УДК 004.942:519.684:550.348.432

# НЕКОТОРЫЕ ТЕХНОЛОГИЧЕСКИЕ АСПЕКТЫ ПРИМЕНЕНИЯ ВЫСОКОПРОИЗВОДИТЕЛЬНЫХ ВЫЧИСЛЕНИЙ НА ГРАФИЧЕСКИХ ПРОЦЕССОРАХ В ПРИКЛАДНЫХ ПРОГРАММНЫХ СИСТЕМАХ

**М. Р. Галимов**[1], **Е. В. Биряльцев**[2]

Рассмотрено решение задачи моделирования естественного микросейсмического поля численным методом с использованием различных технологий высокопроизводительных вычислений, а именно: вычислений на графических процессорах (ГПУ) и суперкомпьютере МВС-100К. Проведено сравнение производительности рассмотренных технологий (CUDA/JCuda, OpenCL) при выполнении на одной и нескольких видеокартах. Приведены оценки эффективности различных методов оптимизации программ на графических процессорах. Изучен вопрос одновременного выполнения вычислений на нескольких видеокартах, распределенных в сети (кластер ГПУ), с использованием технологий межсетевого взаимодействия (MPI, MPJ, GridGain) для обеспечения синхронизации данных.

**Ключевые слова:** высокопроизводительные вычисления, параллельные вычисления, кластер ГПУ, технология ГПУ, моделирование микросейсмического поля.

**1. Введение.** Вычисления с использованием графических процессоров получают все большее распространение [1, 2]. Большинство публикаций посвящено вопросам производительности выполнения различных тестов производительности [3] и реализации некоторых конкретных вычислительных задач [4], преимущественно научного характера. Отмечаемый в данных публикациях прирост производительности вычислений на графических процессорах по сравнению с вычислениями на центральных процессорах значительно отличается, изменяясь от десятков до сотен раз [5–7].

Таким образом, потенциальные возможности графических вычислений при решении широкого круга вычислительных задач не подлежат сомнению. Вместе с тем, технологические аспекты реализации вычислительных задач на системах ГПУ исследованы пока недостаточно хорошо. Представляет практический интерес эффективность доступа к ГПУ не только из языка C, но и из такого широко распространенного языка программирования как Java. Появление нескольких технологий доступа к ГПУ и интерфейсов к ним (CUDA, OpenCL и JCuda) ставит вопрос о сравнительной эффективности и особенностях реализации вычислительных задач с использованием этих сред. В последнее время начинают получать распространение кластеры ГПУ [8], что ставит вопрос о выборе технологий организации кластера. В нашей статье мы приводим сравнительный анализ эффективности и аспектов реализации одной вычислительной задачи с использованием различных охватывающих языков, технологий доступа к графическим вычислительным ресурсам и методов организации вычислительных кластеров ГПУ.

**1.1. Тестовая задача.** Выбор охватывающего языка программирования, технологии доступа к устройствам ГПУ особенно важен, если предполагается промышленное применение разработанного программного обеспечения. Одной из областей, где промышленное применение высокопроизводительных вычислений традиционно востребованы, является решение задач сейсморазведки на нефть и газ. Наиболее ресурсоемкой задачей здесь является моделирование распространения сейсмических волн для получения синтетических сейсмограмм. Несмотря на то что в исследовательских целях задачи распространения сейсмических волн в полноволновой постановке решались с 80-х годов прошлого века [9], использование полноволновых вычислений в промышленных пакетах сдерживается большими вычислительными затратами [10].

Известны реализации подобных задач на одиночных устройствах и кластерах ГПУ [11–13]. Однако указанные публикации касаются описания конкретных реализаций и не затрагивают вариантов реализации в различных технологических средах. В качестве основы для изучения технологических особенностей

[1] Казанский филиал Межведомственного суперкомпьютерного центра РАН, ул. Лобачевского, д. 2/31, 420111, г. Казань; ст. науч. сотр., e-mail: mgalimov@ksu.ru

[2] Научно-исследовательский институт математики и механики им. Н. Г. Чеботарева, Казанский (Приволжский) федеральный университет, ул. проф. Нужина, д. 1, 420111, г. Казань; зав. лаб., e-mail: Igenbir@yandex.ru

© Научно-исследовательский вычислительный центр МГУ им. М. В. Ломоносова

реализации полноволнового моделирования на системах ГПУ была использована разработанная ранее система моделирования распространения микросейсмов [14], реализованная на языке Java.

В рассматриваемой реализации была использована явная схема метода конечных элементов, что подразумевает итерационное умножение разреженных матриц на вектор как наиболее ресурсоемкий вычислительный блок. Код основного вычислительного модуля представлен в приложении 6.1. Наиболее трудоемким является выполнение умножения в функции "mult" (приложение 6.2). Эффективная реализация произведений матриц на вектор на мультипроцессорных системах и графических процессорах NVIDIA CUDA уже достаточно подробно исследована [15, 16], в связи с чем мы далее будем отмечать только технологические отличия тех или иных реализаций.

В настоящей статье приводятся результаты численного моделирования для двух моделей, которые в дальнейшем будут называться моделями А и Б. Размер сеточной модели используемых при вычислениях моделей А и Б составляют соответственно 74 Мб (количество узлов 840 848) и 717 Мб (количество узлов 8 170 042). В каждой из задач выполнялось соответственно 1 000 и 10 000 итерационных шагов моделирования распространения сейсмических волн. Под временем выполнения в данной статье подразумевается время выполнения всего вычислительного модуля "mult".

**1.2. Тестовые конфигурации.** В качестве аппаратных средств использовались:

— суперкомпьютер (СК) МВС-100К;

— вычислительный комплекс на базе высокопроизводительных рабочих станций.

СК МВС-100К пиковой производительностью 140.16 терафлоп в настоящий момент находится на второй строчке в списке самых мощных суперкомпьютеров РФ (http://supercomputers.ru). В его состав входят 1460 вычислительных модулей, каждый из которых оснащен двумя четырехъядерными процессорами Intel Xeon, работающими на частоте 3 ГГц. Для объединения узлов кластера в единое решающее поле используется технология Infiniband.

Вычислительный комплекс на базе высокопроизводительных рабочих станций, объединяет две рабочие станции, объединенных 100 Мб сетью Ethernet. Каждая рабочая станция включает в себя один четырехъядерный процессор Intel®920 Core$^{TM}$ i7, три вычислительных графических процессора Tesla C1060 и 24 Гб оперативной памяти.

В рамках исследования были реализованы и рассмотрены следующие локальные и распределенные программно-аппаратные решения.

| Локальные решения | Распределенные решения |
|---|---|
| 1. Однопроцессорные программные реализации на языке Java и C++. Однопроцессорные программные реализации на языке Java и C++ использовались в качестве базы для сопоставления результатов, полученных с помощью последующих реализаций.<br>2. Мультипроцессорная программная реализация на языке C++ (стандарт OpenMP). Данная реализация предназначалась для проведения оценки эффективности распараллеливания решаемой задачи на мультиядерных однопроцессорных рабочих станциях.<br>3. Программные реализации с использованием технологий ГПУ. Для проведения вычислений на графических процессорах были разработаны программные решения с использованием различных технологий ГПУ: C++/CUDA [17], C++/OpenCL, а также Java-интерфейса к CUDA — JCuda [18]. Для этих реализаций были рассмотрены вопросы влияния известных методов оптимизации вычислений на графических процессорах — использование разделяемой, константной и текстурной памяти [19, 20]. | 1. Мультипроцессорная программная реализация на языке C++ (стандарт MPI).<br>2. Мультипроцессорная и мультиграфическая программная реализация на языке C++ (стандарт MPI) и CUDA.<br>3. Мультипроцессорная и мультиграфическая программная реализация на языке Java++ (библиотека MPJ) и JCuda.<br>4. Мультипроцессорная и мультиграфическая программная реализация на базе программной Grid-платформы, с использованием JCuda. |

При разработке распределенных решений за основу была принята программная архитектура, согласно которой один программный процесс обеспечивает вычисление на одном центральном или графическом процессоре. Это позволило упростить взаимодействие между процессами независимо от их архитектурного отношения друг к другу (локальные или сетевые).

Ключевым моментом при создании распределенных решений является организация процедуры синхронизации обрабатываемых данных между процессами приложения, выполняющимися на разных узлах. Традиционно для этого используется MPI — стандарт организации взаимодействия между процессами

приложения, распределенных между вычислительными узлами. Программная реализация, созданная на основе данного подхода и выполняющая вычисления на распределенных центральных процессорах, использовалась в качестве эталонной для сравнения с другими распределенными решениями, а также для замеров производительности при выполнении на СК "МВС-100К".

