MPI_CHAR, 0, commNode));
```

```
  // Принимаем решение об "окрашивании"
  nColor = !strcmp(sNodeNameBuf, sNodeName);
  MPI_SAFE_CALL(MPI_Comm_split(commNode, nColor, g_nProcRank,
&commNodeTemp));

  MPI_SAFE_CALL(MPI_Comm_free(&commNode));
  MPI_SAFE_CALL(MPI_Comm_dup(commNodeTemp, &commNode));
  MPI_SAFE_CALL(MPI_Comm_free(&commNodeTemp));
 }
```

На втором этапе инициализации приложения каждый процесс с рангом 0 в субкоммуникаторе узла (node master) собирает информацию об установленных на узле видеокартах. Для этого используются стандартные функции CUDA cudaGetDeviceCount и cudaGetDeviceProperties. На основе значения поля clockRate структуры cudaDeviceProp выбирается необходимое число самых высокопроизводительных видеокарт и осуществляется их привязка к процессам. Соответствующий фрагмент исходного кода содержится в листинге 4.

Листинг 4. Привязка процессов к видеокартам

```
// nNodeProcRank - ранг процесса в коммуникаторе узла commNode

// Устройства узла (сортированные по величине тактовой частоты)
int *devices = (int*)malloc(nNodeDeviceCount * sizeof(int));
int *rates = (int*)malloc(nNodeDeviceCount * sizeof(int));
```

```
if (nNodeProcRank == 0)
{
  for (int i = 0; i < nNodeDeviceCount; i++)
  {
        cudaDeviceProp prop;
        CUDA_SAFE_CALL(cudaGetDeviceProperties(&prop, i));

        int j;
        for (j = i - 1; j >= 0 && rates[j] < prop.clockRate; j--)
        {
              devices[j + 1] = devices[j];
              rates[j + 1] = rates[j];
        }

        devices[j + 1] = i;
        rates[j + 1] = prop.clockRate;
  }
}

MPI_SAFE_CALL(MPI_Bcast(devices, nNodeDeviceCount, MPI_INT, 0,
commNode));
MPI_SAFE_CALL(MPI_Bcast(rates, nNodeDeviceCount, MPI_INT, 0,
commNode));
```

```
// Выбираем устройства
CUDA_SAFE_CALL(cudaSetDevice(devices[nNodeProcRank]));
```

На третьем этапе инициализации процесс Master принимает решение о разделении строк системы между процессами для обеспечения равномерной загрузки системы. Простейшее решение этой задачи может быть получено в предположении, что вычисление каждой строки требует одинакового количества операций, а быстродействие видеокарты определяется ее тактовой частотой. В таком случае, для обеспечения равномерной загрузки, число строк, вычисляемых на i-ой видеокарте, должно быть пропорционально ее тактовой частоте и будет равняться $\approx N^2 \cdot clockRate[i] \Big/ \sum_k clockRate[k]$.

2.6 Метод квантовых траекторий (квантовый метод Монте-Карло)

Суть метода квантовых траекторий или, как его ещё часто называют в литературе, квантовый метода Монте-Карло (МК) [17], состоит в том, чтобы свести задачу к решению *j*-го числа уравнений для диссипативной волновой функции системы содержащее *N* переменных (в нашей задаче число уровней осциллятора) и описывающих динамику системы в единичной реализации, то есть $\rho^j(t) = \left|\Psi^j(t)\right\rangle\left\langle\Psi^j(t)\right|$. Для сопоставления полученного решения методом МК с результатом полученного на основе уравнения для матрицы плотности (методом "растягивания" матрицы в вектор или с помощью разложения по матрицам Гелл-Манна), необходимо провести усреднение по всем *j* квантовым траекториям.

Продемонстрируем применимость метода МК для конкретной задачи - расчета диссипативной динамики нелинейного осциллятора. Во-первых, проведем конечно-разностную аппроксимацию уравнения для ρ

$$\rho(t+\Delta t) = U\rho(t)U^{+} + \Delta t\gamma\hat{a}\rho(t)\hat{a}^{+}, \qquad (23)$$

где $U = e^{-iH_{eff}\Delta t}$, а $H_{eff} = H - \frac{i\hbar\gamma}{2}\hat{a}^{+}\hat{a}$. При учёте того, что резервуар для осциллятора бозонный термостат (6) и оператор шума (в форме Линдблада) для осциллятора - это оператор уничтожения $\hat{a}$ [16]. Первое слагаемое в выражении (23) описывает диссипативную динамику системы с эффективным гамильтонианом Вигнера–Вайскопфа H_{eff}, а второе слагаемое - квантовые скачки. Попытаемся придать последнему слагаемому вероятностный смысл, имея в виду, что с уравнением (23) можно связать некоторый случайный процесс, которые будет имитировать относительный вклад диссипативной динамики.

Рассмотрим некоторую реализацию чистого состояния $\rho^{j}(t) = \left|\Psi^{j}(t)\right\rangle\left\langle\Psi^{j}(t)\right|$, а волновую функцию системы разложим по фоковскому базису $\left|\Psi^{j}(t)\right\rangle = \sum_{n} C_{n}^{j}(t)\left|n\right\rangle$, тогда будем иметь уравнения для коэффициентов $C_{n}^{j}(t)$, где индекс j соответствует различным квантовым траекториям.

Анализируя уравнение (23) можно увидеть, что изменение матрицы плотности обусловлено двумя возможными вкладами

1) либо с вероятностью $P^{j}(t)$ изменение квантовой траектории сопровождается скачком волновой функции и перенормировкой коэффициентов разложения $C_{n}^{j}(t)$, что соответствует действию оператора шума в системе (оператора уничтожения $\hat{a}$):

$$C_n^j(t+\Delta t)=\frac{\sqrt{n}C_{n-1}^j(t)}{\sqrt{\sum_n n\left|C_n^j(t)\right|^2}} \tag{24}$$

Вероятность квантовых скачков определяется следующим выражением:

$$P^j(t)=\gamma\sum_n n\left|C_n^j(t)\right|^2\Delta t \tag{25}$$

2) либо с вероятностью $1-P^j(t)$ согласно

$$\frac{\partial}{\partial t}C_n^j(t)=-\frac{i}{\hbar}H_{eff}C_n^j(t) \tag{26}$$

происходит изменение коэффициентов волновой функции (“траектории”) за счет диссипативной динамики.. В этом случае интервал времени Δt разбивается на ещё более мелкие шаги по времени и дифференциальное уравнение (24) решается стандартным метом Рунге-Кутта четвертого порядка.

Считая, что время дискретно и меняется с интервалом Δt, то численный алгоритм для вычисления одной j-ой квантовой траектории, представляющую одну реализацию эксперимента – диссипативную динамику нелинейного квантового осциллятора, можно представить на схеме рис.8.

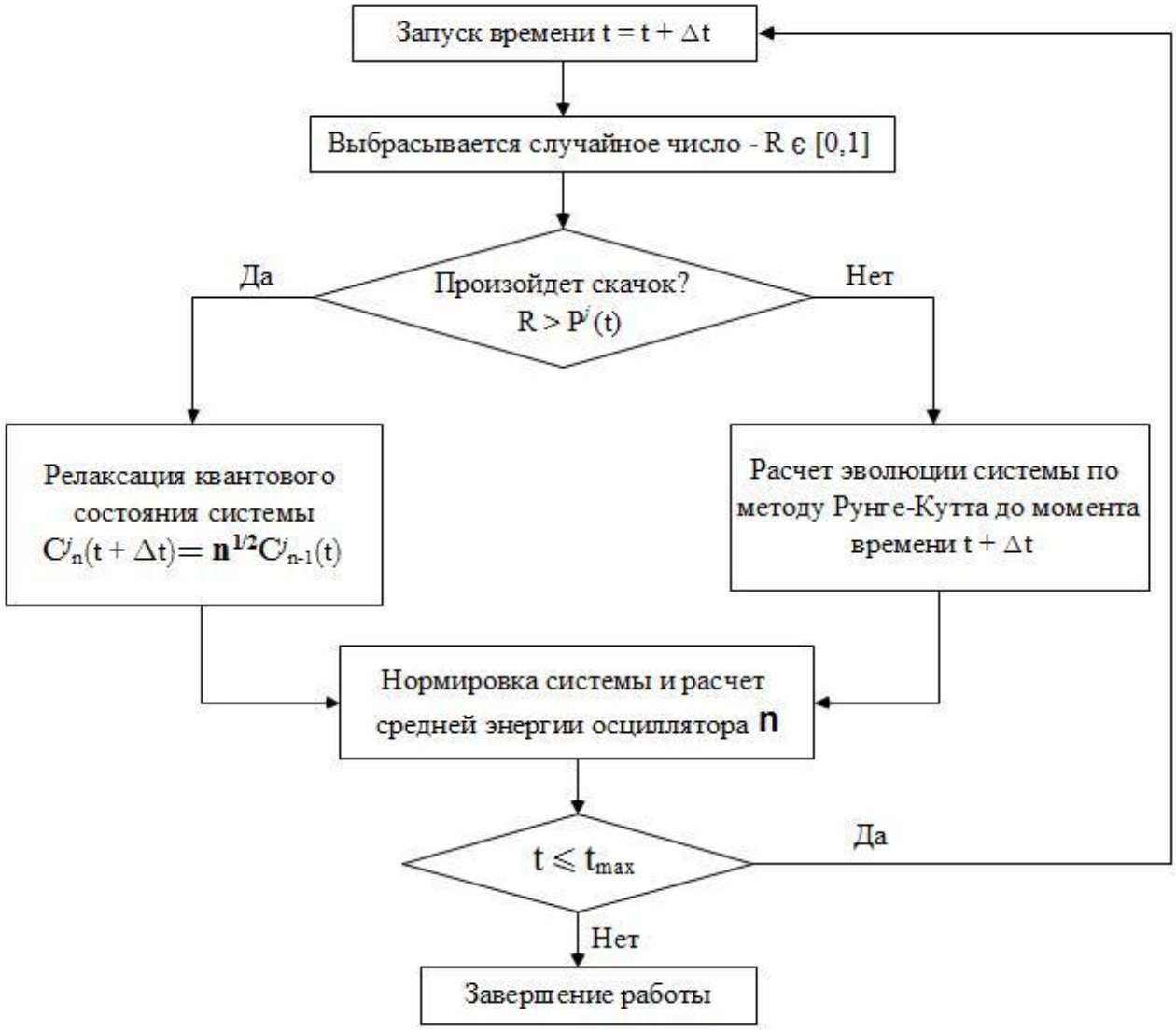


Рис. 8 Алгоритм моделирования диссипативной динамики потокового кубита, основанного на квантовом методе Монте-Карло.

Учитывая, что процесс релаксации является случайным, каждая траектория уникальна. Полученное решение необходимо усреднить по M реализациям, обычно M порядка 1000-10000, для достижения точности квантового МК, определяемой $1/\sqrt{M}$

$$\rho(t) = \frac{\sum_{j=1}^{M} \left|\Psi^j(t)\right\rangle\left\langle\Psi^j(t)\right|}{M} = \frac{\sum_{j=1}^{M}\sum_{n=0}^{N}\left|C_n^j(t)\right|^2}{M} . \qquad (27)$$

В качестве примера в данной работе рассчитывается среднее число

квантов осциллятора (средняя энергия):

$$\overline{n} = \frac{\sum_{j=1}^{M}\sum_{n=0}^{N} n\left|C_n^j(t)\right|^2}{M}. \qquad (28)$$

Реализации статистически независимы, поэтому есть возможность запускать каждую реализацию независимо в отдельном потоке (на отдельном процессоре), собирая затем данные и усредняя. Отсутствует необходимость в обмене данными между потоками в процессе работы программы, что позволяет использовать многопроцессорные системы, особенно системы на базе GPU-ускорителей практически с максимальной эффективностью.

2.7 Схема распараллеливания квантового метода Монте-Карло

Как сказано выше, расчет физических величин может быть организовано на GPU устройстве с применением технологии CUDA [1,5]. Поскольку реализации статистически независимы, отсутствует необходимость в обмене данными между блоками и потоками внутри отдельных блоков, создаваемых на GPU, на каждом процессоре выполняется одна и та же подпрограмма, что позволяет обеспечить параллелизм на уровне данных (SIMT).

Однако так как для достижения точности метода МК в 1% необходимо произвести расчет 10000 квантовых траекторий эволюции системы. Поэтому для большей производительности программы эффективно использовать в расчетах Кластерную систему, узлы которой содержат GPU. Для обеспечения работы нескольких GPU была использована технология программирования для систем с распределенной памятью MPI [19, 20].

Для использования всех имеющихся графических ускорителей работа с данными организована следующим образом (см. схему на рис.9):

- на хосте (управляющем компьютере) запускается основной модуль программы, который инициализирует данные о параметрах системы,

вычисляет начальное состояние, а также создает структуры необходимых библиотек;

- считываются данные о количестве реализаций (квантовых траекториях) и делается несколько копий структуры с информацией о гамильтониане, которые будут использоваться на отдельных GPU, причем данные разбиваются на пачки (для каждого GPU) таким образом, чтобы на один блок приходилось 256 потоков, а количество блоков CUDA определяет самостоятельно запрашивая необходимые данные о встроенных GPU-ускорителях;
- с помощью функций MPI с исполняемого хоста происходит рассылка информации на каждое GPU устройство, сколько квантовых траекторий будет рассчитываться;
- инициализируется ядро вычислений на каждом из GPU и запускается алгоритм работы МК (см. схему на рис. 8), промежуточные результаты накапливаются в памяти каждого из используемых графических адаптеров;
- после завершения расчетов всех траекторий на отдельной GPU происходит усреднение данных (расчет средней энергии осциллятора на 1 GPU устройстве);
- с помощью функций MPI происходит сбор информации на хост о получившихся средних значениях на каждой из работающих GPU;
- на хосте происходит усреднение данных по всем GPU устройствам и запись в файл необходимых сведений о расчетах.

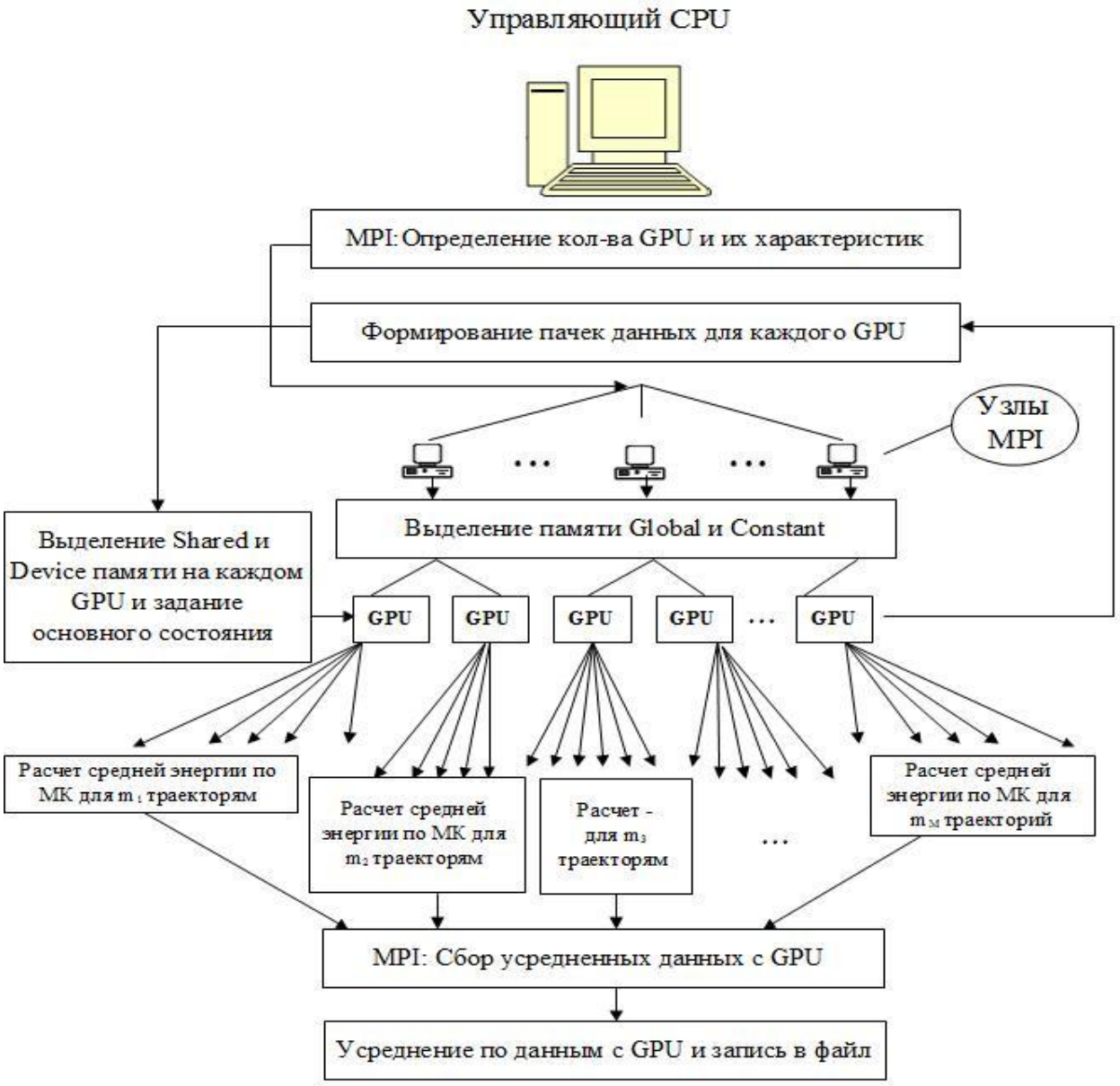


Рис. 9 Алгоритм вычисления средней энергии осциллятора на нескольких GPU, где блоки «Расчет средней энергии МК» реализуются согласно алгоритму, приведенному на рис. 8.

Одним из важных условий эффективности работы программы на GPU является оптимальное использование имеющихся типов памяти: глобальной, разделяемой и памяти констант. В частности для решения данной задачи необходимы следующие наборы данных: информация о гамильтониане (хранятся в регистрах разделяемой памяти), константы (хранятся в кешируемой памяти констант), текущие населённости уровней (хранятся в

регистрах разделяемой памяти) и результирующие данные (имеют большой объем, хранятся в глобальной памяти). Регистры являются быстрым типом памяти, но имеют небольшой объем, поэтому использование их для работы с результирующими данными не представляется возможным. Глобальная память, напротив, позволяет хранить большие объемы данных, но является относительно медленной. Однако в данной задаче обращения к результирующим данным массивов, находящимся в глобальной памяти, немного, поэтому хранение в глобальной памяти не оказывает существенного влияния на производительность программы. А также отметим, что если при усовершенствовании архитектуры GPU объем разделяемой памяти будет увеличен, то существует возможность хранить результирующие данные в данной памяти GPU, что позволит ещё более эффективно использовать обращение к памяти и ускорить расчет.

Листинг 5. Исполняемая функция на GPU для метода МК

