УДК 621.391:396

ПРИМЕНЕНИЕ ГРАФИЧЕСКОГО ПРОЦЕССОРА В РЕСУРСОЕМКИХ ВЫЧИСЛЕНИЯХ НА БАЗЕ БИБЛИОТЕКИ OPENCL

С. А. Запрягаев, А. А. Карпушин

Воронежский государственный университет

Поступила в редакцию 04.11.2010 г.

Аннотация. В статье рассматривается возможность использования графического процессора для решения ресурсоемких численных задач. Исследуются проблемы и ограничения, возникающие при переносе вычислений с центрального процессора на графический, а также предлагаются варианты оптимизаций приложений, использующих графический процессор.
Ключевые слова. Параллельные вычисления, вычисления на графическом процессоре.
Abstract. The article describes advantages and disadvantages of graphics processing unit usage as massively parallel computation device. Common issues and limitations, regarding to calculations transfer from CPU to GPU, are investigated. Several solutions, which greatly improve performance of GPU calculations, are proposed.
Keywords. Massively-parallel calculations on GPU, GPGPU, OpenCL.

ВВЕДЕНИЕ

Широкая потребность реализации высококачественной, интерактивной трехмерной графики привел в последние годы к существенному технологическому развитию графических процессоров (GPU), являющихся неотъемлемой частью любого персонального компьютера. Графический процессор приобрел качество высокопроизводительного устройства, основанного на применении параллельных технологий. При этом современный графический процессор предоставляет возможность осуществлять программирование обработки исходных данных на уровне прямых команд графического процессора.

Высококачественная трехмерная графика, требующая от GPU параллельной обработки ресурсоемких данных, привела к созданию специальной, в определенном смысле, уникальной архитектуры GPU. Современный графический процессор проектируется так, что основное число его транзисторов в чипе процессора используется для обработки данных, в ущерб кэшированию и управлению ходом программы. Таким образом, GPU особенно хорошо справляется с задачами, которые могут быть описаны как вычисления, параллельные по данным (data-parallel) – одна и та же программа выполняется над каждым элементом данных – с высоким отношением числа арифметических операций к числу операций с памятью.

Технологическая модернизация графических процессоров привела к тому, что вычислительная мощность GPU стала значительно опережать вычислительную мощность центрального процессора (CPU). Для примера, современная видеокарта NVIDIA GTX 280 содержит 240 ядер с пиковой производительность 933 Гигафлопс, что обеспечивает выполнение миллиардов операций в секунду с вещественными числами. В то время как современный центральный процессор Intel Core 2 Quad Q9550 имеет пиковую производительность только в 45.28 Гигафлопс, что почти в двадцать раз меньше производительности GPU.

Реализация систем программирования GPU привела к использованию его не только для графических приложений, но и для решения широкого класса иных задач. Такое применение графического процессора получило название GPGPU (General Purpose computations on Graphics Processing Unit). При этом типичными областями применения GPGPU стали такие направления IT технологий как: видеообработка [1], вычислительная химия [2], визуализация в медицине [3], обработка сигналов [4] и др. Отдельные вычислительные задачи, перенесен-

ные на GPU, позволили достигнуть вычислительной мощности, соответствующей кластеру из 30 CPU [5]. Использование графического процессора нашло свое применение и в задачах реализации сложных нейронных сетей. Так, в работе [6] продемонстрировано, что время обучения нейронной сети с использованием GPU сократилось в 150 раз. Еще одним перспективным направлением использования графического процессора является его применение в системах управления базами данных [7, 8, 9, 10]. Графический процессор в этом случае может выполнять вспомогательные задачи, такие как сортировка данных при добавлении новой записи в базу данных [7], выборка записей [9, 10], сжатие данных для их быстрой передачи [11].

Цель настоящей работы заключается в развитии технологии программирования переноса численных расчетов с центрального процессора на графический для достижения увеличенной производительности. Применение численных расчетов с использование графического процессора рассмотрено на примерах определения квантово-механических свойств молекулярных кластеров, а также в задаче обучения многослойной нейронной сети.

