МИНИСТЕРСТВО ВЫСШЕГО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФГБОУ ВО «ДАГЕСТАНСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

Для выполнения лабораторных работ по дисциплине

ТЕХНОЛОГИИ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ

Для бакалавров направления подготовки

01.03.02 - Прикладная математика и информатика

Махачкала 2021

Методические указания для выполнения лабораторных занятий по дисциплине технологии параллельного программирования. Для бакалавров направления подготовки бакалавров 01.03.02 - Прикладная математика и информатика. - Махачкала: ДГТУ, 2021, 32 с.

Методические указания для выполнения лабораторных занятий по дисциплине технологии параллельного программирования.

Цель данных методических указаний – объяснить студентам техническую сторону выполнения контрольных работ и изложить некоторые общие рекомендации по изучению программного материала по дисциплине технологии параллельного программирования.

Даны методические указания для оказания помощи студентам при изучении дисциплины технологии параллельного программирования.

Составитель: доцент кафедры ПМиИ Пиняскин В.В.

Рецензенты:

к.т.н., доцент каф. ПМиИ ДГТУ Канаев М.М.

Заместитель генерального директора по информационным технологиям ООО «Дагестан-Парус» к.ф-м.н Карапац А.Н.

Печатается по решению Ученого совета Дагестанского государственного технического университета от «___» _____. 2021г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью лабораторных работ является изучение параллельных вычислительных систем, освоение средств, методов и алгоритмов параллельного программирования в средах поддержки параллельных вычислений OpenMP, NVIDIA CUDA и MPI для многоядерных рабочих станций, графических процессоров, вычислительных сетей и кластеров.

В этом пособии содержится:

– краткое введение в архитектуру параллельных вычислительных систем;

– введение в проблематику параллельного программирования для параллельных систем различных классов («множественный поток команд, множественный поток данных» (МКМД) с общей памятью, МКМД с распределенной памятью, и «одиночный поток команд, множественный поток данных»);

– основные сведения об используемых технологиях параллельного программирования;

– методические указания к лабораторным работам:

1. Параллельное программирование для систем с общей памятью с использованием технологии OpenMP.

2. Параллельное программирование для графического процессора в среде NVidia CUDA.

Архитектура параллельных вычислительных систем

Применение параллельных вычислительных систем (ПВС) и суперкомпьютеров является стратегическим направлением развития вычислительной техники [1]. Это вызвано не только принципиальным ограничением максимально возможного быстродействия обычных последовательных ЭВМ, но и практически постоянным наличием вычислительных задач, для решения которых возможностей существующих средств вычислительной техники всегда оказывается недостаточно. Точнее, почти сразу после осознания того факта, что созданные к настоящему моменту компьютеры не в состоянии решить за приемлемое время многие задачи. Выход из создавшегося положения напрашивался сам собой. Если один компьютер не справляется с решением задачи за нужное время, то попробуем взять два, три, десять компьютеров и заставим их *одновременно* работать над различными частями общей задачи, надеясь получить соответствующее ускорение.

Существует множество способов организации параллельно работающих вычислительных систем [1, 2]. Параллельность вычислений, когда в один и тот же момент выполняется одновременно несколько операций обработки данных, осуществляется, в основном, за счет введения избыточности функциональных устройств (многопроцессорности) [3]. В этом случае можно достичь ускорения процесса решения вычислительной задачи, если осуществить разделение применяемого алгоритма на информационно независимые части и организовать выполнение каждой части вычислений на разных функциональных устройствах

(процессорах, ядрах, …). Подобный подход позволяет выполнять необходимые вычисления с меньшими затратами времени. Возможность получения максимально возможного ускорения ограничивается (по крайней мере, в принципе) только числом имеющихся процессоров и количеством параллено выполняющихся частей в вычислениях.