Создание вычислительных кластеров с использованием графических процессоров является в настоящее время одним из наиболее интересных направлений развития высокопроизводительных систем. Среди посвященных этому вопросу исследований можно отметить, например, [21, 22]. В нашем исследовании были рассмотрены некоторые возможные способы программной организации кластера ГПУ на примере двух вычислительных узлов, связанных локальной сетью Ethernet 100 Мб. Для организации синхронизации данных использовался как стандарт MPI, так и стандарт MPJ-Express, реализующий стек MPI на языке Java [23, 24].

В настоящий момент быстро развиваются Grid-вычисления. Важными особенностями таких вычислений является доступ к вычислительным мощностям, распределенным в сети Интернет, а не в локальных сетях. В рамках данного исследования была изучена технологичность доступа к графическим вычислениям через Grid-среду. Для создания Grid-кластера была использована программная Java-платформа GridGain [25]. В нем реализована вычислительная парадигма, известная как MapReduce, т.е. разбиения задачи на большое количество небольших заданий, каждое из которых может быть выполнено на любом из узлов.

**2. Вычислительные эксперименты.**

**2.1. Базовые реализации.** Помимо имеющейся реализации на Java были реализованы однопроцессорные и мультипроцессорные (стандарт OpenMP) программные решения с использованием языка C++. Как видно из табл. 1, программная реализация на C++ быстрее реализации на Java примерно на 20%. Мультипроцессорная реализация быстрее однопроцессорной более чем в четыре раза при выполнении на одном четырехъядерном процессоре Intel®920 Core™ i7 с технологий Hyperthreading.

Таблица 1

Время выполнения при использовании различных классических технологий

| Модель | № | Решение | Время вычислений, с | Ускорение к C++ |
|---|---|---|---|---|
| Модель А | 1. | Java | 208 | 0.8 |
| | 2. | C++ | 157 | 1 |
| | 3. | C++ OpenMP | 35 | 4.4 |
| Модель Б | 1. | Java | 19 384 | 0.84 |
| | 2. | C++ | 16 368 | 1 |
| | 3. | C++ OpenMP | 3 661 | 4.5 |

Далее были проведены вычисления на СК МВС-100К. Для этого было реализовано программное решение на основе стандарта MPI. Результаты проведенных экспериментов представлены на рис. 1.

**2.2. Проведение вычислений на одном графическом процессоре.** Наиболее простое решение при организации хранения и обработки данных на графических процессорах предполагает их размещение в глобальной памяти. Глобальная память размещается в ОЗУ памяти видеокарты и является основным местом размещения данных для обработки в ГПУ, и позволяет осуществлять обмен данными между ЦПУ и ГПУ. Работа с глобальной памятью мало отличается от работы с обычной памятью компьютера и интуитивно понятна разработчику. Код организации умножения матрицы на вектор, предназначенный для выполнения на графическом процессоре, мало отличается от подобного исполняемого на центральном процессоре (приложение 6.3), изменения связаны с блочно-ядерной организацией вычислений на ГПУ. Изменения в основном вычислительном модуле (приложение 6.4) обусловлены необходимостью выделения и обмена данными с графическим процессором, а также запуска вычислений на нем.

Существенным недостатком глобальной памяти является ее высокая латентность, что замедляет сильно влияет на общую скорость вычислений. Поэтому далее были использованы различные известные методы оптимизации доступа к памяти для увеличения общей скорости вычислений. Для этого можно воспользоваться техниками доступа к разделяемой, константной и текстурной видам памяти.

Разделяемая память расположена непосредственно в потоковых мультипроцессорах и поэтому является наиболее быстрой. Ее наиболее эффективно использовать для кеширования данных, к которым осуществляется множественный доступ нитей блока. Сам механизм кеширования данных, а также после-

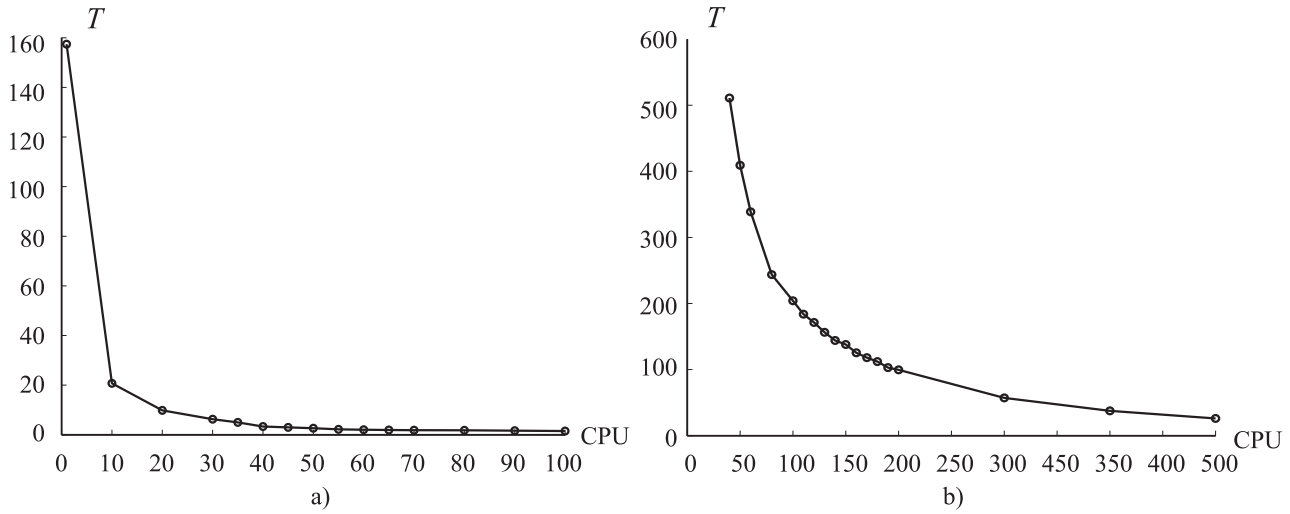


Рис. 1. Время моделирования в зависимости от количества одновременно используемых процессоров при вычислениях на "МВС-100К": a) модель А, b) модель Б

дующего доступа к ним нитей блока должен быть организован и запрограммирован разработчиком. С учетом крайне ограниченного размера разделяемой памяти, выделяемой каждому блоку, создание такого механизма может быть нетривиальным и значительно усложнить код ядра (приложение 6.5).

Константная и текстурные памяти, так же как и глобальная память, расположены в ОЗУ, но считанные данные кешируются, поэтому повторное чтение может быть очень быстрым. Ограничением этих видов памяти является, например, их доступность только на чтение для ГПУ. С точки зрения разработчика их использование не является особенно трудоемким и не значительно усложняет код (приложение 6.6) и (приложение 6.7).

Таблица 2

Прирост производительности в зависимости от технологии и методов оптимизации

| Модель | № | Решение | Время выполнения, с | Дополнительное ускорение, % | | |
|---|---|---|---|---|---|---|
| | | | | Разделяемая память | Константная память | Текстурная память |
| Модель А | 1. | CUDA | 11 | 18% | 33% | 33% |
| | 2. | JCuda | 12 | 25% | 33% | 33% |
| | 3. | OpenCL | 12 | 17% | 40% | |
| Модель Б | 4. | CUDA | 1 118 | 22% | 33% | 32% |
| | 5. | JCuda | 1 120 | 21% | 35% | 30% |
| | 6. | OpenCL | 1 111 | 20% | 36% | |

Для получения оптимальной версии программы были применены: технология использования разделяемой памяти для хранения и доступа к данным общих матриц, технология использования константной памяти для хранение вектора истинных номеров диагоналей общих матриц и технология использования текстурной памяти для хранения векторов перемещения. В табл. 2 представлен относительный прирост ускорения вычислений при последовательном применении оптимизаций в указанном порядке. Таким образом, был достигнут близкий к трехкратному прирост производительности по сравнению с первоначальной реализацией на ГПУ (табл. 3). В дальнейшем полученные программные реализации с максимальной производительностью использовались при организации одновременных вычислений на нескольких графических процессорах.

Полученные программные реализации можно также считать достаточно оптимизированными, так, например, основные затраты вычислительных ресурсов приходятся на вычислительные возможности графического процессора (96%), а все остальные операции (обмен и обновление данных и прочее) занимают незначительное в процентном соотношении время (по результатам анализа с помощью CUDA Visual

Таблица 3

Прирост производительности при комплексном использовании методов оптимизации

| Модель | № | Решение | Стандартное решение, с | Оптимизированное решение, с | Ускорение |
|---|---|---|---|---|---|
| Модель А | 1. | CUDA | 11 | 4 | 3 раза |
| | 2. | JCuda | 12 | 4 | 3 раза |
| | 3. | OpenCL | 12 | 6 | 2 раза |
| Модель Б | 4. | CUDA | 1 118 | 404 | 3 раза |
| | 5. | JCuda | 1 120 | 405 | 3 раза |
| | 6. | OpenCL | 1 111 | 569 | 2 раза |

Profiler), т.е. невычислительные затраты ресурсов минимальны.