```
void calculate_gpu(void * initialC, void * nr_m, void * nr1_m, void *
nr2_m,
                        void * qp1_m, void * qp2_m)
{
    cudaDeviceProp props; // определение информации о GPU
    cudaGetDeviceProperties(&props, 0);
     int numBlocks = props.multiProcessorCount;
     int numThreads = (hostKmax + numBlocks - 1) / numBlocks; //
распределение данных
     if (numThreads > 256)                          // по блокам
          numThreads = 256;

     complex * d_initialC;      // выделение памяти
     cudaMalloc(&d_initialC, 2 * Nm * sizeof(complex));
     cudaMemcpy(d_initialC, initialC, 2 * Nm * sizeof(complex),
cudaMemcpyHostToDevice);
```

```
	real * hostZero = new real[hostTmax];
	for (int i = 0; i < hostTmax; ++i)
		hostZero[i] = 0;
	real *d_nr_m;
	cudaMalloc(&d_nr_m, hostTmax * sizeof(real));
	cudaMemcpy(d_nr_m, hostZero, hostTmax * sizeof(real),
cudaMemcpyHostToDevice);

	delete [] hostZero;

	calculate_kernel<<< numBlocks, numThreads >>>(d_initialC,
d_nr_m)// запуск ядра
	cudaDeviceSynchronize();
	cudaMemcpy(nr_m, d_nr_m, hostTmax * sizeof(real),
cudaMemcpyDeviceToHost);

	cudaFree(d_initialC);
	cudaFree(d_nr_m);