1. АРХИТЕКТУРА GPU

Графический процессор, первоначально предназначенный для задач визуализации, с течением времени эволюционировал в высокопараллельное многоядерное вычислительное устройство (рис. 1). Это связано с тем фактом, что в задачах визуализации в основном требовалось выполнять один и тот же алгоритм над каждым элементом исходных данных. При этом результат применения алгоритма над одним элементом исходных данных независим от результата применения алгоритма над любым другим элементом данных.

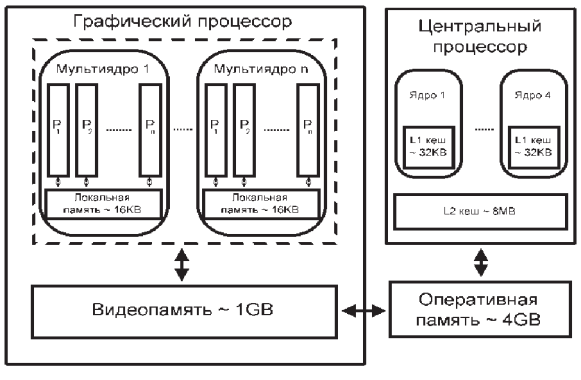

Рис.1. Архитектура графического процессора.

Графический процессор состоит из большого числа SIMD (Single Instruction, Multiple Data) ядер, каждое из которых выполняет одну и ту же микропрограмму над разными данными. В терминологии программирования графического процессора такую микропрограмму называют ядерной функции (kernel function). На рис. 1. SIMD ядра имеют обозначение P_i.

Графический процессор иерархически объединяет несколько SIMD ядер в так называемые *рабочие группы* (workgroups), между которыми осуществляется распределение данных. Рабочие группы могут обмениваться дополнительными данными посредством *локальной памяти* внутри группы и позволяют использовать синхронизирующие примитивы для синхронизации между отдельными SIMD ядрами. В то же время, графический процессор не предоставляет средств для синхронизации работы отдельных рабочих групп между собой. На рис. 1. рабочие группы обозначены «Мультиядро i».

Графический процессор также имеет иерархическую структуру памяти, с которой может работать разработчик. На самом высоком уровне иерархии каждая из рабочих групп может оперировать с общей глобальной памятью – видеопамятью, которая имеет высокую пропускную способность и высокую задержку при обращении к ней. Внутри рабочей группы также доступна локальная память, общая для нескольких SIMD ядер. Каждое SIMD ядро имеет также свою собственную память небольшого объема.

Для примера, видеокарта NVIDIA GTX 280 имеет 1GB видеопамяти, пропускную способность 141 GB/s и задержку на доступ в 400-600 тактов. Если отдельные SIMD ядра внутри рабочей группы обращаются к видеопамяти последовательно, эти обращения группируются в одно фактическое обращение. Правильно используя эту особенность можно в разы сократить количество фактических обращений к видеопамяти, что может существенно повысить производительность алгоритма и повысить общую пропускную способность данных для алгоритма. Каждая рабочая группа, как уже было отмечено выше, имеет и небольшой объем локальной памяти (как правило не больше 16Kb), которая имеет очень низкую задержку на доступ. В этой части памяти могут храниться все промежуточные данные алгоритма, чтобы минимизировать обращения к глобальной памяти.

Как отмечено выше, операции доступа к глобальной видеопамяти имеют существенную задержку. Поэтому, в целях оптимизации, нужно минимизировать обращения к видеопамяти как со стороны основной программы, выполняемой на центральном процессоре, так и со стороны микропрограмм, выполняемых на графическом процессоре.

2. БИБЛИОТЕКА OPENCL ДЛЯ ИСПОЛЬЗОВАНИЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ НА GPU

Одной из первых попыток описать «общие» вычисления на графическом процессоре явилась технология, получившая название CUDA (Compute Unified Device Architecture). Технология была разработана компанией Nvidia и только для видеокарт, производимых данной компанией. Вследствие этого данная технология в настоящей работе детально не рассматривалась.