При рассмотрении проблемы организации параллельных вычислений следует различать следующие возможные режимы выполнения независимых частей программы:

- многозадачный режим (режим разделения времени), при котором для выполнения нескольких процессов используется единственный процессор. Данный режим является псевдопараллельным, когда активным (исполняемым) может быть один, единственный процесс, а все остальные процессы находятся в состоянии ожидания своей очереди. Применение режима разделения времени может повысить эффективность организации вычислений (например, если один из процессов не может выполняться из-за ожидания вводимых данных, то процессор может быть задействован для выполнения другого, готового к исполнению процесса). Кроме того, в данном режиме проявляются многие эффекты параллельных вычислений (необходимость взаимоисключения и синхронизации процессов и др.), и, как результат, этот режим может быть использован при начальной подготовке параллельных программ;
- параллельное выполнение, когда в один и тот же момент может выполняться несколько команд обработки данных. Такой режим вычислений может быть обеспечен не только при наличии нескольких процессоров, но и при помощи конвейерных и векторных обрабатывающих устройств;
- распределенные вычисления; данный термин обычно применяют для указания параллельной обработки данных, при которой используется несколько обрабатывающих устройств, достаточно удаленных друг от друга, в которых передача данных по линиям связи приводит к существенным временным задержкам. Как результат, эффективная обработка данных при таком способе организации вычислений возможна только для параллельных алгоритмов с низкой интенсивностью потоков межпроцессорных передач данных. Перечисленные условия являются характерными, например, при организации вычислений в многомашинных вычислительных комплексах, образуемых объединением нескольких отдельных ЭВМ с помощью каналов связи локальных или глобальных информационных сетей.

При разработке параллельных алгоритмов решения задач вычислительной математики принципиальным моментом является анализ эффективности использования параллелизма [4], состоящий обычно в оценке получаемого *ускорения* процесса вычисления (сокращения времени решения задачи). Формирование подобных оценок ускорения может осуществляться применительно к выбранному вычислительному алгоритму *(оценка эффективности распараллеливания конкретного алгоритма)*. Другой важный подход может состоять в построении оценок максимально возможного ускорения процесса получения решения задачи конкретного типа *(оценка эффективности параллельного способа решения задачи)*.

Архитектура параллельных компьютеров с самого начала их создания и применения развивалась в самых различных направлениях. Большое разнообразие вычислительных систем породило естественное желание ввести для них какую-то классификацию. На основе числа этих потоков выделяется четыре класса архитектур.

1. ОКОД (SISD – Single Instruction Single Data) – единственный поток команд и единственный поток данных. По сути дела это классическая машина фон Неймана. К этому классу относятся все однопроцессорные системы.

2. ОКМД (SIMD – Single Instruction Multiple Data) – единственный поток команд и множественный поток данных. Типичными представителями являются матричные компьютеры, в которых все процессорные элементы выполняют одну и ту же программу, применяемую к своим (различным для каждого процессора) локальным данным. Иногда к этому классу относят и векторно-конвейерные компьютеры, если каждый элемент вектора рассматривать как отдельный элемент потока данных. В настоящее время этот класс пополнился графическими процессорами, позволившими существенно увеличить достижимый порог производительности при сравнительно невысокой стоимости.

3. МКОД (MISD – Multiple Instruction Single Date) – множественный поток команд и единственный поток данных. Автор классификации не смог привести ни одного примера реально существующей системы, работающей на этом принципе. Иногда в качестве представителей такой архитектуры называют векторно-конвейерные компьютеры, однако такая точка зрения не получила широкой поддержки.

4. МКМД (MIMD – Multiple Instruction Multiple Date) – множественный поток команд и множественный поток данных. К этому классу относятся практически все современные многопроцессорные системы.

В классе МКМД наибольший интерес представляют собой системы с общей памятью и системы с распределенной памятью.

ОБЩАЯ ХАРАКТЕРИСТИКА ЛАБОРАТОРНЫХ РАБОТ

В каждой лабораторной работе нужно разработать параллельную программу для решения одной из следующих задач:

1. Решение системы линейных алгебраических уравнений.
2. Сортировка массива данных различными методами.
3. Перемножение матриц.
4. Вычисление корней алгебраического уравнения одной переменной большой размерности.
5. Разложение большого числа на простые множители.
6. Вычисление обратной матрицы.
7. Поиск кратчайшего пути на графе.
8. Умножение полиномов многих переменных.
9. Численное дифференцирование.
10. Численное интегрирование.
11. Численное моделирование.