	cudaFree(device_Ed);
	cudaFree(device_h);
	cudaFree(device_q);
	cudaFree(device_r);
	cudaFree(device_inter);
}
```

Заметим, что точность квантового метода МК зависит от числа реализаций $\sim 1/\sqrt{M}$, где M – число реализаций. В силу наличия стохастических процессов двойная точность для расчётов не требуется, а использование операций с одинарной точностью на GPU, существенно ускоряет работу приложения, т.к. вычисления с двойной точностью на

несколько порядков медленнее. Для обеспечения сходимости результатов моделирования диссипативной динамики необходимо наличие большого числа реализаций ($M > 10^3$) и генерирования последовательностей случайных чисел непосредственно для каждой точки временной траектории. Генерация необходимого массива случайных чисел производилась на GPU с помощью возможностей CUDA библиотеки – CURAND. Использование данной библиотеки существенно позволило увеличить скорость выполнения расчётов, так как скорость генерации случайных чисел на GPU в 50 раз больше, чем на CPU.

В реализации программы также была использована готовая библиотека cuda_complex.hpp.

3 Вычислительная установка (схема кластера НИФТИ ННГУ)

Методическая работа базируется на оборудовании, закупленном в рамках программы развития ННГУ как национального исследовательского университета – вычислительного кластера НИФТИ ННГУ. Кластер состоит из девяти рабочих станций (см. схему на рис. 10), восемь из которых содержат графические ускорители nVidia® Tesla.

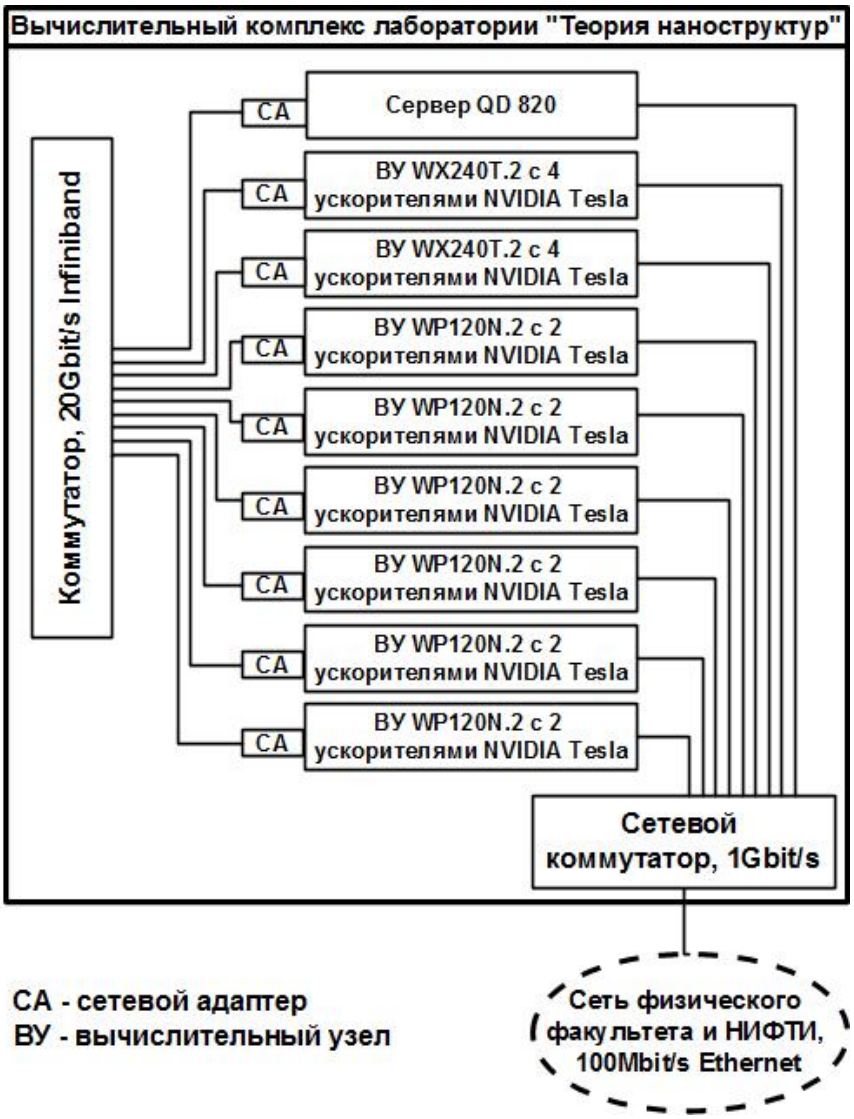


Рис. 10 Схема кластера лаборатории «Теория наноструктур» НИФТИ ННГУ им. Лобачевского.

Краткое техническое описание оборудования и сети:

1) **Сервер Flagman QD820 (**8 процессоров AMD® Opteron™ SixCore, 16 x DIMM 4096Mb DDR-II, 4 x HDD 300Gb SerialATA 10000rpm**);**

2) процессоров AMD® Opteron™ SixCore, 16 x DIMM 4096Mb DDR-II, 4 x HDD 300Gb SerialATA 10000rpm**);**

3) **2 станции Flagman WX240T.2 (**2 процессора Intel® Xeon® X5550, 12 x DIMM 2048Mb DDR-III, 4 вычислителя nVidia® Tesla® C1060 4096Mb DDR-III**);**

4) **6 станций Flagman WP120N.2 (**процессор Intel® Core™ i7 I7-950, 6 x 2048Mb DDR-III, 2 вычислителя nVidia® Tesla® C1060 4096Mb DDR-III**).**

Сетевое оборудование: Вычислительная сеть: 20 Gbit/s InfiniBand (коммутратор Mellanox MTS3600); Служебная сеть: 1 Gbit/s (коммутатор 3Com, 24 ports).

Для удобства доступа к кластеру организован удаленный доступ с помощью стандартной программы ОС Windows – ***Удаленный рабочий стол***, из аудиторий физического факультета ННГУ им. Н.И. Лобачевского №419 (лаборатория «Теория наноструктур») и на №537 (терминал класс кафедры теоретической физики, физического факультета).

Для выполнения лабораторной работы в оборудованном терминал классе (аудитория №537), необходимо произвести следующие действия:

Пуск → Все программы → Стандартные → Подключение к удаленному рабочему столу

или

C:\Windows\system32\mstsc.exe

Для подключения к выбранному компьютеру укажите IP-адрес и порт:

Для подключения к Lab25-6 – **85.143.6.98:3389**

Для подключения к Lab25-7 – **85.143.6.98:3390**

Для подключения к Lab25-8 – **85.143.6.98:3391**

Введите имя учетной записи и пароль:

Имя учетной записи: **cluster\Theorlab**

Пароль: **Theorlab**

Для завершения работы просто закройте окно Удаленного рабочего стола.

4 Результаты расчетов и полученное ускорение

В данной методической работе разработаны два параллельных алгоритма решения уравнения для матрицы плотности, один из алгоритмов основан на разложении уравнения по матрицам Гелл-Манна (см. раздел 2.5) и другой алгоритм на основе квантового метода Монте-Карло (см. раздел 2.7). Результатом работы стали приложения, реализованные на языке CUDA C с использованием пакета CUDA Toolkit 6.5 и библиотек MPI. Тестирование производилось на кластере НИФТИ ННГУ (см. раздел 3).

Для проведения численных экспериментов по вычислению среднего числа фотонов нелинейного джозефсоновского осциллятора. Рассмотрим, как ведет себя среднее число фотонов в каждом акте измерения и в среднем по реализациям. Пусть осциллятор в начальный момент времени был заселен на уровень $n_0 = 15$ тогда, если внешнего возбуждения нет, т.е. $f_0 = 0$, среднее число фотонов с течением времени уменьшается экспоненциально, что видно на рис. 11, где черная кривая, полученная решением уравнения для матрицы плотности методом разложения по матрицам Гелл-Манна, где в качестве метода решения дифференциального уравнения (19) была выбрана классическая схема Рунге-Кутта четвертого порядка, хорошо подходящая для систем с диссипацией. Однако, полеченная усредненная кривая (черная кривая на рис. 11) описывает лишь поведение системы в среднем, в каждой реализации моменты испускания кванта энергии (фотона) случайны, что было промоделировано на основе квантового метода МК (например, единичная траектория - синяя кривая на рис. 11). Отметим, что при

усреднении по M = 10000 квантовым траекториям наблюдается полное совпадение зависимостей, посчитанных методом на основе матриц Гелл-Манна и квантовым методом МК (черная кривая на рис. 11). В случае квантового осциллятора данная черная кривая полностью совпадает с экспоненциальной зависимостью убывания энергии. При расчетных параметрах (γ = 0,005) система должна затухнуть на времени t = 200T = 1256 (ед), что и подтверждается численным расчетом.

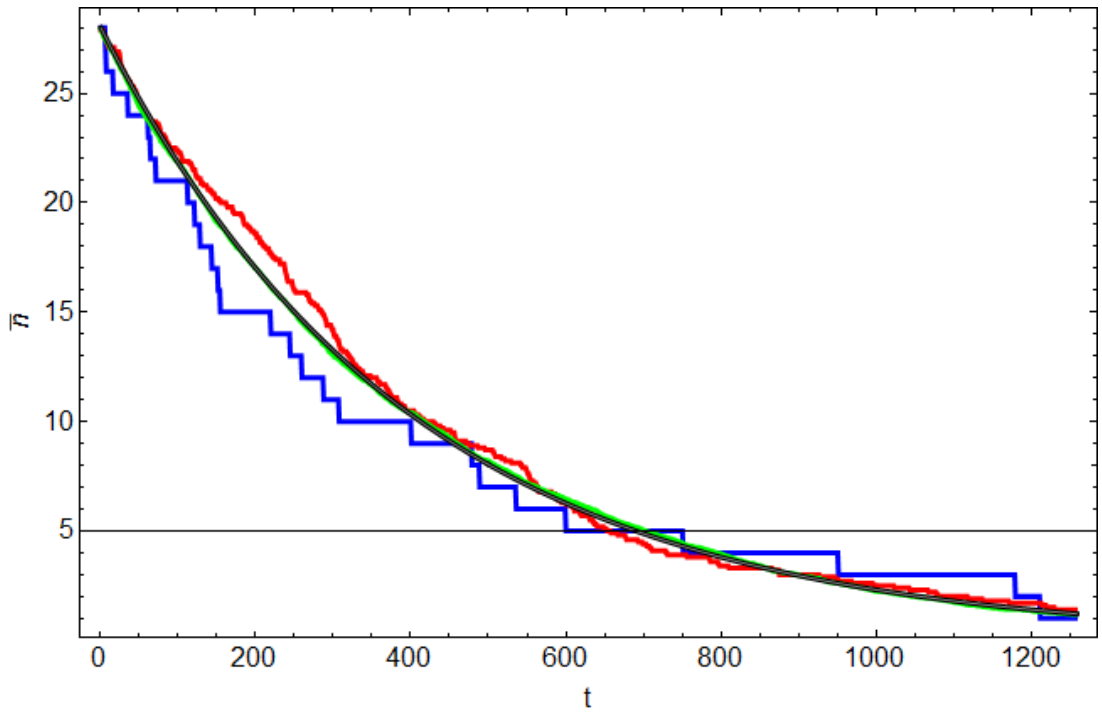


Рис. 11. Зависимость среднего числа фотонов $\bar{n}$ нелинейного осциллятора от времени для разного числа реализаций: синяя кривая M = 1, красная - M = 100, зеленая - M = 1000, черная - M = 10000. Параметры системы: $\omega_0 = 1$, $f_0 = 0$, $\mu = 0.1$, $\gamma = 0.005$.

Далее производилось моделирование с возбуждающей силой на интервале времени $0 \le t \le 50$ с шагом 0.001. Параметры системы для гамильтониана (5): ω_0 = 1.02, μ = 0.1, f_0 = 0.2, γ = 0.1.