Результатом развития GPU и его использования в задачах, несвязанных с компьютерной графикой, стала разработка единого стандарта описания вычислений на высокопараллельных системах – OpenCL (Open Computational Library). Сформировавшаяся библиотека OpenCL появилась на основе ранее разработанной технологии Nvidia CUDA, описывающей интерфейс взаимодействия приложения с вычислительными ресурсами графического процессора. В отличии от технологии CUDA, технология OpenCL описывает модель вычислений без связи с конкретным типом устройства, на котором эти вычисления будут исполняться. В силу того, что OpenCL разработан именно как стандарт вычислений на высокопараллельных системах, многие специфические возможности технологии CUDA были исключены из стандарта. В связи с этим, можно отметить, что, в целом, технология CUDA имеет больше возможностей, в сравнении с OpenCL, для описания параллельных вычислений, если в качестве вычислительного устройства выступает графический процессор Nvidia.

OpenCL позволяет описывать вычисления, абстрагируясь от конкретного устройства, на котором эти вычисления будут реализованы. В общем случае, алгоритмы, написанные с использованием OpenCL, могут исполняться на нескольких ядрах центрального процессора, на графическом процессоре, или на процессорах IBM Cell/B.E. В своей реализации OpenCL использует расширения языка C для описания алгоритма.

Структура библиотеки OpenCL состоит из нескольких ключевых элементов. Корневое приложение OpenCL, называемое *хост-приложением*, соединяется с одним или несколькими *вычислительными единицами* OpenCL (compute units) по средствам программного интерфейса. В свою очередь вычислительная единица состоит из многих *вычислительных элементов* (processing element), каждый из которых выполняет код параллельно по модели SIMD (Single Instruction – Multiple Data). При этом вычислительный элемент выполняет один и тот же код параллельно для многих элементов данных. Выполняемый код представляет собой небольшие программы на языке C и называется *ядром* (kernel). Программа на OpenCL представляет собой набор небольших функций, компилируемых «на лету». Функции OpenCL добавляются в *очередь выполнения*, из которой происходит выборка функций на выполнения или по порядку, или в случайном порядке. В целом, программа OpenCL, данные для ее работы и устройство выполнения задают *контекст выполнения*.

Ключевые элементы библиотеки OpenCL представлены на рис. 2.

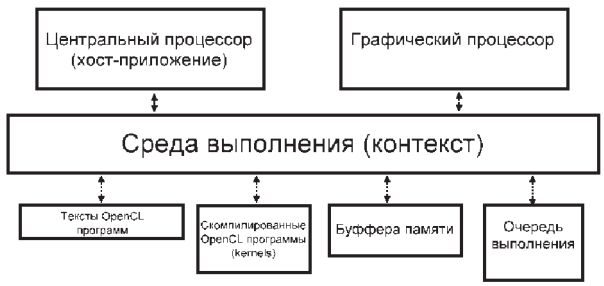

Рис. 2. Ключевые элементы библиотеки OpenCL.

Типичной архитектурой приложений, использующих графический процессор для вычислений, является следующая:

1. *Хост-приложение*, выполняемое центральным процессором, инициализирует *контекст выполнения* OpenCL. На этом шаге определяется тип устройства для высокопараллельных вычислений (CPU или GPU), создается одна или несколько *очередей выполнения* и

задаются их свойства: выбор команд по порядку или в случайном порядке.

2. Средствами библиотеки OpenCL выполняется компиляция микропрограмм (kernels), которые затем становятся доступны для использования из хост-приложения.

3. Задаются исходные данные – буфера памяти, содержащие входные данные алгоритма. Для каждого буфера памяти указывается его расположение (оперативная, видеопамять или локальная память), характер использования (только чтение, только запись, чтение и запись), и происходит его заполнение реальными данными.

4. Хост-приложение помещает в очередь выполнения вызовы микропрограмм и их параметры для последующего выполнения графическим процессором. Момент выбора микропрограммы из очереди выполнения определяется графическим процессором и не контролируется разработчиком.

5. Хост-приложение дожидается результата выполнения микропрограммы на графическом процессоре и осуществляет копирование полученных данных из видеопамяти в оперативную память.