Рекомендуется задача № 1 – решение системы линейных алгебраических уравнений (СЛАУ) большой размерности. Матрица коэффициентов СЛАУ выдается преподавателем или создается самостоятельно с использованием программы GenSLAU, формирующей либо текстовый, либо бинарный файл (по выбору студента). И в том и в другом случае первым элементом файла является размерность матрицы N, следующие N групп элементов содержат N+1 число с плавающей точкой, из которых первые N чисел – это коэффициенты соответствующей строки матрицы, а последнее – свободный член этой строки. В бинарном формате каждым элементом является 4-х байтное вещественное число, разделителей нет. В текстовом формате разделителем между элементами является символ пробела (с кодом 0x20), а в качестве разделителя между группами элементов используется последовательность символов с кодами 0x0D, 0x0A (перевод строки, возврат каретки).На каждую работу студенту выдается рекомендуемый индивидуальный вариант разбиения матрицы коэффициентов СЛАУ между ветвями параллельной программы.

ЛАБОРАТОРНАЯ РАБОТА 1

Тема: «Параллельное программирование с использованием OpenMP»

Цель работы: –изучить технологию OpenMP и основы разработки параллельных программ для многоядерных процессоров, написать и отладить на языке С параллельную программу для решения поставленной задачи на системе с общей памятью.

Основные сведения об OpenMP

Технология OpenMP в настоящее время является одним из наиболее популярных средств параллельного программирования для компьютеров с общей памятью, базирующейся на традиционных последовательных языках программирования и использовании специальных комментариев. За основу берётся последовательная программа, а для создания её параллельной версии пользователю предоставляется набор директив, функций и переменных окружения. Технология OpenMP нацелена на то, чтобы пользователь имел один и тот же вариант программы как для параллельного, так и для последовательного режима выполнения. Параллелизм в OpenMP реализуется с помощью многопоточности. При запуске программы создается единственный «главный» (master) поток, который затем создает набор «подчиненных» (slave) потоков, и вычисления распределяются между всеми потоками. Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами (ядрами), причём количество процессоров/ядер не обязательно должно быть больше или равно количеству потоков. Другими словами, потоки параллельной программы могут обычным образом конкурировать между собой за время процессора/ядра.

Важным достоинством технологии OpenMP является возможность реализации так называемого инкрементального программирования, когда программист постепенно находит участки в программе, содержащие ресурс параллелизма, с помощью предоставляемых механизмов делает их параллельными, а затем переходит к анализу следующих участков. Таким образом, распараллеленные участки постепенно охватывают всё большую часть программы. Этот подход значительно облегчает процесс адаптации последовательных программ к параллельным компьютерам, а также отладку и оптимизацию программ.

Модель исполнения параллельной программы, подготовленной с помощью технологии OpenMP, можно сформулировать следующим образом:

- Программа содержит набор последовательных и параллельных областей (или секций или регионов).
- В начальный момент времени создается главный поток, выполняющий впоследствии все последовательные области программы.
- При входе в параллельную область главным потоком выполняется операция *fork,* порождающая совокупность подчиненных потоков. Каждый поток имеет свой уникальный числовой идентификатор (главному потоку соответству-

ет 0). При распараллеливании циклов все параллельные потоки исполняют один и тот же код, но с разными данными. В общем случае потоки могут исполнять различные фрагменты кода.

– При выходе из параллельной области всеми потоками выполняется операция *join*. Завершается выполнение всех потоков, кроме главного.

На рис. 1. проиллюстрировано создание и исполнение двух параллельных областей. В первой области все потоки приходят к моменту выхода из нее практически одновременно, во второй – в разные моменты времени (кАк правило, это более соответствует реальной картине).