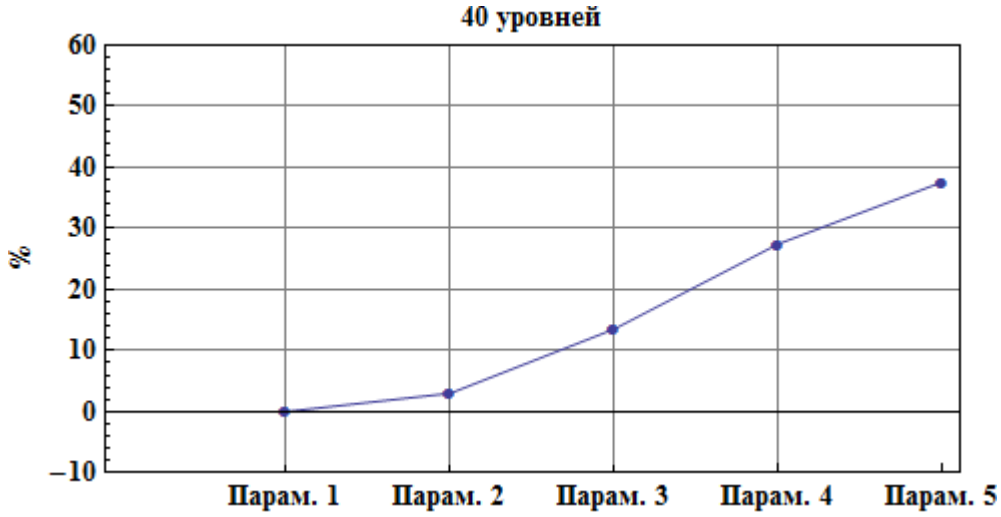
В начале приведем результаты измерения ускорения работы приложения, основанном на решении уравнении матрицы плотности методом

разложения по матрицам Гелл-Манна, при различных значениях числа уровней квантовой системы и различных конфигурациях запуска Описание параметров запуска приведено в таблице 4, а эффективность работы представлена на графиках рис. 12 и рис. 13.

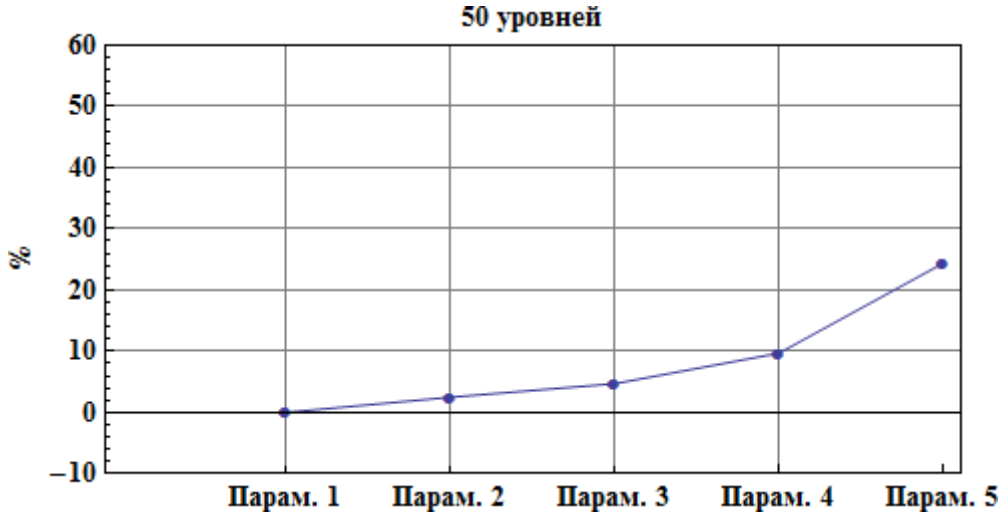
Таблица 4. Параметры запуска

	Описание
Параметры 1	1 узел, 1 видео карта
Параметры 2	1 узел, 2 видео карты
Параметры 3	2 узла, 2 видео карты на каждом узле. Всего 4 видео карты
Параметры 4	3 узла, 2 видео карты на каждом узле. Всего 6 видео карт
Параметры 5	4 узла, 2 видео карты на трех узлах и 4 видео карты на одном узле. Всего 10 видеокарт

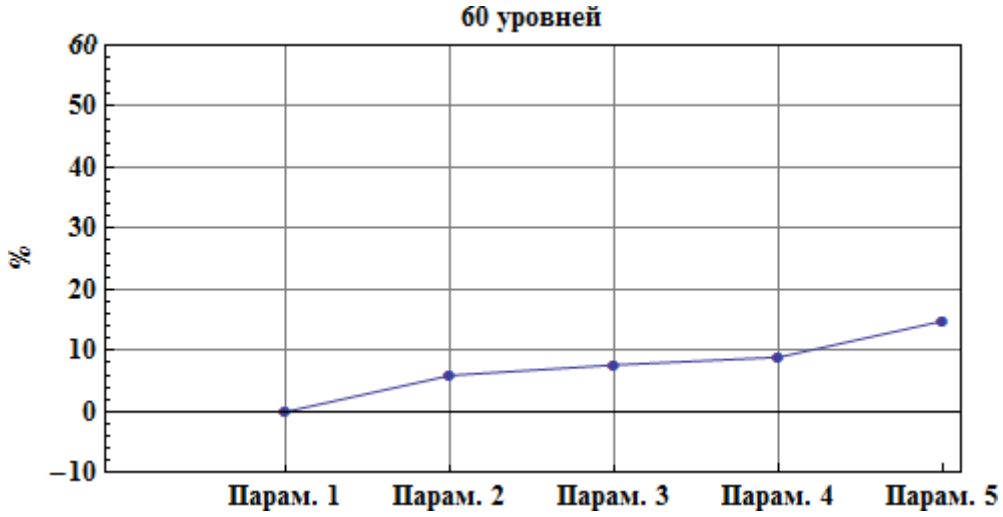
а



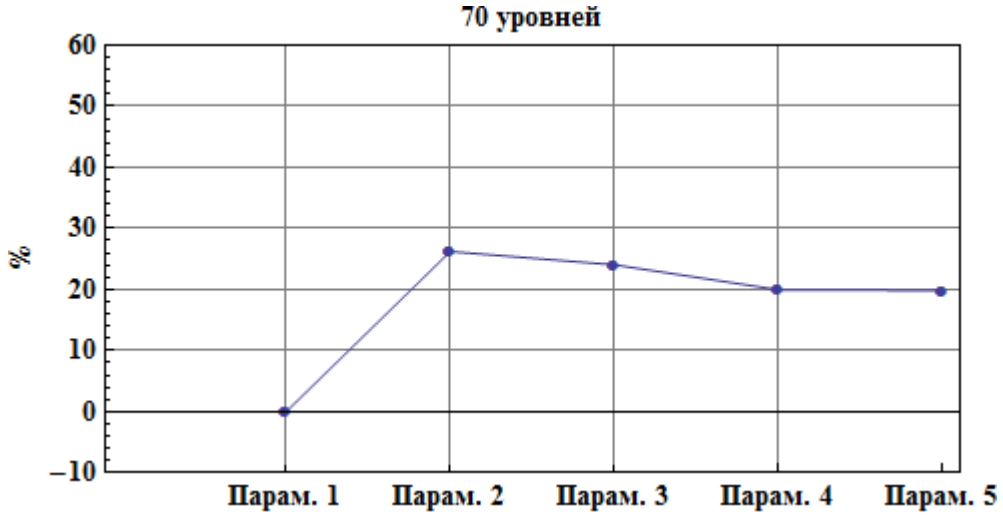
б


50 уровней
60
50
40
30
20
10
0
–10
%
Парам. 1
Парам. 2
Парам. 3
Парам. 4
Парам. 5

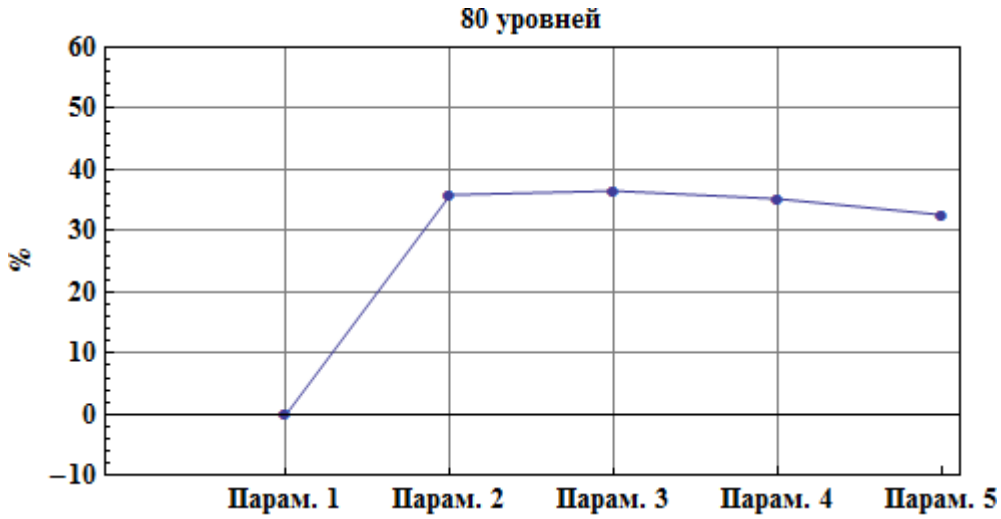
в


60 уровней
60
50
40
30
20
10
0
–10
%
Парам. 1
Парам. 2
Парам. 3
Парам. 4
Парам. 5

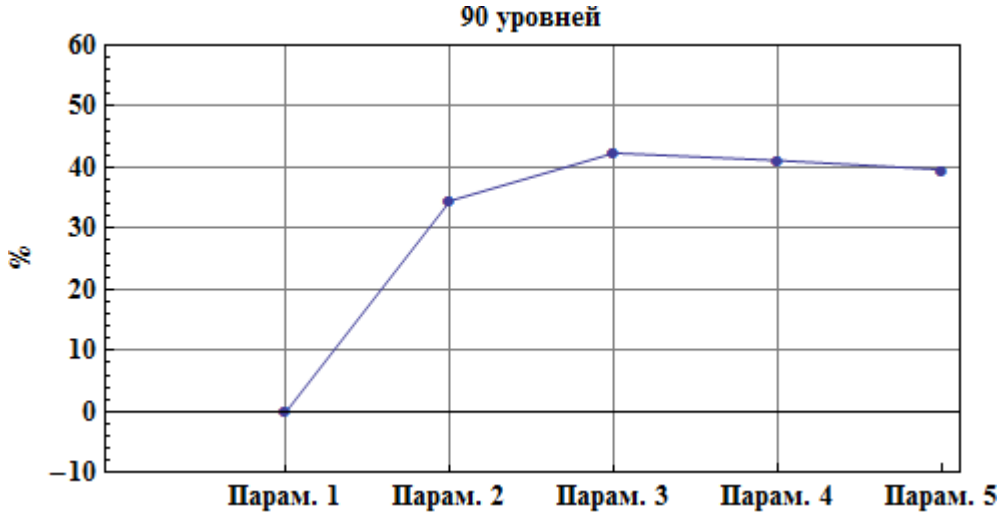
г


70 уровней
60
50
40
30
20
10
0
–10
%
Парам. 1
Парам. 2
Парам. 3
Парам. 4
Парам. 5

д



е



ж

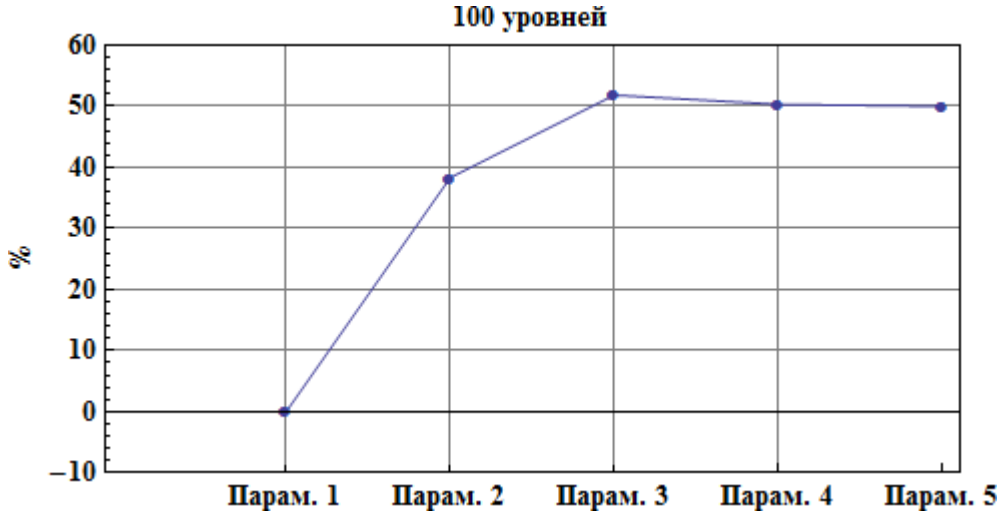


Рис. 12. Зависимость ускорения приложения от конфигурации при различном числе уровней квантовой системы

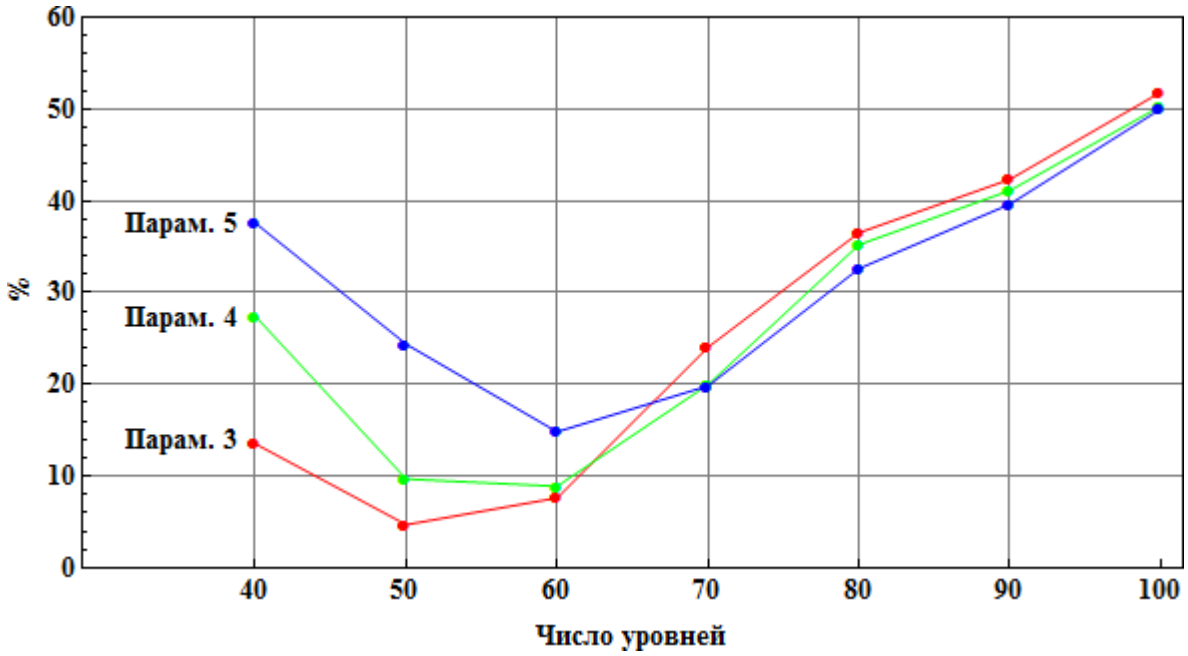


Рис. 13. Зависимость времени работы приложения от числа уровней квантовой системы при различных параметрах запуска

Из графиков, представленных на рисунке 13, видно, что при больших значениях числа уровней квантовой системы наблюдается рост ускорения. Однако наличие максимумов на графиках 12.г, 12.д, 12.е и 12.ж свидетельствует о том, что ускорение зависит отнюдь не только от числа используемых узлов и видеокарт, но и от количества передаваемых данных. При данной реализации передача большого количества данных приводит к уменьшению потенциального прироста производительности. Одним из способов оптимизации может стать использование большего числа ассинхронных вызовов отправки и приема данных, что в свою очередь потребует изменения используемого классического метода Рунге-Кутты, принадлежащего к классу последовательных методов. Вопрос о распараллеливании метода Рунге-Кутты неоднократно рассматривался во множестве различных работ, например в [21-23].

Далее диссипативная динамика нелинейного осциллятора с теми же параметрами системы была рассчитана на основе метода МК. Показано, что рост ускорения достигается за счет увеличении числа квантовых траекторий. Результаты измерения времени работы приложения при 40 уровнях осциллятора в зависимости от усреднения по различному числу квантовых траекторий M приведены на рис. 14. Описание параметров запуска приведено в таблице 5.

Таблица 5. Параметры запуска для метода МК

Кол-во GPU	Число квантовых траекторий (M)			
	$M = 1$	$M = 100$	$M = 1000$	$M = 10000$
1	94.50	120,8	150,91	786,156
2	94,625	110,707	113,758	480,123
4	94,565	108,065	109,466	360,70
6	94,559	107,478	108,95	243,613
8	94,692	106,293	108,39	123,625
10	95,13	106,50	107,86	126,965

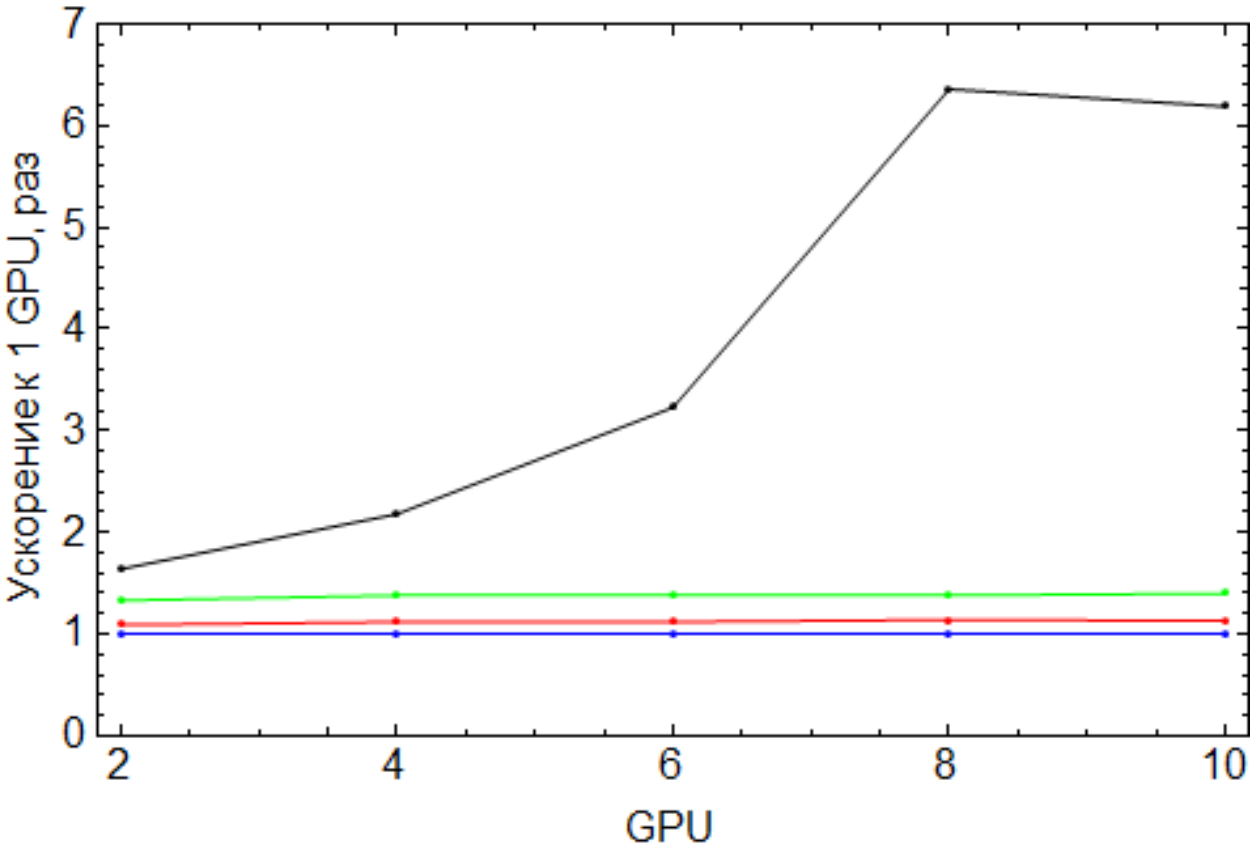


Рис. 14 Ускорение работы программы на основе метода МК в зависимости от числа использованных GPU по отношению к одному GPU устройству.Синня кривая для $M = 1$, красная - $M = 100$, зеленая - $M = 1000$, черная - $M = 10000$.

Из графика, представленного на рис. 14, видно, что при увеличении числа реализаций (траекторий M в квантовом методе Монте-Карло) наблюдается существенный выйгрыш в ускореении при использовании GPU устройств. Для малого числа реализаций, т.е. если число квантовых траекторий меньше числа потоков на видиокарте, ускорения при использовании одновременно нескольких GPU ускорителей получить не удается (см. на рис. 13 синяя, красная и зеленая кривая). Это связано с тем, что рассылаемое число заданий для вычислений меньше, чем количество потоков даже на одной видиокарте, не говоря уже о нескольких. Для эффективной работы на каждый GPU должно приходиться не менее 1024 путей (и желательно, чтобы количество путей было кратно 1024), т.е.

например при M = 4000 достаточно использовать 4 процесса (GPU), ускорения от использования большего количества процессов не будет. Это обусловлено тем, что время расчета одной траектории мало, а при «перегрузке» графического ускорителя наступает эффективная загруженность и достигается оптимальная работа. Прирост производительности вычислений в зависимости от количества используемых графических ускорителей для квантового метода Монте-Карло при условии загруженности (число квантовых траекторий больше, чем число доступных потоков на видиокартах), это говорит о хорошем масштабировании задачи при переходе к множеству GPU. Например, прирост производительности для 10000 квантовых траекторий при использовании 10 GPU увеличился в 6 раз по сравнению с одной видиокартой.

Если сравнивать производительность разработанных параллельных версий программ на основе метода МК с использованием технологии CUDA и последовательной версии кода на языке C++ на CPU, то выигрыш в расчетах по времени оказывается очень существенным. Тестирование CPU версии кода производилось на (Intel Core i7 3770K @ 3.50 GHz, использовалось 1 ядро), данные по времени собраны в таблице 2. Видно, что даже для малого числа реализаций (M ~ 100), наблюдается ускорение во времени счета в 5 раз относительно CPU за счет использования графического ускорителя. С ростом числа реализаций GPU версия имеет неоспоримое преимущество, что видно из нижней строки таблицы 2.

Таблица 6. Время расчета усредненных M траекторий на GPU устройствах и последовательной версии на CPU. Параметры системы аналогичны тем, которые были взяты для моделирования в таблице 1.