В OpenCL введены три типа памяти: глобальная память, локальная и память отдельного вычислительного элемента. Первый тип – это глобальная память. Такая память доступна в любом месте внутри kernel и является независимой от локальной рабочей группы. Доступ к глобальной памяти является самым медленным, по сравнению с доступом к другим типам памяти. Внутри локальной группы возможно использования локальной памяти. Данные, расположенные в локальной памяти доступны только в пределах исполнения этой локальной группы. Другими словами, из одной локальной группы невозможно обратиться к локальным данным другой группы. Данный тип памяти работает значительно быстрее, чем глобальная память. Наконец, третий и самый быстрый тип памяти - это память отдельного элемента исполнения (work-item private memory). Этот тип памяти по сути доступен только в пределах исполнения конкретного элемента исполнения и служит для сохранения промежуточных результатов вычислений внутри элемента исполнения.

Хотя использование графического процессора значительно ускоряет производимые вычисления, тем не менее, использование GPU все же имеет и ряд ограничений.

Первое ограничение связано с точностью вычислений. Стандарт OpenCL позволяет работать только с 32-битными вещественными числами (single-precision floating point). Такие числа имеют в мантиссе только 6 значащих цифр после запятой. Поддержка вещественных чисел расширенной точности (double-precision floating point) включается с помощью расширений. Это означает, что производитель видеокарты определяет добавлять или нет поддержку вещественных чисел с расширенной точностью.

Второе ограничение связано с наличием атомарных операций. Атомарные операции, такие как увеличение или уменьшение значения целочисленной переменной на единицу, также реализуются на уровне расширений. Это означает, что в общем случае, необходимо использовать синхронизирующие примитивы (барьеры), чтобы использовать счетчик внутри локальной группы.

Кроме указанных выше ограничений существуют ограничения на доступ к памяти. Непоследовательный доступ к памяти может привести с большой потере производительности. Чтобы эффективно использовать параллельные вычисления на графическом процессоре необходимо обращаться к памяти специальным образом, избегая, где это возможно, непоследовательный доступ.

3. ИСПОЛЬЗОВАНИЕ OPENCL

Несмотря на то, что синтаксис OpenCL является расширением синтаксиса языка C, программы, использующие OpenCL можно писать и на других языках. Это достигается за счет того, что функции OpenCL компилируются во время выполнения программы в код под конкретное устройство, на котором затем происходит выполнение. Такая структура позволяет разделить код функций OpenCL (kernels) от кода, который использует эти функции.

Так, существует несколько программных библиотек, позволяющих использовать OpenCL в других языках программирования и средах, таких как, FORTRAN, Python (PyOpenCL, Theano, CLyther), Delphi, MATLAB и т.п. Более того, компиляция OpenCL кода “на лету” позволяет динамически изменять параметры и реализацию kernels.

В настоящей работе исследуется применение языка Python [12] для проведения численных расчетов. Достоинством языка Python является возможность осуществлять быструю разработку: Python является интерпретируемым языком, не требует времени на компиляцию программы целиком и позволяет модифицировать уже исполняемый код. Python является очень гибким и достаточно простым языком, с небольшим числом синтаксических правил описания языка. Недостатком языка является его интерпретируемый характер: вычисления, выполняемые средствами языка, работают медленнее по сравнению с другими языками.

Для увеличения производительности языка были созданы специальные программные пакеты:

· NumPy[13] – пакет, ускоряющий работу с числами и массивами данных на основе их внутренней реализации на языке C++;

· SciPy[14] – пакет, предоставляющий дополнительные возможности для высокоточных научных расчетов;

· PyOpenCL[15] – пакет, позволяющий оперировать данными пакетов NumPy и SciPy на графическом процессоре на основе стандарта OpenCL.

Библиотеки CLyther[16] и Theano[17] являются довольно новыми, полностью скрывающими написание кода функций OpenCL. В этих библиотеках функции вычислений пишутся на языке Python, и затем автоматически преобразуются в код на языке C, который исполняется OpenCL. Естественно, что не каждую функцию возможно преобразовать таким образом, и поэтому в этих библиотеках существуют ограничения на задание функций для вычислений.