При изучении аспектов использования технологии ГПУ для научных исследований нельзя обойти вниманием вопрос влияния на производительность вычислений с различной точностью. Согласно техническим характеристикам производительность графических процессоров сильно зависит от используемой точности вычислений. Наибольшая производительность достигается при вычислениях с одинарной точностью. При вычислениях с двойной точностью производительность снижается теоретически на порядок (при сравнении пиковой производительности, заявленной производителем) и примерно в два раза [26] на практике. Это подтверждается данными, приведенными в табл. 4.

Таблица 4

Время выполнения при различной точности вычисления для модели А

| № | Решение | Одинарная точность, мс | Двойная точность, с | Снижение производительности |
|---|---|---|---|---|
| 1. | C++ OpenMP | 35.2 | 42.4 | 1.2 раз (17%) |
| 2. | CUDA (1 ГПУ) | 4.1 | 7.4 | 1.8 раз (44%) |
| 3. | CUDA (2 ГПУ) | 2.2 | 3.9 | 1.8 раз (44%) |
| 4. | CUDA (3 ГПУ) | 1.6 | 2.8 | 1.8 раз (44%) |

Снижение быстродействия вычислений на графических процессорах существенно больше, чем на обычных процессорах. Можно отметить, что данный фактор может оказать влияние лишь в вычислениях, которые требуют повышенной точности. Кроме того, в последних моделях графических процессоров компании NVIDIA производительность вычислений с двойной точностью была повышена в четыре раза [27].

**2.3. Проведение вычислений на нескольких графических процессорах.** Следующим логическим этапом исследований была организация вычислений на нескольких графических процессорах. За основу были взяты программные реализаций с комплексной оптимизацией, рассмотренные выше.

Для реализации одновременных вычислений на нескольких графических процессорах матрица с входными данными разбивается на последовательные блоки, которые передаются соответствующим графическим процессорам. Для обеспечения корректности вычислений графическому процессору кроме основных данных передаются и граничные данные смежных блоков, размер блоков граничных данных зависит от модели. При проведении вычислений на каждом цикле производится синхронизация данных, под которой понимается обмен граничными данными между графическими процессорами.

После реализации программы были достигнуты показатели прироста производительности на каждый следующий графический процессор более 40% и получены окончательные результаты замеров производительности в зависимости от используемых технологий и количества графических процессоров, которые представлены на рис. 2. Дополнительно на рис. 3 приведены графики изменения ускорения при использовании нескольких графических процессоров относительно расчета на одном графическом процессоре и ускорения относительно времени выполнения на одном центральном процессоре. Как видно из графиков, эффективность выполнения модели одновременно на нескольких графических процессорах определяет-

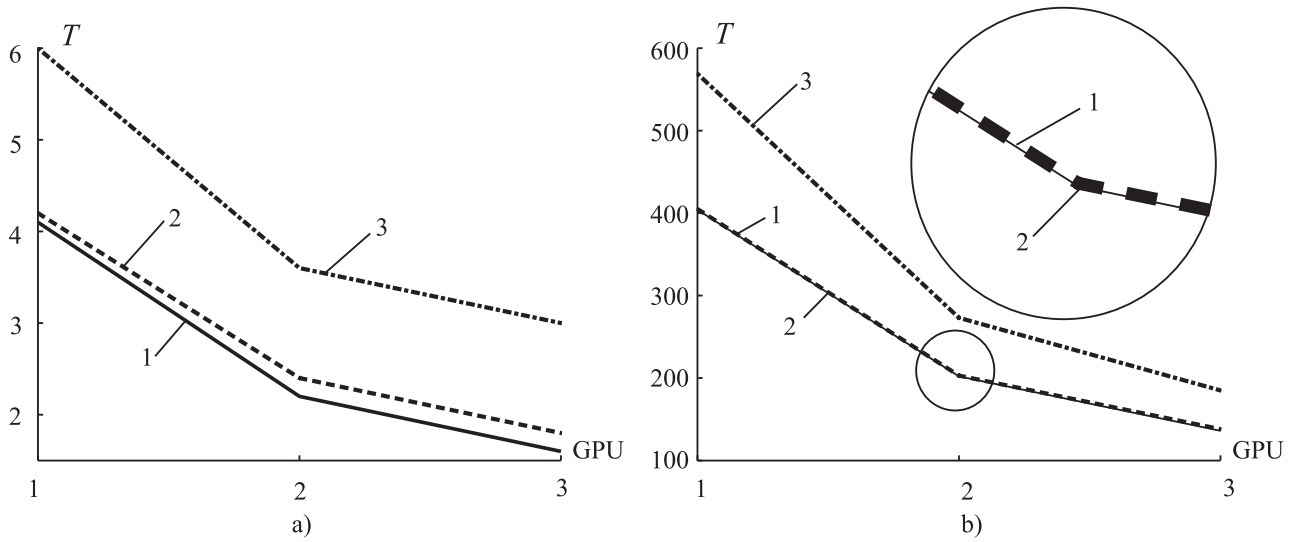


Рис. 2. Время моделирования в зависимости от количества одновременно используемых локальных графических процессоров и технологии: a) модель А, b) модель Б;
1 — CUDA, 2 — JCUDA, 3 — OPENCL

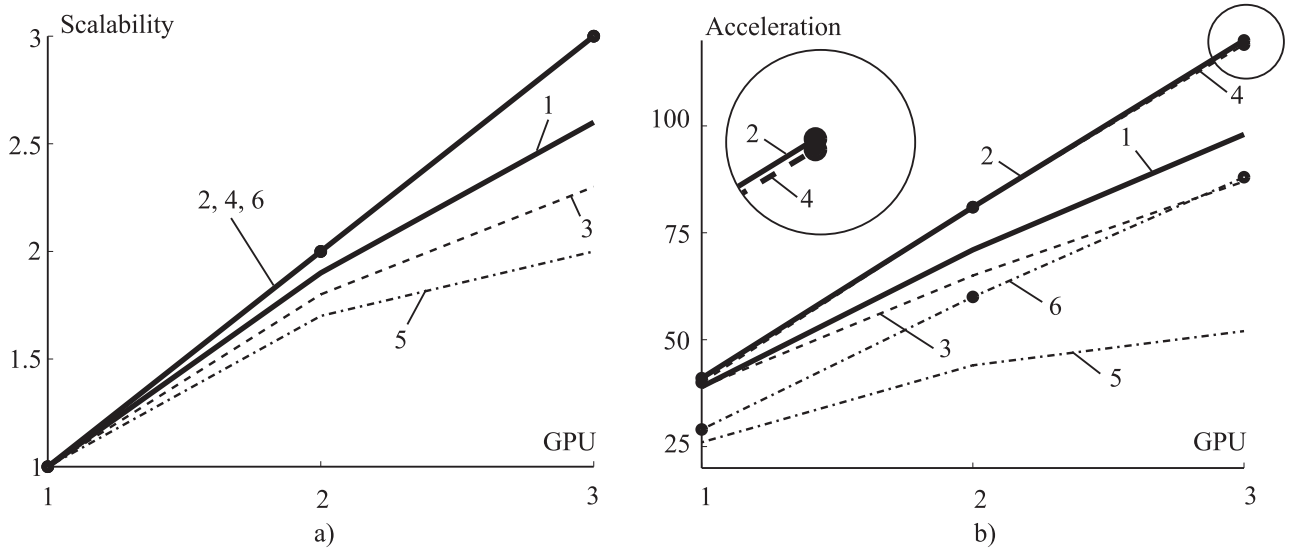


Рис. 3. Ускорение при выполнении на локальных графических процессорах для моделей А и Б: a) относительно выполнения на одном графическом процессоре, b) относительно выполнения на одном центральном процессоре; 1 — CUDA (A), 2 — CUDA (B), 3 — JCUDA (A), 4 — JCUDA (B), 5 — OPENCL (A), 6 — OPENCL (B)

ся размером модели. Для моделей большей размерности (модель Б) значения ускорений относительно времени выполнения на одном графическом процессоре лежат на одной прямой линии. Для модели А характерно снижение эффективности распараллеливания вычислений по сравнению с моделью Б. Это можно объяснить тем, что задержки при обмене данными между графическими процессорами становятся значительными к времени вычислений в одном цикле при небольших размерах модели.

**2.4. Проведение одновременных вычислений на графических процессорах в рамках двухузлового кластера.** Дальнейшим направлением повышения общей производительности программного комплекса является реализация возможности проведения одновременных вычислений на нескольких графических процессорах, расположенных на разных рабочих станциях, т.е. рассмотрение вопросов функционирования прототипа кластера графических процессоров (ГПУ кластера).