Число реализаций, M / Тип устройства	1 CPU	1 GPU	4 GPU	10 GPU
$M = 1$	63,73 сек	94.50 сек	94,565 сек	95,13 сек
$M = 100$	639,10 сек	120,8 сек	108,065 сек	106,50 сек
$M = 1000$	64590,26 сек	150,91 сек	109,466 сек	107,86 сек
$M = 10000$	653247,14 сек	786,156 сек	360,70 сек	126,965 сек

5 Указания по выполнению лабораторной работы

1) Установить необходимые драйвера для CUDA (см. раздел 1.5) и убедиться, что программный тест завершается успешно.
2) Ознакомиться с алгоритмом на основе "растягивания" матрицы плотности (см. раздел 2.3).
3) Ознакомиться с алгоритмом на основе разложение матрицы плотности по группам Ли (матрицам Гелл-Мана) (см. раздел 2.4).
4) Ознакомиться с алгоритмом на основе квантового метода Монте-Карло для расчета диссипативной динамики квантовых систем, на примере квантового осциллятора (см. раздел 2.6).
5) Реализовать последовательные версии на CPU согласно разделам 2.3; 2.4 и 2.6 и сравнить полученные результаты вычислений.
6) Ознакомиться с основами программирования графических процессоров с использованием технологии CUDA и выполнить распараллеливание последовательной версии на основе разделов 2.4 и 2.6. Проверить

корректность реализации путем сравнения результатов с результатами последовательной версии.

6 Индивидуальные дополнительные задания

1) Распараллелить предложенный алгоритм программы (раздел 2.4) с помощью параллельных технологий OpenMP и MPI. Сравнить время расчета для различных параллельных алгоритмов.
2) Распараллелить предложенный алгоритм программы (раздел 2.6) с помощью параллельных технологий OpenMP и MPI. Сравнить время расчета для различных параллельных алгоритмов.

ЗАКЛЮЧЕНИЕ

Настоящее научно–методическое пособие посвящено применению гетерогенных вычислительных систем для моделирования физических процессов. В работе описаны основные принципы работы с технологией CUDA, установка и компиляция программы, описано лабораторное оборудование «Кластер НИФТИ ННГУ», возможности доступа к оборудованию для расчетов. А также представлена математическая модель и создан программный комплекс, который реализует параллельные вычисления на многопроцессорном кластере (с использованием протокола MPI) и графических процессорных устройств (с использованием технологии CUDA).

Используя разработанный программный комплекс, в качестве примера, выполнено численное моделирование диссипативной динамики многоуровневых квантовых систем на основе квантового метода Монте-Карло и метода разложения по некоммутативным группам Ли (матрицам Гелл-Мана) на примере нелинейного квантового осциллятора. Показано, что применимость GPU ускорителей позволило существенно ускорит расчеты. Развитая в работе техника расчета диссипативной динамики осцилляторных систем естественным образом может быть распространена на более сложные системы.

Таким образом, в работе показана эффективность и целесообразность проведения расчетов сложных физических задач на GPU ускорителей, что позволить проводить численные физические эксперименты за разумное время.

Список литературы

1. Д. Сандерс, Э. Кэндрот, *Технология CUDA в примерах. Введение в програмирование графических процессов*: Пер. с англ. Слинкина А.А., научный редактор Боресков А.В. М.: ДМК Пресс, 2011. –232 с.

2. Суперкомпьютерные технологии в науке, образовании и промышленности (Третий выпуск)/Под редакцией: академика В.А. Садовничего, академика Г.И. Савина, чл.-корр. РАН Вл.В. Воеводина.-М.: Издательство Московского университета, 2012. [http://hpc-russia.ru/book3_ready.html]

3. TOP500 Project [http://www.top500.org/]; TOP50 суперкомпьютеров [http://top50.supercomputers.ru]; Graph500 [http://www.graph500.org/]; HPC Challenge Benchmark [http://icl.cs.utk.edu/hpcc/index.html]

4. Эксафлопные технологии. Концепция по развитию технологии высокопроизводительных вычислений на базе суперэвм эксафлопного класса (2012-2020 гг.) [http://filearchive.cnews.ru/doc/2012/03/esk_tex.pdf]

5. А.В. Боресков, А.А. Харламов, *Основы работы с технологией CUDA*. Издательство "ДМК Пресс", 2010

6. http://www.3dnews.ru/905212 (дата обращения: 20.11.2014)

7. *NVIDIA CUDA 6.5 Programming Guide* http://docs.nvidia.com/cuda/index.html#axzz3JdUtOlvx

8. S.Haroche, J.M.Raimond. *Exploring the Quantum-Atoms, Cavities and Photons*, Ch.3. Oxford Univ. Press, Oxford, UK, 2006.

9. J.Q.You, F.Nori. Nature, 474, 585 (2011);

10. I.Buluta, S.Ashhab, F.Nori. Rep. Prog. Phys. 74, 104401 (2011).

11. G.L.Baker, J.A.Blackburn. *The Pendulum*. Oxford Univ. Press, 2005.

12. I. Siddiqi *et. al.*, Phys. Rev. B 73, 054510 (2006)

13. H. Khalil. *Nonlinear Systems*. Prentice-Hall, 1996.

14. A. Barone, G. Paterno *Physics and applications of the Josephson*

effect. New York. Wiley, 1982.

15. А.А. Измалков. Макроскопические квантовые эффекты в потоковом кубите: Дис. … к. ф.-м. н. Москва, 2005

16. М.О. Скалли, М.С. Зубайри *Квантовая оптика*. Под ред. В.В. Самарцева. – М.: ФИЗМАТЛИТ, 2003. - 512 с

17. M. B. Plenio and P. L. Knight, Rev. Mod. Phys. 70, 101 (1998).

18. С. Адлер, Р. Дашен. *Алгебры токов и их применение в физике частиц*. М.: Мир, 1970

19. В.П. Гергель, Р.Г. Стронгин. *Основы параллельных вычислений для многопроцессорных вычислительных систем*. - Н.Новгород, ННГУ, 2 изд, 2003.

20. В. П. Гергель, *Теория и практика параллельных вычислений. Бином. Лаборатория знаний*, 2007. – 424 с.

21. N. H. Cong, T. Mitsui. A class of explicit parallel two-tep Runge-Kutta methods. Japan. J. Ind. Appl. Math., 14 (1997)

22. B. P. Sommeijer. Explicit, high-order Runge-Kutta-Nyström methods for parallel computers. Appl. Numer. Math., 13 (1993)

23. B. P. Sommeijer. Parallel-iterated Runge-Kutta methods for stiff ordinary differential equations. Appl. Numer. Math., 45 (1993)

ПРИЛОЖЕНИЕ А. Пример программы на CUDA: расчет диссипативной динамики нелинейного квантового осциллятора квантовым методом Монте-Карло