Программный пакет PyOpenCL позволяет задавать kernels в коде программы (как строковые переменные) и вызывать их как обычные функции, передавая в качестве параметров массивы NumPy. Вызванная функция начнет работать параллельно с основной программой. Как только результаты выполнения функции на OpenCL понадобятся в основной программе, необходимо вызвать специальную функцию ожидания результата. Такой подход позволяет максимально эффективно распределить нагрузку между CPU и GPU задачами. Типичной ситуацией является запуск нескольких OpenCL функций подряд, затем выполнение задач на CPU, которые не могут быть перенесены на GPU (обработка ввода пользователя, чтение данных из файлов и т.д.) вплоть до момента, когда возникнет необходимость получения данных из GPU, затем вызов функции ожидания данных GPU.

В целом, можно сказать, что программный пакет PyOpenCL позволяет эффективно и без ограничений применять параллельные вычисления на графическом процессоре с использованием языка Python.

4. ПРИМЕРЫ ПРИМЕНЕНИЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ НА GPU

4.1. СЕЧЕНИЕ УПРУГОГО РАССЕЯНИЯ ЭЛЕКТРОНОВ НА СЛОЖНЫХ МОЛЕКУЛЯРНЫХ КЛАСТЕРАХ

В качестве практического примера применения библиотеки OpenCL в настоящей работе рассматривается задача ускорения квантово-механических вычислений сечения упругого рассеяния на наноразмерных системах.

Разработанный программный модуль написан на языке Python, с использованием библиотеки PyOpenCL, позволяющей использовать OpenCL внутри приложений на Python. Выбор языка реализации основан на удобстве использования и его кроссплатформенности.

Исходными данными для разработанного модуля являются cub-файлы данных, полученные с помощью программного комплекса Gaussian03 [18], значение начальной энергии падающего электрона и направление его падения по отношению к геометрии кластера. Cub-файлы содержат значения электронной плотности $n(\vec{r})$ сложного молекулярного кластера, внутри заданного пользователем объема в форме куба. Сечение упругого рассеяния рассчитывается в Борновском приближении по формуле:

$$\frac{\partial\sigma}{\partial\Omega} = \frac{4m^2e^2}{\hbar^4 q^4}\left|\int \rho(\vec{r})\exp(i\vec{q}\cdot\vec{r})dv\right|^2, \qquad (1)$$

где m – масса электрона, e – заряд электрона, $\hbar$ – постоянная Планка, $\vec{q} = \vec{k} - \vec{k}'$, $\vec{k}$ и $\vec{k}'$ – волновые векторы электрона в начальном и конечном состояниях соответственно, $\rho(\vec{r})$ – плотность заряда молекулярного кластера:

$$\rho(\vec{r}) = \sum_{k=1}^{N} Z_k e\delta(\vec{r} - \vec{r}_k) - en(\vec{r}). \qquad (2)$$

Здесь Z_k – заряд ядра k-го атома в кластере, $\vec{r}$ – положение k-го атома в кластере, а $n(\vec{r})$ – электронная плотность кластера.

В качестве исходного молекулярного кластера рассматривался фуллерен C_{60}, cub-файл электронной плотности $n(\vec{r})$ которого содержал 10^6 точек.

Для эффективного вычисления сечения упругого рассеяния выполнялся следующий алгоритм, минимизирующий обмен данными между центральным и графическим процессорами:

1. Для заданной начальной энергии E электрона определяется список всевозможных векторов q, определяемых двумя углами θ и φ. Углы θ и φ определяют положение конечного вектора электрона $\vec{k}'$ относительно начального вектора $\vec{k}$. Формирование списка векторов q осуществляется полностью на графическом процессоре. По завершению формирования списка, данные копируются в оперативную память, где с ними может работать центральный процессор.