Результаты вычислений на ГПУ кластере приведены на рис. 4. Дополнительно на рис. 5 отображены графики изменения ускорения при использовании нескольких графических процессоров относительно расчета на одном графическом процессоре и ускорения относительно времени выполнения на одном

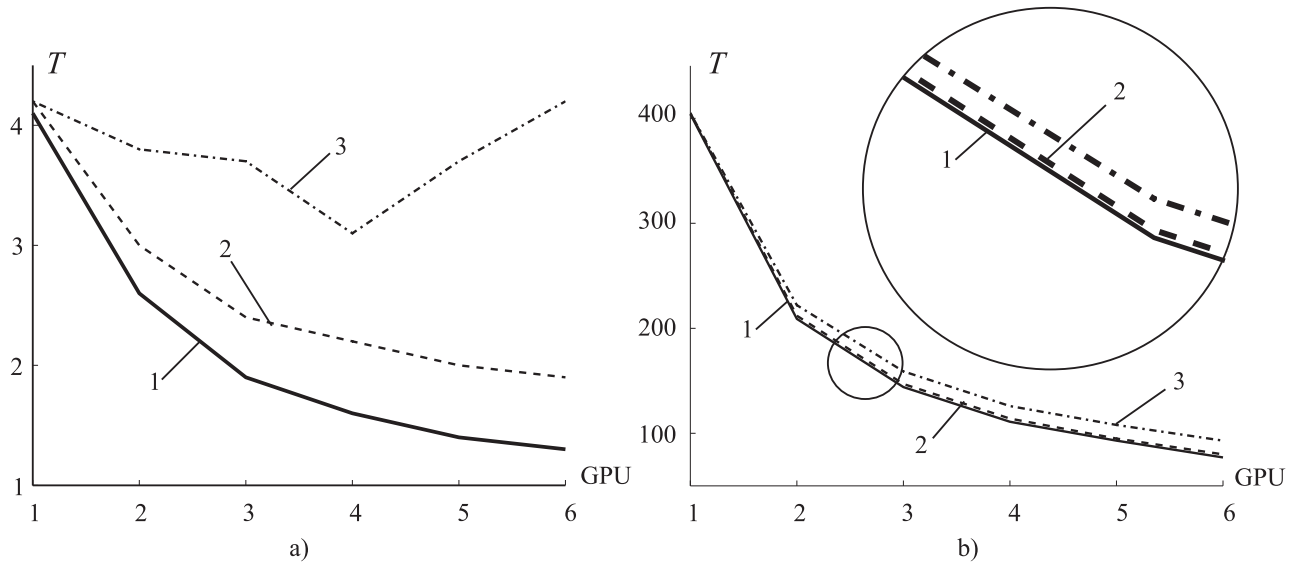


Рис. 4. Время моделирования в зависимости от количества одновременно используемых распределенных графических процессоров и технологии: a) модель А, b) модель Б; 1 — MPI CUDA, 2 — MPJ JCUDA, 3 — GG JCUDA

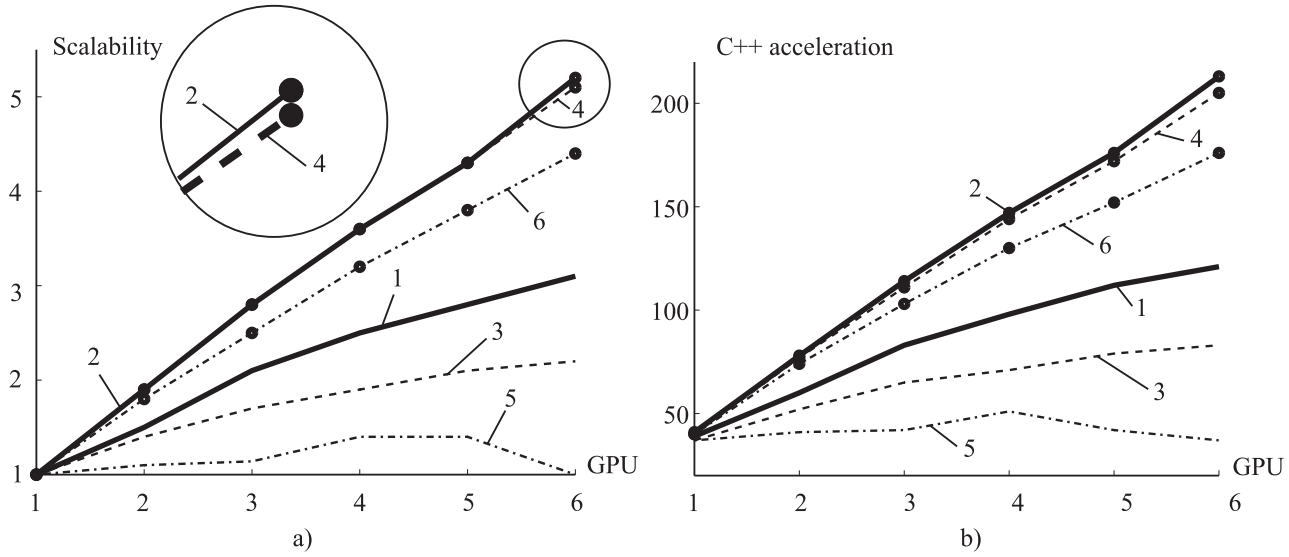


Рис. 5. Ускорение при выполнении на распределенных графических процессорах для моделей А и Б: a) относительно выполнения на одном графическом процессоре, b) относительно выполнения на одном центральном процессоре; 1 — MPI CUDA (A), 2 — MPI CUDA (B), 3 — MPJ JCUDA (A), 4 — MPJ JCUDA (B), 5 — GG JCUDA (A), 6 — GG JCUDA (B)

центральном процессоре. Как видно из графиков, эффективность выполнения модели одновременно на нескольких графических процессорах определяется размером модели. Для моделей большей размерности эффективность выше.

Если сравнивать производительность распределенных вычислений на графических процессорах с использованием различных технологий обмена данными, то видно, что для модели Б производительность вполне сопоставимая. В случае же модели А эффективность технологий обмена данными MPJ и GridGain значительно ниже эффективности использования MPI, для технологии GridGain эффективность вычислений на 6 графических процессорах ниже эффективности вычислений на 4 графических процессорах. Столь низкая производительность решения на основе грида при небольших размерах задачи объясняется следующими причинами: во-первых, GridGain является более высокоуровневым программным решением,

чем библиотеки MPI и MPJ, и в ней отсутствуют механизмы оптимизации передачи данных на уровне транспортных протоколов. Во-вторых, в данной программной реализации была использована синхронная модель обмена данными между узлами в отличие от асинхронной передачи, используемой в реализациях на MPI и MPJ. С увеличением размерности модели увеличивается время вычислений на графических процессорах и значение времени обмена данными по сети снижается. Таким образом, эффективность технологий становится практически одинаковой.

Проведенные вычисления на нескольких графических процессорах при использовании первоначальной версии программы показали, что прирост производительности на каждый следующий графический процессор составляет около 30%. Данный результат не был признан удовлетворительным, и был осуществлен поиск причин столь низкой эффективности одновременного использования нескольких графических процессоров. Было замечено, что при отсутствии обмена между графическими процессорами вычисления первого блока данных происходило значительно быстрее последующих. Причина была обнаружена и заключалась в известной особенности [19] организации оптимального чтения данных из глобальной памяти графического процессора, а именно: крайне важным является обеспечение условий для возможности объединения всех запросов одного полу-warp'а в один большой запрос (coalescing) на чтение непрерывного блока памяти. В данном случае это касалось необходимости выравнивания блока из 16 слов данных, к которым идет обращение из полуварпа. Причина их первоначального невыровненного состояния заключалась в том, что для всех блоков данных кроме первого характерно наличие граничных данных перед блоком основных данных, а так как размер граничных данных зависит от модели и в общем случае не кратен 64, то расположение 64-байтовых блоков основных данных в памяти оказывается неоптимальным. Этот пример еще раз свидетельствует о том, что для достижения максимальной производительности вычислений на графических процессорах и их группах требуется тщательный учет особенностей их архитектуры и функционирования.