```
#include <iostream>
#include <fstream>
#include <math.h>
#include <complex>
#include <vector>
#include <list>
#include "eigen3/Eigen/Core"
#include "eigen3/Eigen/Eigenvalues"
#include "Timer.h"

#include "median_cuda.h"



/// Uncomment to use MPI
#define USE_MPI
#ifdef USE_MPI
#include <mpi.h>
#endif


///#ifdef MSVC_VER
#ifdef _MSC_VER
#include <windows.h>

static inline double round(double val)
{
    return floor(val + 0.5);
}
#endif

typedef std::complex<real> complex;

// Imaginary unit
const complex I = complex(0.0, 1.0);



real dt, ts;
int Nstep, tperiod;
real ddt;
int nt, tmax; // В оригинале был real(?)
int kmax;
real omegaQ, omega, rf, gammaF, gammaE, gammaO, Omega, f0;
real Ams, mu, lambda, omegaJ, deltaOmegaJ, omegaRabi, period;
int processRank, numProcesses;



void init() {
       ///kmax = 10;
       kmax = 10;
    dt = 0.1;
       ts = 1.0;
```

```
        tperiod = 5;
        nt = 18;
    f0 = 0.;
        omegaQ = 1.0;
        omega = 1.01;
        gammaF = 0.0;
        gammaE = 0.0;
        gammaO = 0.1;
        Omega = 1.0;
        Ams = 0.;
        mu = 0.0;
        lambda = 0.;
        omegaJ = 1.0;
        deltaOmegaJ = 0;

        Nstep = (2*M_PI)/dt;
        tmax = round(Nstep)*tperiod;
        ddt = dt/30; // TODO: Why 20?
        omegaRabi = sqrt(pow(omegaQ - omega, 2) + pow(Ams, 2));
        period = 2*M_PI/omegaRabi;

#ifdef USE_MPI
        int argc = 0;
        char ** argv = 0;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);
        MPI_Comm_rank(MPI_COMM_WORLD, &processRank);
#else
        numProcesses = 1;
        processRank = 0;
#endif
}

void computeInitialC(complex * initialC)
{
        typedef Eigen::Matrix
                <
                        complex
                ,Eigen::Dynamic
                ,Eigen::Dynamic
                > matrix;

        matrix Hq = matrix::Zero(2, 2);
        Hq << omegaQ,   0.0,
                0.0,  -omegaQ;
        Hq*=0.5;
        // Hq matches

        Eigen::ComplexEigenSolver <matrix> eigen;
        eigen.compute(Hq, true);
        Eigen::EigenSolver <matrix>::EigenvectorsType eve = eigen.eigenvectors();
        matrix::EigenvaluesReturnType eign = eigen.eigenvalues();
        matrix evea(eve.rows(), eve.cols());
        for(int i=0; i<eigen.eigenvectors().cols(); i++) {
                evea.col(i) = eve.col(i).normalized();
        }

        for(int i=2; i<=evea.rows(); i++) {
                for(int j=i; j>=2; j--) {
                        if(eign(j-2).real() > eign(j-1).real()) {
                                eign.row(j-1).swap(eign.row(j-2));
                                evea.col(j-1).swap(evea.col(j-2));
```