2. Каждый из вычисленных векторов q участвует в Фурье-преобразовании электронной плотности в точках, равномерно распределенных внутри куба данных. На этом этапе определяется вклад значений электронной плотности в каждой точке куба в окончательное значение сечения упругого рассеяния. Вычисления выполняются параллельно над каждой точкой куба на графическом процессоре. Результатом вычислений являются массивы данных для действительной и мнимой частей Фурье-преобразования.

3. Определяется вклад заряда ядер в значение сечения упругого рассеяния. Вычисления выполняются параллельно для каждого атома на графическом процессоре.

4. Центральный процессор ожидает получения результатов вычислений на этапах 2 и 3 и осуществляет операцию редукции над полученными массивами данных, вычисляет значение сечения упругого рассеяния. Этот этап является узким местом алгоритма, так как ожидание данных от графического процессора приводит к простою в работе центрального процессора.

5. Полученные значения сечения упругого рассеяния для списка векторов q интегрируются по углу φ для получения зависимости сечения упругого рассеяния от угла рассеяния θ. Полученный результат интегрируется по углу θ для получения величины полного сечения рассеяния. Интегрирование данных осуществляется на графическом процессоре методом Симпсона.

Первоначальная реализация алгоритма была написана на языке C++, без применения специальных приемов распараллеливания – Реализация 1. В Реализации 2 для вычислений использован улучшенный модуль на языке Python с применением пакета PyOpenCL, но в качестве вычислительного устройства параллельных вычислений прямо указывался центральный процессор. В Реализации 3 для вычислений использован графический процессор. Во всех случаях конфигурация компьютера для тестирования включала: CPU – Intel Core 2 Duo, RAM - 2GB, GPU - GeForce 8800GT.

Полученные результаты тестирования приведены на рис. 3.

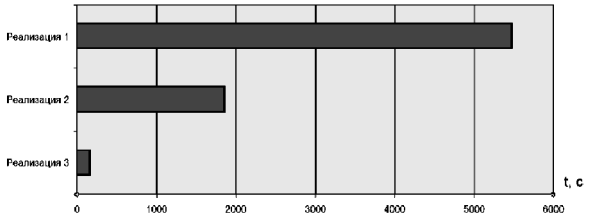

Рис. 3. Сравнение времени выполнения реализаций, в секундах.

Как видно из рис. 3 общее время выполнения удалось уменьшено в 33 раза, с полутора часов до 3-х минут. Уменьшение времени выполнения Реализации 2 более чем в 2 раза по сравнению с Реализацией 1 объясняется тем, что OpenCL использует все ядра CPU и генерирует более оптимизированный код, с использованием таких расширений как SSE2. Представленная Реализация 3 использует еще не все документированные возможности OpenCL. В частности операции редукции по-прежнему выполнялись на CPU – что является узким местом кода. При оптимизации данного узкого места на GPU пиковое ускорение Реализации 3 может составить 60-80 раз.

4.2. ОБУЧЕНИЕ МНОГОСЛОЙНОЙ НЕЙРОННОЙ СЕТИ

Другой важной практической задачей, для которой перенос вычислений на графический процессор дает существенное ускорение, является задача обучения многослойных нейронных сетей.

В многослойных нейронных сетях нейроны одного слоя никак не связаны между собой, и это позволяет вычислять выходные

сигналы всех нейронов внутри этого слоя параллельно.

В качестве приложения, демонстрирующего преимущества использования графического процессора для обучения и вычисления искусственных нейронных сетей, в настоящей работе была разработана программная библиотека для работы с нейронными сетями.

Программная библиотека написана на языке программирования Python с использованием программных пакетов: NumPy и PyOpenCL, позволяющих эффективно обрабатывать большие массивами числовых данных на центральном и графическом процессорах.

Библиотека позволяет конфигурировать структуру нейронной сети (количество слоев, количество нейронов в каждом слое, связи слоев между собой), выбирать один из реализованных алгоритмов обучения [19] (метод наискорейшего спуска, метод сопряженных градиентов, RPROP, Quickprop), задавать дополнительные параметры обучения и начальные данные.