**3. Обсуждение.** При проведении исследований, описанных в настоящей статье, уделялось большое внимание поиску существенных преимуществ различных технологий ГПУ друг перед другом. Сравнение результатов показывает, что технологии незначительно отличаются друг от друга по достигаемой производительности, а также по удобству использования. Разница в быстродействии между реализациями на CUDA и JCuda практически несущественна. Вообще, в связи с тем, что JCuda является Java-интерфейсом к CUDA и что код, выполняемый на ГПУ, в обоих случаях идентичен, а код, выполняемый на ЦПУ, нересурсоемок, то такие результаты вполне ожидаемы. Это подтверждается также и тем, что с ростом размера модели разница в быстродействии еще более снижается, так как относительное время на дополнительные невычислительные затраты, связанные с вызовами из Java C-процедур, уменьшается, а время продуктивного использования ГПУ увеличивается. Однако в настоящий момент имеются определенные трудности при достижении максимальной производительности в реализациях вычислений с использованием технологии OpenCL при использовании текстурной памяти: так, авторам не удалось добиться ускорения на соответствующей реализации программы. Но с учетом полученных результатов при использовании других методов ускорения вычислений, а также позиционирования данной технологии как открытого и универсального стандарта для параллельных вычислений в гетерогенных средах, данная технология выглядит более перспективной в направлении организации универсальных вычислений. В то же время для получения максимальной производительности на графических процессорах компании NVIDIA предпочтительнее применение технологии CUDA как наиболее полным образом использующую аппаратные возможности. Следовательно, выбор той или иной технологии зависит во многом от предпочтений разработчиков или особенностей проекта.

Таким образом, отвлекаясь от используемых технологий ГПУ, можно подвести некоторые итоги и сравнить производительность выполнения вычисления на графических и центральных процессорах. Как видно из табл. 5, производительность вычислений на графических процессорах более чем в 38 раз превышает производительность вычислений на одном центральном процессоре суперкомпьютера и в 9 раз производительность вычислений на одном четырехъядерном центральном процессоре с использованием технологии OpenMP. Особенно стоит отметить версию программы, реализующую вычисления на нескольких ядрах центральных процессоров с использованием стандарта OpenMP, так как трудоемкость ее создания значительно ниже трудоемкости создания версий для графических процессоров и, по сути, сводится к использованию нескольких специальных директив без написания уникального кода по сравнению с однопроцессорной версией. Особенно код для графических процессоров усложняется при использовании технологий оптимизации, в этом случае его размер может быть в 3–4 раза больше первоначального кода на C++.

Приведенные в статье экспериментальные данные показывают, что использование для вычислений

Таблица 5

Сравнение производительности комплексной оптимизации для различных технологий

| Модель | № | Решение | Время выполнения, с | Ускорение к C++ | Ускорение к C++ OpenMP |
|---|---|---|---|---|---|
| Модель А | 1. | C++ | 157 | 1 | |
| | 2. | C++ OpenMP | 35 | 4 | 1 |
| | 3. | CUDA | 4 | 39 | 9 |
| | 4. | JCuda | 4 | 39 | 9 |
| | 5. | OpenCL | 6 | 26 | 6 |
| Модель Б | 6. | C++ | 16 368 | 1 | |
| | 7. | C++ OpenMP | 3 661 | 4 | 1 |
| | 8. | CUDA | 404 | 41 | 9 |
| | 9. | JCuda | 405 | 41 | 9 |
| | 10. | OpenCL | 569 | 29 | 6 |

Таблица 6

Соответствие числа центральных и графических процессоров

| Модель | № | Время выполнения, с | ЦПУ | ГПУ | ЦПУ/ГПУ |
|---|---|---|---|---|---|
| Модель А | 1. | 4.1 | 38 | 1 | 38 |
| | 2. | 2.6 | 50 | 2 | 25 |
| | 3. | 1.98 | 62 | 3 | 21 |
| | 4. | 1.6 | 92 | 4 | 23 |
| | 5. | 1.4 | 101 | 5 | 20 |
| | 6. | 1.36 | 103 | 6 | 17 |
| Модель Б | 7. | 404 | 51 | 1 | 51 |
| | 8. | 209 | 97 | 2 | 48 |
| | 9. | 144 | 140 | 3 | 46 |
| | 10. | 111 | 182 | 4 | 45 |
| | 11. | 93 | 216 | 5 | 43 |
| | 12. | 77 | 253 | 6 | 42 |

одновременно нескольких графических процессоров приводит к значительному росту производительности. Более высокие показатели производительности достигаются при вычислении более требовательных к ресурсам моделей, что объясняется более полным использованием возможностей графических процессоров, т.е. в случае проведения одновременных вычислений на нескольких графических процессорах размер модели влияет на производительность. Это объясняется необходимостью синхронизации данных между графическими процессорами, и время синхронизации для небольших моделей оказывает более значимое влияние на быстродействие, чем для больших моделей.

В табл. 6 обобщены результаты экспериментов на суперкомпьютере и двухузловом кластере графических процессоров и приведено оценочное количество потребностей в графических и обычных процессорах. Таким образом, для рассмотренной задачи производительность одного графического процессора двухузлового кластера примерно соответствует производительности 40 процессоров суперкомпьютера для ресурсоемких задач, это хорошо согласуется с данными из табл. 5 для технологии CUDA.

При использовании ГПУ кластера более существенным становится влияние задержек в сети на быстродействие вычислений, так как расчетные блоки выполняются быстрее и нагрузка на сеть растет. Для

Таблица 7

Сравнение прироста производительности при локальном и распределенном выполнении на графических процессорах

| Модель | ГПУ | Время выполнения на локальных ГПУ, с | Время выполнения на распределенных ГПУ, с | Разница в быстродействии, % |
|---|---|---|---|---|
| Модель А | 2 | 2.2 | 2.6 | 18% |
| | 3 | 1.6 | 1.9 | 18% |
| Модель Б | 2 | 202 | 209 | 3% |
| | 3 | 136 | 144 | 6% |

модели Б происходит увеличение в 20 раз (табл. 9). Для оценки этого влияния были проведены расчеты на одинаковом количестве графических процессоров, расположенных локально и равномерно распределенных между узлами. Из табл. 7 видно, что обмен данными между узлами уменьшает общую производительность на 18% для модели А и 6% для модели Б.

Оценим влияние пропускной способности сети и скорости обмена данными на общую производительность вычислений в кластере. Для этого проведем вычисления без синхронизации данных между процессорами.

Как видно из табл. 8, для данной программной реализации с увеличением количества используемых графических процессоров растут и временны́е затраты на синхронизацию данных (от 4 до 12%). В то же время производительность программной реализации без синхронизации данных при выполнении на суперкомпьютере практически не отличается от производительности реализации с синхронизацией. Это объяснимо с учетом характеристики используемой сети Infiniband DDR: скорость двунаправленных обменов данными между двумя вычислительными модулями с использованием библиотек MPI находится на уровне 1400 Мбайт/сек. Минимальная латентность составляет 3.2, максимальная 4.5 мкс.

Таблица 8

Влияние синхронизации данных при выполнении моделирования модели Б на двухузловом кластере

| № | ГПУ | С синхронизацией, с (Ускорение к 1 ГПУ, раз) | Без синхронизации, с (Ускорение к 1 ГПУ, раз) | Разница |
|---|---|---|---|---|
| 1. | 2 | 209 (1,9) | 200 (2) | 4% |
| 2. | 3 | 144 (2,8) | 134 (3) | 7% |
| 3. | 4 | 111 (3,6) | 100 (4) | 10% |
| 4. | 5 | 93 (4,3) | 83 (4,8) | 11% |
| 5. | 6 | 77 (5,2) | 68 (6) | 12% |

Таблица 9

Нагрузка на сеть при выполнении вычислений на центральных и графических процессорах для модели Б

| № | Количество | ЦПУ, нагрузка на сеть, кбайт/с | ГПУ, нагрузка на сеть, кбайт/с | Отношение |
|---|---|---|---|---|
| 1. | 2 | 17 | 340 | 20 раз |
| 2. | 6 | 42 | 920 | 21 раз |

**4. Заключение.** Проведенные исследования показывают, что технологии графических вычислений достигли определенной технологической зрелости. Перенос рассмотренной задачи на технологии CUDA/JCuda, OpenCL не сопровождался какими-либо принципиальными проблемами, собственно сам перенос, отладка и тестирование каждого из рассмотренных вариантов заняли не более двух недель.

Сопоставление реализаций на классической кластерной и графических архитектурах показало высокую эффективность реализации задач подобного класса на графических устройствах. Для рассмотренного класса задач одно устройство ГПУ по вычислительным возможностям эквивалентно от 20 до 50 узлов кластера. Мы не проводили сопоставления стоимости архитектурных решений, но очевидно, что они различаются в десятки раз.

Выявленные трудности при использовании текстурной памяти в технологии OpenCL можно считать особенностями текущей реализации библиотеки. В то же время очевидным преимуществом среды OpenCL по сравнению с CUDA и JCuda является бóльшая универсальность, так как данная технология рассчитана на применения как на графических процессорах nVidia, так и графических процессорах AMD/ATI. Снижение эффективности вычислений на текущей реализации OpenCL в рассматриваемой задаче не более 20%, что можно считать адекватной платой за больший, по отношению к аппаратной базе, универсализм решения.