```
                        }
                }
        }
        complex co[Nm];
        for (int i = 0; i < Nm; i++)
                co[i] = 0;
        co[nt-1] = 1;

        for (int i = 0; i < 2 * Nm; ++i)
                initialC[i] = complex(0, 0);
        int t_ = 0;
        for(int j=0; j<Nm; j++) {
                for(int i=0; i<2; i++) {
                        // TODO: WTF?
                        initialC[t_++] = evea(0, i)*co[j];
                }
        }
}



std::vector<real> nr_m;
std::vector<real> nr1_m;
std::vector<real> nr2_m;
std::vector<real> qp1_m;
std::vector<real> qp2_m;


template <typename T>
void printContainer(const char* name, T cont)
{
        std::ofstream fstr;
        fstr.open(name);
        for(unsigned int i = 0; i < cont.size(); ++i)
                fstr<<(i + 1)*dt << "   " << cont[i] << "\n";
        fstr.close();
}

void median (std::vector<real>& m) {
    for(unsigned int i = 0; i<m.size(); ++i) {
                real t = m[i];
#ifdef USE_MPI
                real localT = m[i];
                MPI_Reduce(&localT, &t, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
#endif
                t /= kmax;
                m[i] = t;
    }
}

void medians () {
    median(qp1_m);
    median(qp2_m);
    median(nr_m);
    median(nr1_m);
    median(nr2_m);
}

void output() {
        std::cout<<"\nDumping results... ";
        printContainer("qp1_m", qp1_m);
        printContainer("qp2_m", qp2_m);
```

```
	printContainer("nr1_m", nr1_m);
	printContainer("nr2_m", nr2_m);
	printContainer("nr_m", nr_m);
	std::cout<<"ok\n";
}

int main() {

	init();

	if (processRank == 0)
		std::cout << "Running " << numProcesses << " processes\n";

	complex * initialC = new complex[2 * Nm];
	computeInitialC(initialC);
	init_gpu(kmax, dt, ts, tperiod, nt, f0, omegaQ, omega, gammaF, gammaE, gammaO,
		Omega, Ams, mu, lambda, omegaJ, deltaOmegaJ, Nstep, tmax, ddt, omegaRabi,
period,
		processRank, numProcesses);

	nr_m.resize(tmax, 0);
	nr1_m.resize(tmax, 0);
	nr2_m.resize(tmax, 0);
	qp1_m.resize(tmax, 0);
	qp2_m.resize(tmax, 0);

	Stopwatch *timer = createStopwatch();
	timer->start();

	calculate_gpu(initialC, &nr_m[0], &nr1_m[0], &nr2_m[0], &qp1_m[0], &qp2_m[0]);
	delete [] initialC;

	medians();

	if (processRank == 0) {
		timer->stop();
		std::cout << "Computational time: " << timer->getElapsed() << " seconds\n";
		output();
	}

#ifdef USE_MPI
	MPI_Finalize();
#endif

	return 0;
}

#include "median_cuda.h"

#include <iostream>

#include "cuda_complex.hpp"
typedef cuda_complex::complex<real> complex;


__device__ real dt /*,  ts */;
__device__ int Nstep, tperiod;
__device__ real ddt;
__device__ int nt, tmax;
__device__ int kmax;
__device__ real omegaQ, omega, rf, gammaF, gammaE, gammaO, Omega, f0;
__device__ real Ams, mu, lambda, omegaJ, deltaOmegaJ, omegaRabi, period;
```

```
__device__ int processRank, numProcesses;
__device__ complex *Ed, *h, *q, *r, *inter;


__global__ void init_kernel(int _kmax, real _dt, real _ts, int _tperiod, int _nt,
       real _f0, real _omegaQ, real _omega, real _gammaF, real _gammaE, real _gammaO,
       real _Omega, real _Ams, real _mu, real _lambda, real _omegaJ, real _deltaOmegaJ,
       int _Nstep, int _tmax, real _ddt, int _omegaRabi, int _period,
       int _processRank, int _numProcesses, complex * _Ed,
       complex * _h, complex * _q, complex * _r, complex * _inter)
{
       if (blockIdx.x * blockDim.x + threadIdx.x > 0)
              return;

       kmax = _kmax;
    dt = _dt;
       //ts = _ts;
       tperiod = _tperiod;
       nt = _nt;
    f0 = _f0;
       omegaQ = _omegaQ;
       omega = _omega;
       gammaF = _gammaF;
       gammaE = _gammaE;
       gammaO = _gammaO;
       Omega = _Omega;
       Ams = _Ams;
       mu = _mu;
       lambda = _lambda;
       omegaJ = _omegaJ;
       deltaOmegaJ = _deltaOmegaJ;
       Nstep = _Nstep;
       tmax = _tmax;
       ddt = _ddt;
       omegaRabi = _omegaRabi;
       period = _period;

       Ed = _Ed;
       h = _h;
       q = _q;
       r = _r;
       inter = _inter;

       processRank = _processRank;
       numProcesses = _numProcesses;
}


__device__ void d_KroneckerProduct(const complex * A, int aSize, const complex * B, int
bSize, complex * ret) {
       int retSize = aSize * bSize;
       for(int i = 0; i < aSize; i++)
       for(int j = 0; j < aSize; j++)
              for(int i1 = 0; i1 < bSize; i1++)
              for(int j1 = 0; j1 < bSize; j1++)
                     ret[(bSize * i + i1) * retSize + bSize * j + j1] =
                            A[i * aSize + j] * B[i1 * bSize + j1];
}



__device__ void d_H(real tt, real Am, complex * H_) {
```

```
	complex _a[4] = {omegaQ, 0.0, 0.0, -omegaQ};
	complex _b[4] = {0.0, 1.0, 1.0, 0.0};
	complex _c[4] = {1.0, 0.0, 0.0, 1.0};
	complex _d[4] = {1.0, 0.0, 0.0, 0.0};
	complex I(0, 1);
	for (int i = 0; i < 4; i++)
		H_[i] = _a[i] * complex(0.5) + complex(Am*cos(omega*tt)*0.5)*_b[i] +
complex(gammaF)*I*_c[i] -
			complex(gammaE*0.5)*I*_d[i];
}


__device__ void d_qubit(real tt, real Am, complex * result) {
	complex H_[4];
	d_H(tt, Am, H_);
	return d_KroneckerProduct(Ed, Nm, H_, 2, result);
}

__device__ real d_gun(real f, real tt) {
	return f*cos(Omega*tt);
}

__device__ void d_tr(real f, real tt, complex * result) {
	real coeff = d_gun(f, tt);
	for (int i = 0; i < Nm; i++)
	for (int j = 0; j < Nm; j++)
		result[i * Nm + j] = h[i * Nm + j] * coeff + q[i * Nm + j];
}

__device__ void d_oscil(real f, real tt, complex * result) {
	complex _a[4] = {1.0, 0, 1.0, 0};
	complex _tr[Nm][Nm];
	d_tr(f, tt, &_tr[0][0]);
	d_KroneckerProduct(&_tr[0][0], Nm, _a, 2, result);
}

__device__ void d_Ht(real Am, real f, real tt, complex * result) {
	complex _qubit[2 * Nm][2 * Nm];
	d_qubit(tt, Am, &_qubit[0][0]);
	complex _oscil[2 * Nm][2 * Nm];
	d_oscil(f, tt, &_oscil[0][0]);

	for (int i = 0; i < 2 * Nm; ++i)
	for (int j = 0; j < 2 * Nm; ++j)
		result[i * 2 * Nm + j] = inter[i * 2 * Nm + j] + _qubit[i][j] +
_oscil[i][j];
}

__device__ void d_RK(real Am, real f, complex * c, real t, complex * result, real ddt) {
	complex _Ht[2 * Nm][2 * Nm];
	d_Ht(Am, f, t, &_Ht[0][0]);
	complex I(0, 1);
	for (int i = 0; i < 2 * Nm; i++)
	{
		result[i] = 0;
		for (int j = 0; j < 2 * Nm; j++)
			result[i] += _Ht[i][j] * c[j];
		result[i] *= -I * complex(ddt);
	}
}
```

```
__device__ real d_rdistr(unsigned long long & state) {
	state *= 302875106592253ULL;
	return real((state & ((1ULL << 59) - 1)) / double(1ULL << 59));
}



extern __shared__ real dynamicSharedMemory[];
__global__ void calculate_kernel(const complex * initialC,
	real * global_nr_m, real * global_nr1_m, real * global_nr2_m,
	real * global_qp1_m, real * global_qp2_m)
{
	complex c[2 * Nm], cn[2 * Nm];

	complex __cn[4] = {0.0, 0.0, 1.0, 0.0};
	complex __c[4] = {1.0, 0.0, 0.0, -1.0};
	complex __b[4] = {1.0, 0.0, 0.0, 0.0};
	complex _b[2 * Nm][2 * Nm];
	d_KroneckerProduct(Ed, Nm, __b, 2, &_b[0][0]);

	complex b[Nm];
	real a[Nm];
	real a1[Nm];
	real a2[Nm];

	for(int jj = processRank + (threadIdx.x + blockIdx.x * blockDim.x) * numProcesses;
		jj < kmax; jj += numProcesses * (blockDim.x * gridDim.x))
	{
		for (int i = 0; i < 2 * Nm; i++)
		{
			c[i] = initialC[i];
			cn[i] = 0;
		}

		unsigned long long state = 12345;
		for (int i = 0; i < jj * 10 * tmax; ++i)
		    state *= 302875106592253ULL;

		for(int j = 1; j<=tmax; j++) {
			real t = j*dt;
			real Am = (t <= period) ? Ams : 0.0;
			real f = (t <= 2*period) ? 0.0 : f0;

			real temp1 = 0.0;
			real temp2 = 0.0;
			for(int i = 0; i<Nm; i++) {
				temp1 += norm(c[2*i]);
				temp2 += norm(c[2*i+1]);
			}

			atomicAdd(&global_qp1_m[j-1], temp1);
			atomicAdd(&global_qp2_m[j-1], temp2);

			for (int i = 0; i < Nm; i++)
			{
				b[i] = 0;
				for (int k = 0; k < Nm; k++)
					b[i] += _b[i][k] * c[k];
			}

			complex dotProd(0, 0);
```

```
                        for (int i=0; i<2*Nm; i++)
                               dotProd += c[i] * b[i];
                        real Pe = abs(dotProd);

                        // Расчёт "веса" для скачка осциллятора
                        for(int i=0; i<Nm; i++)
                               a[i] = norm(c[2*i]) + norm(c[2*i+1]);

                        real Posc = 0.0;
                        for(int i=0; i<Nm; i++)
                               Posc += i*a[i];

                        for(int i=0; i<Nm; i++) {
                               a1[i] = norm(c[2*i+1]);
                               a2[i] = norm(c[2*i]);
                        }
                        real P1 = 0.0;
                        for(int i=0; i<Nm; i++)
                               P1 += i*a1[i];

                        real P2 = 0.0;
                        for(int i=0; i<Nm; i++)
                               P2 += i*a2[i];

                        atomicAdd(&global_nr1_m[j-1], P1);
                        atomicAdd(&global_nr2_m[j-1], P2);
                        atomicAdd(&global_nr_m[j-1], Posc);

                        real gamma = gammaF + gammaE*Pe + gammaO*Posc;

                        real ran = d_rdistr(state);