В целях оптимизации работы, передача данных между оперативной памятью и видеопамятью в библиотеке сведена к минимуму. Хост-приложение, по своей сути, в процессе обучения нейронной сети лишь осуществляет добавление команд в очередь выполнения графического процессора и изредка сравнивает значение среднеквадратичной погрешности обучения с целевым значением для прекращения процесса обучения. Сравнение среднеквадратичной погрешности с целевым значением происходит не каждую итерацию, так как это требует ожидания данных от графического процессора, что приводит к остановке очереди выполнения.

В целом, алгоритм процесса обучения состоит из следующих шагов:

1. Сконфигурировать нейронную сеть, сформировать обучающие выборки и выбрать алгоритм и параметры обучения.

2. Скопировать данные обучающих выборок в видеопамять.

3. Для каждой обучающей выборки:

3.1. Вычислить нейронную сеть в прямом направлении.

3.2. С помощью сопряженного графа определить компоненты вектора градиента для каждой межнейронной связи.

3.3. Модифицировать веса межнейронных связей в соответствии с градиентом и параметрами обучения.

3.4. Посчитать среднеквадратичную погрешность.

3.5. Модифицировать параметры обучения в соответствии с алгоритмом обучения.

Следует отметить, что шаги 3.1. – 3.5. выполняются полностью на графическом процессоре. Центральный процессор осуществляет только добавление команд в очередь выполнения, но никак не контролирует момент исполнения этих команд.

Для анализа производительности библиотеки был проведен ряд тестов. В качестве основной метрики производительности была выбрана метрика среднего времени, затрачиваемого на модификацию веса одной межнейронной связи, вычисляемого по формуле (3):

$$\tau = \frac{T}{N \cdot W}, \qquad (3)$$

где τ – среднее время модификации одного веса межнейронной связи, T – общее время обучения, N – количество итераций обучения, W – общее количество весов нейронной сети.

В ходе выполненных тестов установлено, что среднее время модификации одного веса межнейронной связи значительно уменьшалось с увеличением числа нейронов и связей и стабилизировалось на отметке в 5-7 наносекунд для 4-хслойной сети с общим количеством нейронов порядка 1000 и общим количество связей порядка $3 \cdot 10^5$. При этом тестовый компьютер имел следующую конфигурацию: CPU – Intel Core 2 Duo, RAM – 2GB, GPU – GeForce 8800GT.

Был также проведен сравнительный анализ производительности с другими библиотеками для работы с нейронными сетями. В качестве кандидата для сравнения была выбрана одна из самых быстрых и оптимизированных библиотек на сегодняшний день – FANN (Fast Artificial Neural Network Library) [20]. Эта библиотека использует вычисления только на центральном процессоре. FANN также позволяет вывести метрику среднего времени, затрачиваемого на модификацию веса одной межнейронной связи. Полученные данные свидетельствуют о том, что пиковая производительность библиотеки FANN в 10–12 наносекунд достигается при общем количестве нейронов около 100. При дальнейшем увеличении размеров нейронной сети, производительность начинает уменьшаться. Это прежде всего связано с тем, что центральный процессор не может

эффективно обрабатывать большие объемы данных, так как они не помещаются полностью в кэш центрального процессора.

Из полученных результатов можно сделать вывод о том, что использование графического процессора при вычислении и обучении нейронной сети оказывается чрезвычайно эффективным при больших размерах сети, в отличии от реализации на центральном процессоре.

ЗАКЛЮЧЕНИЕ

Перенос вычислений с центрального процессора на графический процессор может существенно увеличить скорость выполнения ряда вычислительноемких задач.

Однако, не всякая реализация алгоритма на графическом процессоре дает прирост в скорости по сравнению с реализацией на центральном процессоре. Основными критериями, определяющими будет ли реализация на GPU быстрее реализации на CPU, являются: а) наличие больших объемов данных и б) отсутствие зависимостей между отдельными элементами данных.

В целом, параллельные вычисления на GPU могут выполняться медленнее по двум основным причинам: ограничение вычислительной мощности GPU и ограничения, связанные с пересылаемыми объемами данных. Для использования графического процессора с максимальной эффективностью, необходимо определить, что конкретно вызывает задержки при вычислениях. В случае, если задержки связаны с вычислительной мощностью GPU, необходимо оптимизировать конкретные функции OpenCL. Последнее возможно, например, за счет использования препросчитанных данных на ранее выполненном этапе. В случае, если задержки связаны с объемами пересылаемых данных, необходимо использовать больше операций внутри одной функции OpenCL, избегая дополнительных этапов при расчетах.