Рассмотренная технология CUDA и Java-интерфейс к ней JCuda показали сопоставимую эффективность, что дает необходимую гибкость при выборе технологий реализации, в том числе при наличии унаследованного программного обеспечения на Java.

Исследованные технологии доступа к ГПУ легко комплексируются с технологиями организации вычислительных кластеров. Кластерные реализации с использованием ГПУ показали высокую масштабируемость на исследованном диапазоне количества ГПУ. Была показана оправданность использования открытой платформы GridGain для построения вычислительных кластеров для задач, не требующих интенсивного обмена данными между вычислительными узлами. Важным аспектом организации кластера ГПУ является снижение нагрузки на сеть до уровня, делающего приемлемым применение доступных сетевых решений класса 1G Ethernet.

В результате реализации задачи моделирования распространения микросейсмов на технологии ГПУ удалось довести время расчета практически полезных моделей с нескольких часов до нескольких минут. Это дало возможность перевести задачу из разряда исследовательского инструмента в промышленно используемый инструмент при разведке залежей природных углеводородов.

**5. Благодарности.** Авторы благодарят компанию “Градиент” за предоставленные исходные тексты базового варианта программного обеспечения и многочисленные полезные консультации.

**6. Приложения.**

**6.1. Основной вычислительный модуль.**

```
while (t <= tModel) {

    if (t <= m * impulsDt)
        for (int i = 0; i < uzudarSize; i++) {
            f[2 * uzudar[i]] = power.get(m, 2 * i);
            f[2 * uzudar[i] + 1] = power.get(m, 2 * i + 1);
        }

    mult(nrz, NUMDIAG_HALF_M, kk1, offsets, ff1, q1);
    mult(nrz, NUMDIAG_HALF_M, kk2, offsets, ff2, q2);

    for (int i = 0; i < nrz; i++)
        ff3[i] = dtdt * (f[i] - q2[i] - q1[i]) / dbst_h[i];

    real *tff1 = ff1;
    ff1 = ff2;
    ff2 = ff3;
    ff3 = tff1;
    t += dt;

    if ((t > m * impulsDt) & (m < prows - 1)) m++;
}
```

**6.2. Умножение матрицы в диагональном формате на вектор.**

```
for (int row = 0; row < rows; row++) {
    rez[row] = data[rows * (numCols - 1) + row] * x[row];

    for (int i = 0; i < numCols - 1; i++) {
        const int n = NUMDIAG_HALF_M - i - 2;
        int offset = offsets[n];
        int col = row + offset;

        if (col >= 0 && col < nrz)
```

```
          rez[row] += data[n * rows + col] * x[col];

        int scol = row - offset;

        if (scol >= 0 && scol < nrz)
          rez[row] += data[n * rows + scol + offset] * x[scol];
    }
}
```

### 6.3. Основной вычислительный модуль при реализации на ГПУ.

```
cudaSetDevice(dev);
cudaGetDeviceProperties(&(curDeviceProp), dev);

kernel_blockSize = (curDeviceProp).maxThreadsDim[0];
dimGrid  = grid_dim(nrz, kernel_blockSize);
dimBlock = dim3(kernel_blockSize);
memorySizeRow = nrz * sizeof(float);

cudaMalloc((void**)&(q1_d), memorySizeRow);
cudaMalloc((void**)&(q2_d), memorySizeRow);
cudaMalloc((void**)&(ff1_d), memorySizeRow);
cudaMalloc((void**)&(ff2_d), memorySizeRow);
cudaMalloc((void**)&(ff3_d), memorySizeRow);
cudaMalloc((void**)&(dbst_d), memorySizeRow);
cudaMalloc((void**)&(f_d),    memorySizeRow);
cudaMalloc((void**)&(wave_d), memorySizeRow);

cudaMemset(ff1_d, 0, memorySizeRow);
cudaMemset(ff2_d, 0, memorySizeRow);
cudaMemset(ff3_d, 0, memorySizeRow);

kk1_d = new_gpu_vector_2d(nrz, NUMDIAG_HALF_M, kk1_h);
kk2_d = new_gpu_vector_2d(nrz, NUMDIAG_HALF_M, kk2_h);

cudaMemcpy(dbst_d, dbst_h, memorySizeRow, cudaMemcpyHostToDevice)

cudaMalloc((void**)&(offset_d), memoryOffset);
cudaMemcpy(offset_d, offset_h, memoryOffset, cudaMemcpyHostToDevice);

while (t < tnor){

    while( t < twrite && t < tnor ){

        if( t <= (m * impulsdt ) ){
            for (int i = 0; i < uzudarSize; i++){
                f[2 * uzudar[i]] = smPower.get(m, 2 * i);
                f[2 * uzudar[i]] = smPower.get(m, 2 * i + 1);
            }

            checkCUDAError(cudaMemcpy(f_d, f, nrz * sizeof(float), cudaMemcpyHostToDevice));
        }

        call_mult(dimGrid, dimBlock, q1_d, q2_d,..., dbst_d, kk1_d, kk2_d, ff1_d, ff2_d, offset_d);

        float *tempPointer = ff1_d;
        ff1_d = ff2_d;
        ff2_d = ff3_d;
        ff3_d = tempPointer;

        t += dt;

        if( (t > (m * impulsdt) ) & ( m < (prows - 1) ) ) m++;
    }

    twrite += dtwrite;
}
```

### 6.4. Умножение матрицы в диагональном формате на вектор при реализации на ГПУ.

```
__device__ void qMult(float *q_d, const int nrz, const int nnul, float* ff_d, float* kk_d, offsets){
```

```
    const int row = (blockIdx.y * gridDim.x + blockIdx.x) * blockDim.x +
threadIdx.x;

    if (row < nrz) {

        q_d[index] = (float) 0;
        q_d[row] += kk_d[ (NUMDIAG_HALF_M - 1) * nrz + row] * ff_d[row];

        for (int i = 0; i < NUMDIAG_HALF_M; i++) {
            const int offset = offsets[i];
            const int col = row + offset;

            if (col >= 0 && col < nrz)
                q_d[row] += kk_d[ i * nrz + col] * ff_d[col];

            const int scol = row - offset;

            if (scol >= 0 && scol < nrz)
                q_d[row] += kk_d[ i * nrz + scol + offset] * ff_d[col];
        }
    }
}

__global__ void mult(float* q1_d, float* q2_d, float* ff3_d,  const int nrz, const int nnul, const float dt,
                     float *wave_d, float* f_d, float* dbst_d, float* kk1_d, float*kk2_d, float* ff1_d,
                     float* ff2_d, int *offsets)
{

    const int row = (blockIdx.y * gridDim.x + blockIdx.x) * blockDim.x + threadIdx.x;

    qMult(q1_d, nrz, nnul,  ff1_d, kk1_d, offsets);
    qMult(q2_d, nrz, nnul, ff2_d, kk2_d, offsets);

    if (row < nrz) {
        ff3_d[row] = (dt * dt) * (  f_d [row] - q2_d[row] - q1_d[row]) /  dbst_d
[row];
        wave_d[row] =  (ff3_d[row] - ff2_d [row]) / dt;
    }
}
```