                        if(ran < dt*gamma) {
                               real s = d_rdistr(state);
                               if(s < gammaF/gamma)
                               {
                                      complex tmp[2 * Nm][2 * Nm];
                                      d_KroneckerProduct(Ed, Nm, __c, 2, &tmp[0][0]);
                                      complex tmpC[2 * Nm];
                                      for (int i = 0; i < 2 * Nm; i++)
                                      {
                                             tmpC[i] = 0;
                                             for (int k = 0; k < 2 * Nm; k++)
                                                    tmpC[i] += tmp[i][k] * c[k];
                                      }
                                      for (int i = 0; i < 2 * Nm; i++)
                                             c[i] = tmpC[i];
                               }
                               else {
                                      if(gammaF/gamma<s && s<(gammaF/gamma +
gammaE/gamma*Pe))
                                      {
                                             complex tmp[2 * Nm][2 * Nm];
                                             d_KroneckerProduct(Ed, Nm, __cn, 2, &tmp[0][0]);
                                             for (int i = 0; i < 2 * Nm; i++)
                                             {
                                                    cn[i] = 0;
                                                    for (int k = 0; k < 2 * Nm; k++)
                                                           cn[i] += tmp[i][k] * c[k];
                                             }
                                      }
                                      else {
```

```
                                        for (int i = 0; i < 2 * Nm; i++)
                                                cn[i] = 0;
                                        for(int n = 2; n <= Nm; n++) {
                                                cn[2*n-3 - 1] =
(real)(1.0/sqrt((real)n))*c[2*n-1 -1];
                                                cn[2*n-2 - 1] =
(real)(1.0/sqrt((real)n))*c[2*n - 1];
                                        }
                                }
                                real d = 0;
                                for(int i=0; i < 2*Nm; i++) {
                                        d += norm(cn[i]);
                                }
                                d = sqrt(d);
                                for (int i = 0; i < 2 * Nm; i++)
                                        c[i] = cn[i] / d;
                        }
                }
                else {
                        real ts = j*dt;
                        complex k1[2 * Nm], k2[2 * Nm], k3[2 * Nm], k4[2 * Nm], tmp[2
* Nm];
                        while(ts < (j+1)*dt) {
                                d_RK(Am, f, c, ts, k1, ddt);
                                for (int i = 0; i < 2 * Nm; i++)
                                        tmp[i] = c[i] + complex(0.5)*k1[i];
                                d_RK(Am, f, tmp, ts + 0.5*ddt, k2, ddt);
                                for (int i = 0; i < 2 * Nm; i++)
                                        tmp[i] = c[i] + complex(0.5)*k2[i];
                                d_RK(Am, f, tmp, ts + 0.5 * ddt, k3, ddt);
                                for (int i = 0; i < 2 * Nm; i++)
                                        tmp[i] = c[i] + k3[i];
                                d_RK(Am, f, tmp, ts + ddt, k4, ddt);
                                for (int i = 0; i < 2 * Nm; i++)
                                        c[i] += complex(1.0/6.0) * (k1[i] +
complex(2.0)*k2[i] +
                                                complex(2.0) * k3[i] + k4[i]);
                                ts += ddt;
                        }
                        real d = 0;
                        for (int m=0; m < 2 * Nm; m++)
                                d += norm(c[m]);
                        d = sqrt(d);
                        for (int m = 0; m < 2 * Nm; m++)
                                c[m] /= d;
                }
        }
    }
}


void _KroneckerProduct(const complex * A, int aSize, const complex * B, int bSize,
complex * ret) {
    int retSize = aSize * bSize;
    for(int i = 0; i < aSize; i++)
    for(int j = 0; j < aSize; j++)
        for(int i1 = 0; i1 < bSize; i1++)
        for(int j1 = 0; j1 < bSize; j1++)
            ret[(bSize * i + i1) * retSize + bSize * j + j1] =
                A[i * aSize + j] * B[i1 * bSize + j1];
}
```

```
int hostTmax;
int hostKmax;
complex *device_Ed, *device_h, *device_q, *device_r, *device_inter;
void init_gpu(int _kmax, real _dt, real _ts, int _tperiod, int _nt,
	real _f0, real _omegaQ, real _omega, real _gammaF, real _gammaE, real _gammaO,
	real _Omega, real _Ams, real _mu, real _lambda, real _omegaJ, real _deltaOmegaJ,
	int _Nstep, int _tmax, real _ddt, int _omegaRabi, int _period,
	int _processRank, int _numProcesses)
{
	int numDevices;
	cudaGetDeviceCount(&numDevices);
	int gpuIdx = _processRank / (_numProcesses / 2);
	cudaSetDevice(gpuIdx);
	std::cout << "Process " << _processRank << " is using GPU #" << gpuIdx << "\n";

	complex * host_Ed = new complex[Nm * Nm];
	complex * host_h = new complex[Nm * Nm];
	complex * host_q = new complex[Nm * Nm];
	complex * host_r = new complex[Nm * Nm];
	complex * host_inter = new complex[4 * Nm * Nm];
	for (int i = 0; i < Nm; i++)
		for (int j = 0; j < Nm; j++)
		{
			if (i == j)
				host_Ed[i * Nm + j] = 1;
			else
				host_Ed[i * Nm + j] = 0;
			host_h[i * Nm + j] = 0;
			host_q[i * Nm + j] = host_Ed[i * Nm + j];
			host_r[i * Nm + j] = host_Ed[i * Nm + j];
		}
	for(int i = 0; i<Nm-1; i++) {
		host_h[i * Nm + i + 1] = sqrt((real)(i+1));
		host_h[(i + 1) * Nm + i] = sqrt((real)(i+1));
	}
	complex I(0, 1);
	for(int i=0; i<Nm; i++) {
		host_q[i * Nm + i] = i*_omegaJ - _mu*i*i - i*_gammaO*0.5f*I;
	}
	for(int i=0; i<Nm-1; i++) {
		host_r[i * Nm + i] = _lambda*i*_omegaJ - _mu*_lambda*((real)0.25)*i*i;
	}
	complex inter_a[4] = {1.0, 0, 1.0, 0};
	_KroneckerProduct(host_r, Nm, inter_a, 2, host_inter);

	cudaMalloc(&device_Ed, Nm * Nm * sizeof(complex));
	cudaMalloc(&device_h, Nm * Nm * sizeof(complex));
	cudaMalloc(&device_q, Nm * Nm * sizeof(complex));
	cudaMalloc(&device_r, Nm * Nm * sizeof(complex));
	cudaMalloc(&device_inter, 2 * Nm * 2 * Nm * sizeof(complex));
	cudaMemcpy(device_Ed, host_Ed, Nm * Nm * sizeof(complex), cudaMemcpyHostToDevice);
	cudaMemcpy(device_h, host_h, Nm * Nm * sizeof(complex), cudaMemcpyHostToDevice);
	cudaMemcpy(device_q, host_q, Nm * Nm * sizeof(complex), cudaMemcpyHostToDevice);
	cudaMemcpy(device_r, host_r, Nm * Nm * sizeof(complex), cudaMemcpyHostToDevice);
	cudaMemcpy(device_inter, host_inter, 4 * Nm * Nm * sizeof(complex),
cudaMemcpyHostToDevice);

	init_kernel<<<1, 1>>>(_kmax, _dt, _ts, _tperiod, _nt,
		_f0, _omegaQ, _omega, _gammaF, _gammaE, _gammaO,
		_Omega, _Ams, _mu, _lambda, _omegaJ, _deltaOmegaJ,
```

```
                _Nstep, _tmax, _ddt, _omegaRabi, _period,
                _processRank, _numProcesses,
                device_Ed, device_h,
                device_q, device_r, device_inter);
        delete [] host_Ed;
        delete [] host_h;
        delete [] host_q;
        delete [] host_r;
        delete [] host_inter;

        hostTmax = _tmax;
        hostKmax = _kmax;
}



void calculate_gpu(void * initialC, void * nr_m, void * nr1_m, void * nr2_m,
                                void * qp1_m, void * qp2_m)
{
    cudaDeviceProp props;
    cudaGetDeviceProperties(&props, 0);
       int numBlocks = props.multiProcessorCount;
       int numThreads = (hostKmax + numBlocks - 1) / numBlocks;
       if (numThreads > 64)
              numThreads = 64;

       complex * d_initialC;
       cudaMalloc(&d_initialC, 2 * Nm * sizeof(complex));
       cudaMemcpy(d_initialC, initialC, 2 * Nm * sizeof(complex),
cudaMemcpyHostToDevice);

       real * hostZero = new real[hostTmax];
       for (int i = 0; i < hostTmax; ++i)
              hostZero[i] = 0;
       real *d_nr_m, *d_nr1_m, *d_nr2_m, *d_qp1_m, *d_qp2_m;
       cudaMalloc(&d_nr_m, hostTmax * sizeof(real));
       cudaMemcpy(d_nr_m, hostZero, hostTmax * sizeof(real), cudaMemcpyHostToDevice);
       cudaMalloc(&d_nr1_m, hostTmax * sizeof(real));
       cudaMemcpy(d_nr1_m, hostZero, hostTmax * sizeof(real), cudaMemcpyHostToDevice);
       cudaMalloc(&d_nr2_m, hostTmax * sizeof(real));
       cudaMemcpy(d_nr2_m, hostZero, hostTmax * sizeof(real), cudaMemcpyHostToDevice);
       cudaMalloc(&d_qp1_m, hostTmax * sizeof(real));
       cudaMemcpy(d_qp1_m, hostZero, hostTmax * sizeof(real), cudaMemcpyHostToDevice);
       cudaMalloc(&d_qp2_m, hostTmax * sizeof(real));
       cudaMemcpy(d_qp2_m, hostZero, hostTmax * sizeof(real), cudaMemcpyHostToDevice);
       delete [] hostZero;



       calculate_kernel<<< numBlocks, numThreads >>>(d_initialC, d_nr_m, d_nr1_m,
d_nr2_m, d_qp1_m, d_qp2_m);
       cudaDeviceSynchronize();
       cudaMemcpy(nr_m, d_nr_m, hostTmax * sizeof(real), cudaMemcpyDeviceToHost);
       cudaMemcpy(nr1_m, d_nr1_m, hostTmax * sizeof(real), cudaMemcpyDeviceToHost);
       cudaMemcpy(nr2_m, d_nr2_m, hostTmax * sizeof(real), cudaMemcpyDeviceToHost);
       cudaMemcpy(qp1_m, d_qp1_m, hostTmax * sizeof(real), cudaMemcpyDeviceToHost);
       cudaMemcpy(qp2_m, d_qp2_m, hostTmax * sizeof(real), cudaMemcpyDeviceToHost);

       cudaFree(d_initialC);
       cudaFree(d_nr_m);
       cudaFree(d_nr1_m);
```

```
cudaFree(d_nr2_m);
cudaFree(d_qp1_m);
cudaFree(d_qp2_m);
cudaFree(device_Ed);
cudaFree(device_h);
cudaFree(device_q);
cudaFree(device_r);
cudaFree(device_inter);}
```