Перенос реализации алгоритма на графический процессор обычно сопряжен с его значительной модификацией. Это в первую очередь связано с методикой при работе с памятью и дополнительными ограничениями, накладываемыми графическим процессором. Однако оптимизированная реализация алгоритма на графическом процессоре, как правило, способна существенно ускорить выполнение программы, что и продемонстрировано в данной работе.

СПИСОК ЛИТЕРАТУРЫ

1. Bart Pieters et al. Motion estimation for H.264/AVC on multiple GPUs using NVIDIA CUDA. Proc. SPIE, Vol. 7443, 74430X (2009)

2. *John E.* Stone et al. High Performance Computation and Interactive Display of Molecular Orbitals on GPUs and Multi-core CPUs. ACM International Conference Proceeding Series; Vol. 383 (2009)

3. *Hongsheng Li* "Actin Filament Tracking Based on Particle Filters and Stretching Open Active Contour Models", Int'l Conf. Medical Image Computing and Computer Assisted Intervention (MICCAI), 2009

4. *Carmine Clemente* PROCESSING OF SYNTHETIC APERTURE RADAR DATA WITH GPGPU.

5. *Joshua A.* Anderson, et al. General purpose molecular dynamics simulations fully implemented on graphics processing units, Journal of Computational Physics 227(10), May 2008, DOI 10.1016/j.jcp.2008.01.047

6. *Guzhva A., Dolenko S., Persiantsev I.* Multifold Acceleration of Neural Network Computations Using GPU. Artificial Neural Networks – ICANN 2009, DOI 10.1007/978-3-642-04274-4, 2009

7. *Govindaraju N. K., Gray J., Kumar R., Manocha D.* Gputerasort: High performance graphics coprocessor sorting for large database management. In SIGMOD, 2006.

8. *Govindaraju N. K., Lloyd B., Wang W., Lin M., Manocha D.* Fast computation of database operations using graphics processors. In SIGMOD, 2004.

9. *He B., Lu M., Yang K., Fang R., Govindaraju N. K., Luo Q., Sander P. V.* Relational query co-processing on graphics processors. In TODS, 2009.

10. *He B., Yang K., Fang R., Lu M., Govindaraju N. K., Luo Q., Sander P. V.* Relational joins on graphics processors. In SIGMOD, 2008.

11. Wenbin Fang, Bingsheng He, Qiong Luo: "Database Compression on Graphics Processors", PVLDB/VLDB 2010.

12. http://www.python.org

13. http://www.numpy.org

14. http://www.scipy.org

15. http://documen.tician.de/pyopencl

16. http://clyther.sourceforge.net/

17. http://deeplearning.net/software/theano/

18. Gaussian 03, Revision C.02, M. J. Frisch, G. W. Trucks, et al., Gaussian, Inc., Wallingford CT, 2004.

19. *Осовский С.* Нейронные сети для обработки информации/Пер. с польского И.Д. Рудинского. Изд-во: Финансы и статистика, 2002. – 344 с.

20. http://leenissen.dk/fann/

Запрягаев Сергей Александрович – д. ф.-м. н., проф. каф. цифровых технологий Воронежского государственного университета. Тел. (4732) 208-257. Email: zsa@main.vsu.ru

Карпушин Андрей Александрович – аспирант, кафедра цифровых технологий, Воронежский государственный университет. Тел. (4732) 208-257. Email: reven86@gmail.com

Zapryagaev S.A. – Doctor of Physics-math. Sciences, Professor of the dept. of digital technologies Voronezh State University. Tel. (4732) 208-257. Email: zsa@main.vsu.ru

Karpushin A.A. – Post-graduate student of the dept. of digital technologies Voronezh State University. Tel. (4732) 208-257. Email: reven86@gmail.com