### 6.5. Умножение матрицы в диагональном формате на вектор при реализации на ГПУ с использованием разделяемой памяти.

```
extern __shared__ float array[];

__device__ void qMult(float *q_d, float *buf_ff_d, float *buf_ff_d_nnul, float *buf_ff_d_l_nnul, const int nrz,
                      const int nnul, float* ff_d, float* kk_d, offsets)
{

    const int row = (blockIdx.y * gridDim.x + blockIdx.x) * blockDim.x + threadIdx.x;
    const int buf_ff_Ind      = BOUNDARY_ADD      + threadIdx.x;
    const int buf_ff_Ind_nnul = BOUNDARY_ADD_NNUL + threadIdx.x;

    buf_ff_d        [buf_ff_Ind]      =  0;
    buf_ff_d_nnul   [buf_ff_Ind_nnul] =  0;
    buf_ff_d_l_nnul [buf_ff_Ind_nnul] =  0;

    if ( threadIdx.x >= blockDim.x - BOUNDARY_ADD)
        buf_ff_d [buf_ff_Ind + BOUNDARY_ADD] = 0;

    if ( threadIdx.x >= blockDim.x - BOUNDARY_ADD_NNUL){
        buf_ff_d_nnul   [buf_ff_Ind_nnul + BOUNDARY_ADD_NNUL] = 0;
        buf_ff_d_l_nnul [buf_ff_Ind_nnul + BOUNDARY_ADD_NNUL] = 0;
    }

    __syncthreads ();

    buf_ff_d        [buf_ff_Ind]      = ff_d[row];
    buf_ff_d_nnul   [buf_ff_Ind_nnul] = ( (row + nnul) <  nrz    ? ff_d[ row + nnul] : 0);
    buf_ff_d_l_nnul [buf_ff_Ind_nnul] = ( (row - nnul) >= 0 ? ff_d[ row - nnul] : 0);

    if (threadIdx.x < BOUNDARY_ADD)
```

```
        buf_ff_d [threadIdx.x] = (row - BOUNDARY_ADD < 0 ? 0 : ff_d[row  - BOUNDARY_ADD] );

    if ( (threadIdx.x >= blockDim.x - BOUNDARY_ADD) && (row + BOUNDARY_ADD) < nrz )
        buf_ff_d [buf_ff_Ind + BOUNDARY_ADD] = (row  + BOUNDARY_ADD < nrz ? ff_d[row + BOUNDARY_ADD] : 0);

    if (threadIdx.x < BOUNDARY_ADD_NNUL) {
        buf_ff_d_nnul [threadIdx.x] = ( row  + nnul - BOUNDARY_ADD_NNUL < nrz ? ff_d[row + nnul -
                BOUNDARY_ADD_NNUL] : 0 );
        buf_ff_d_l_nnul [threadIdx.x] = (row - nnul - BOUNDARY_ADD_NNUL < 0 ? 0 : ff_d[row - nnul -
                BOUNDARY_ADD_NNUL] );
    }

    if (threadIdx.x >= blockDim.x - BOUNDARY_ADD_NNUL) {
        buf_ff_d_nnul [buf_ff_Ind_nnul + BOUNDARY_ADD_NNUL] = ( row + nnul + BOUNDARY_ADD_NNUL < nrz ?
                ff_d [row + nnul + BOUNDARY_ADD_NNUL]: 0 );
        buf_ff_d_l_nnul [buf_ff_Ind_nnul + BOUNDARY_ADD_NNUL] = (row - nnul + BOUNDARY_ADD_NNUL < 0 ?
                0 : ff_d[row - nnul + BOUNDARY_ADD_NNUL]);
    }

    q_d[row] = (float) 0;
    __syncthreads ();

    if (row < nrz) {

        q_d[row] += kk_d[ (NUMDIAG_HALF_M - 1) * nrz + row] * buf_ff_d[buf_ff_Ind];

        for (int i = 0; i < NUMDIAG_HALF_M; i++) {
            const int offset = offsets[i];
            const int col = row + offset;
            const int buf_col = buf_ff_Ind + offset;

            if (col >= 0 && col < nrz) q_d[row] += kk_d[i * nrz + col] * buf_ff_d[buf_col];

            const int scol = row - offset;
            const int buf_scol = buf_ff_Ind - offset;

            if (scol >= 0 && scol < nrz) q_d[row] += kk_d[i * nrz + scol + offset] * buf_ff_d[buf_scol];
        }
    }
}

__global__ void mult(float* q1_d, float* q2_d, float* ff3_d,  const int nrz,  const int nnul, const float dt,
                     const int block_size, float *wave_d, float* f_d, float* dbst_d, float* kk1_d, float* kk2_d,
                     float* ff1_d, float* ff2_d, int *offsets)
{

    const int row = (blockIdx.y * gridDim.x + blockIdx.x) * blockDim.x + threadIdx.x;
    const int array_size      =  block_size + 2 * BOUNDARY_ADD;
    const int array_size_nnul =  block_size + 2 * BOUNDARY_ADD_NNUL;

    float *buf_ff_d        = (float*) array;
    float *buf_ff_d_nnul   = (float*) &buf_ff_d [array_size];
    float *buf_ff_d_l_nnul = (float*) &buf_ff_d_nnul [array_size_nnul];

    qMult(q1_d, buf_ff_d, buf_ff_d_nnul, buf_ff_d_l_nnul, nrz, nnul,  ff1_d, kk1_d, offsets);
    __syncthreads ();
    qMult(q2_d, buf_ff_d, buf_ff_d_nnul, buf_ff_d_l_nnul, nrz, nnul, ff2_d, kk2_d, offsets);
    __syncthreads ();

    if (row < nrz) {
        ff3_d[row] = (dt * dt) * (  f_d [row] - q2_d[row] - q1_d[row]) /  dbst_d [row];
        wave_d[row] =  (ff3_d[row] - ff2_d [row]) / dt;
    }
}
```

### 6.6. Основной вычислительный модуль при реализации на ГПУ при полной оптимизации.

```
cudaSetDevice(dev);
cudaGetDeviceProperties(&(curDeviceProp), dev);
...
kk1_d = new_gpu_vector_2d(nrz, NUMDIAG_HALF_M, kk1_h);
kk2_d = new_gpu_vector_2d(nrz, NUMDIAG_HALF_M, kk2_h);
```

```
cudaMemcpy(dbst_d, dbst_h, memorySizeRow, cudaMemcpyHostToDevice)

bind_texture_kk(kk1_d, kk2_d, nrz * NUMDIAG_HALF_M * sizeof(float),  nrz * NUMDIAG_HALF_M * sizeof(float));
bind_texture_ff(ff1_d, ff2_d, memorySizeRow);
checkCUDAError(cudaMemcpyToSymbol("c_offset", offset_h, memoryOffset, 0, cudaMemcpyHostToDevice));

while (t < tnor){

    while( t < twrite && t < tnor ){

        ...
        call_mult(dimGrid, dimBlock, q1_d, q2_d, ff3_d, nrz, nnul, dt, kernel_blockSize, wave_d, f_d, dbst_d);

        float *tempPointer = ff1_d;
        ff1_d = ff2_d;
        ff2_d = ff3_d;
        ff3_d = tempPointer;

        bind_texture_ff(ff1_d, ff2_d, memorySizeRow);

        t += dt;

        if( (t > (m * impulsdt) ) & ( m < (prows - 1) ) ) m++;
    }
    twrite += dtwrite;
}
```

### 6.7. Умножение матрицы в диагональном формате на вектор при реализации на ГПУ при полной оптимизации.

```
extern __shared__ float array[];
__constant__ int c_offset[NUMDIAG_HALF_M];
texture<float, 1, cudaReadModeElementType> tx_kk1_d;
texture<float, 1, cudaReadModeElementType> tx_kk2_d;
texture<float, 1, cudaReadModeElementType> tx_ff1_d;
texture<float, 1, cudaReadModeElementType> tx_ff2_d;

static __inline__ __device__ float tx_1Dfetch(texture<float, 1, cudaReadModeElementType> tx, int i) {
    return tex1Dfetch(tx, i);
}

__device__ void qMult(float *q_d, float *buf_ff_d, float *buf_ff_d_nnul, float *buf_ff_d_l_nnul, const int nrz,
                      const int nnul, texture<float, 1, cudaReadModeElementType> tx_ff_d, texture<float, 1,
                      cudaReadModeElementType> tx_kk_d){
    ...
    buf_ff_d        [buf_ff_Ind]      =                          tx_1Dfetch (tx_ff_d, row);
    buf_ff_d_nnul   [buf_ff_Ind_nnul] = ( (row + nnul) <  nrz ? tx_1Dfetch (tx_ff_d, row + nnul) : 0);
    buf_ff_d_l_nnul [buf_ff_Ind_nnul] = ( (row - nnul) >= 0   ? tx_1Dfetch (tx_ff_d, row - nnul) : 0);

    if (threadIdx.x < BOUNDARY_ADD)
        buf_ff_d [threadIdx.x] = ( row - BOUNDARY_ADD < 0 ? 0 : tx_1Dfetch(tx_ff_d, row - BOUNDARY_AD) );

    if ( (threadIdx.x >= blockDim.x - BOUNDARY_ADD) && (row + BOUNDARY_ADD) < nrz)
        buf_ff_d [buf_ff_Ind + BOUNDARY_ADD] = ( row + BOUNDARY_ADD < nrz ? tx_1Dfetch(tx_ff_d, row +
                                                 BOUNDARY_ADD) : 0);

    if (threadIdx.x < BOUNDARY_ADD_NNUL) {
            buf_ff_d_nnul [threadIdx.x] = ( row + nnul - BOUNDARY_ADD_NNUL < nrz ?
                                            tx_1Dfetch(tx_ff_d, row + nnul - BOUNDARY_ADD_NNUL) : 0 );
            buf_ff_d_l_nnul [threadIdx.x] = ( row  - nnul - BOUNDARY_ADD_NNUL < 0
? 0 :                                         tx_1Dfetch(tx_ff_d, row  - nnul - BOUNDARY_ADD_NNUL) );
    }

    if (threadIdx.x >= blockDim.x - BOUNDARY_ADD_NNUL) {
            buf_ff_d_nnul [buf_ff_Ind_nnul + BOUNDARY_ADD_NNUL] = ( row + nnul + BOUNDARY_ADD_NNUL < nrz ?
                    tx_1Dfetch(tx_ff_d, row + nnul + BOUNDARY_ADD_NNUL): 0 );
            buf_ff_d_l_nnul [buf_ff_Ind_nnul + BOUNDARY_ADD_NNUL] = ( row - nnul + BOUNDARY_ADD_NNUL < 0 ? 0 :
                    tx_1Dfetch(tx_ff_d, row - nnul + BOUNDARY_ADD_NNUL));
    }

    q_d[row] = (float) 0;
```

```
    __syncthreads ();

    if (row < nrz) {
        q_d[row] += tx_1Dfetch( tx_kk_d, (NUMDIAG_HALF_M - 1) * nrz + row) * buf_ff_d[buf_ff_Ind];
        for (int i = 0; i < NUMDIAG_HALF_M; i++) {
            const int offset = c_offset[i];
            const int col = row + offset;
            const int buf_col = buf_ff_Ind + offset;

            if (col >= 0 && col < nrz) q_d[row] += tx_1Dfetch( tx_kk_d, n * nrz + col) * buf_ff_d[buf_col];

            const int scol = row - offset;
            const int buf_col = buf_ff_Ind - offset;

            if (scol >= 0 && scol < nrz) q_d[row] += tx_1Dfetch( tx_kk_d, n * nrz + scol + offset) *
                                                     buf_ff_d[buf_scol];
        }
    }
}

__global__ void mult(float* q1_d, float* q2_d,   float* ff3_d, const int nrz, const int nnul, const float dt,
                     const int block_size, float *wave_d, float* f_d, float* dbst_d)
{
    ...
    qMult(q1_d, buf_ff_d, buf_ff_d_nnul, buf_ff_d_l_nnul, nrz, nnul,  tx_ff1_d, tx_kk1_d);
    __syncthreads ();
    qMult(q2_d, buf_ff_d, buf_ff_d_nnul, buf_ff_d_l_nnul, nrz, nnul, tx_ff2_d, tx_kk2_d);
    __syncthreads ();

    if (row < nrz) {
        ff3_d[row_inner] = dt * dt * (  f_d [row_inner] - q2_d[row_inner] - q1_d[row_inner]) /  dbst_d [row_inner];
        wave_d[row_inner] =  (ff3_d[row_inner] - tx_1Dfetch(tx_ff2_d, row_inner)) / dt;
    }
}

void bind_texture_kk(float *kk1_d, float *kk2_d, size_t size1, size_t size2)
{
    checkCUDAError(cudaBindTexture(0, tx_kk1_d, kk1_d, size1));
    checkCUDAError(cudaBindTexture(0, tx_kk2_d, kk2_d, size2));
}

void bind_texture_ff(float *ff1_d, float *ff2_d, size_t memorySizeRow)
{
    checkCUDAError(cudaBindTexture(0, tx_ff1_d,  ff1_d,  memorySizeRow));
    checkCUDAError(cudaBindTexture(0, tx_ff2_d,  ff2_d,  memorySizeRow));
}
```

## СПИСОК ЛИТЕРАТУРЫ


1. *Гужва А.Г., Доленко С.А., Персианцев И.Г.* Многократное ускорение нейросетевых вычислений с использованием видеоадаптера // XI Всероссийская научная конференция “Нейроформатика-2009”. Сб. науч. трудов. Москва, МИФИ, 2009. 126–133.
2. *Перепелкин Е.Е., Смирнов В.Л., Ворожцов С.Б.* Использование технологии NVIDIA CUDA при моделировании динамики пучка в ускорителях заряженных частиц // Вестник Российского университета дружбы народов. Серия: Математика, информатика, физика. 2010. № 1. 76–82.
3. *Danalis A., Marin G., McCurdy C., et al.* The Scalable HeterOgeneous Computing (SHOC) benchmark suite // Proc. of the Third Workshop on General-Purpose Computation on Graphics Processors (GPGPU 2010). Pittsburgh, 2010.
4. *Демидов Д.Е., Егоров А.Г., Нуриев А.Н.* Решение задач вычислительной гидродинамики с применением технологии NVIDIA CUDA // Учен. записки Казанского гос. ун-та. Серия физ.-матем. науки. 2010. **152**, кн. 1. 142–154.
5. *Деменков П.С., Иванисенко В.А.* Применение графических ускорителей для решения задачи раскладки графа // Математика в приложениях. Всероссийская конференция, приуроченная к 80-летию академика С. К. Годунова (Новосибирск, 20–24 июля 2009 г.). Тез. докладов. Новосибирск: Ин-т математики СО РАН, 2009. 100–101.
6. *Вишневский О.В., Лаврентьев М.М., Романенко А.А.* Применение графических ускорителей для выявления вырожденных олигонуклеотидных мотивов в регуляторных районах генов эукариот // Математика в прило-

жениях. Всероссийская конференция, приуроченная к 80-летию академика С. К. Годунова (Новосибирск, 20–24 июля 2009 г.). Тез. докладов. Новосибирск: Ин-т математики СО РАН, 2009. 62–63.
7. *Боярченков А.С., Поташников С.И.* Использование графических процессоров и технологий CUDA для задач молекулярной динамики // Вычислительные методы и программирование. 2009. **10**, № 1. 13–27.
8. *Yokota R., Narumi T., Sakamaki R., et al.* Fast multipole methods on a cluster of gpus for the meshless simulation of turbulence // Computer Physics Communications. 2009. **180**, № 11. 2066–2078.
9. *Гогоненков Г.Н.* Изучение детального строения осадочных толщ сейсморазведкой. Москва: Недра, 1987.
10. *Тульчинский П.Г.* Трассировка луча по конечноразностной двумерной сейсмической модели // Компьютерная математика. 2009. № 1. 29–36.
11. *Abdelkhalek R., Calendra H., Coulaud O., et al.* Fast seismic modeling and reverse time migration on a gpu cluster // High Performance Computing and Simulation. Leipzig, 2009. 36–44.
12. *Michea D., Komatitsch D.* Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards // Geophysical J. International. 2010. **182**, № 1. 389–402.
13. *Komatitsch D., Erlebacher G., Goddeke D., Michéa D.* High-order finite-element seismic wave propagation modeling with mpi on a large gpu cluster // J. of Computational Physics. 2010. **229**. 7692–7714.
14. *Бережной Д.В., Биряльцев Е.В., Биряльцева Т.Е.* Анализ спектральных характеристик микросейсм как метод изучения структуры геологической среды // НИИ математики и механики Казанского ун-та. 2003–2007 гг. Казань: Изд-во Казанск. гос. ун-та, 2008. 360–386.
15. *Baskaran M.M., Bordawekar R.* Optimizing sparse matrix-vector multiplication on GPUs: IBM Research Report RC24704 (W0812-047). IBM, 2008.
16. *Williams S., Oliker L., Vuduc R., et al.* Optimization of sparse matrix-vector multiplication on emerging multicore platforms // SC'07: Proc. of the 2007 ACM/IEEE Conference on Supercomputing. New York: ACM, 2007. 1–12 (http://dx.doi.org/10.1145/1362622.1362674).
17. *Nickolls J., Buck I., Garland M., Skadron K.* Scalable parallel programming with cuda // Queue. 2008. **6**, № 2. 40–53 (http://dx.doi.org/10.1145/1365490.1365500).
18. *Yan Y., Grossman M., Sarka V.* JCUDA: a programmer-friendly interface for accelerating Java programs with CUDA // Proc. of the 15th International Euro-Par Conference on Parallel Processing. Delft, 2009. **5704**. 887–899.
19. *Боресков А.В., Харламов А.А.* Основы работы с технологией CUDA. Москва: ДМК Пресс, 2010.
20. NVIDIA CUDA Programming guide. 2009. Version 2.3.
21. *Fan Z., Qiu F., Kaufman A., Yoakum-Stover S.* GPU cluster for high performance computing. 2004 (http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1392977).
22. *Goddeke D., Strzodka R., Mohd-Yusof J., et al.* Using GPUs to improve multigrid solver performance on a cluster // International J. of Computational Science and Engineering. 2008. **4**. 36–55.
23. *Baker M., Carpenter D.* MPJ: A proposed Java Message-Passing API and environment for high performance computing // The 2nd Java Workshop at IPDPS 2000. Cancun, 2000. 552–559.
24. *Shafi A., Manzoor J.* Towards efficient shared memory communications in MPJ express // 2009 IEEE Int. Symposium on Parallel&Distributed Processing. Washington: IEEE Computer Society, 2009. **5704**. 1–7.
25. *Харламов Д.* GridGain — грид уровня предприятия // GridGain Systems. 2008 (www.gridgain.com).
26. *Bell N., Garland M.* Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation, 2008.
27. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. NVIDIA Corporation, 2009.

Поступила в редакцию
02.09.2010