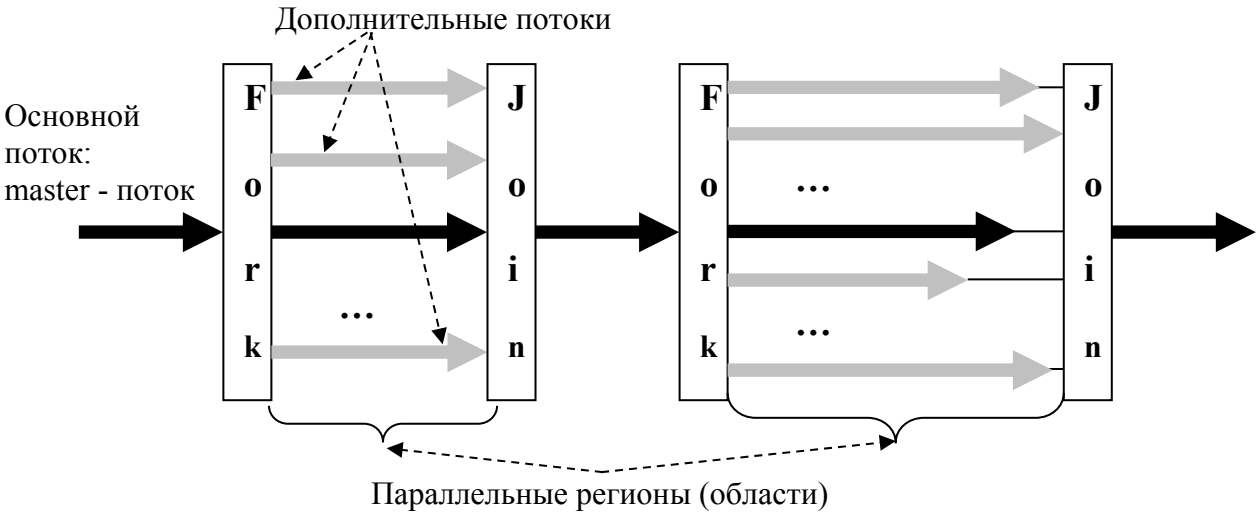

Рис. 1. Создание двух параллельных регионов

Обычно допускается возможность образования вложенных параллельных областей, когда поток, созданный как дополнительный, становится master-потоком для новой параллельной области, как показано на рис. 2.

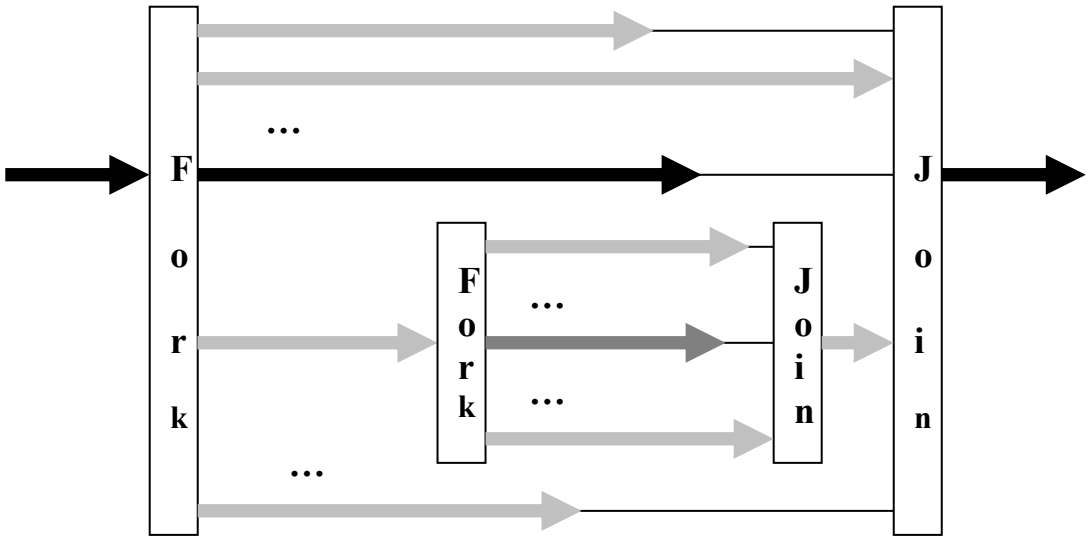

Рис. 2. Создание вложенных параллельных регионов

Под OpenMP понимается совокупность следующих компонент:

– *Директивы компилятора* – используются для создания потоков, распределения работы между потоками и их синхронизации. Директивы включаются программистом в исходный текст программы.
– *Подпрограммы библиотеки времени выполнения* – используются для установки и определения атрибутов потоков. Вызовы этих подпрограмм включаются программистом в исходный текст.
– *Переменные окружения* – используются для управления поведением параллельной программы. Переменные окружения задаются для среды выполнения параллельной программы соответствующими средствами операционной системы.

Использование директив компилятора и подпрограмм библиотеки времени выполнения подчиняется правилам, которые различаются для разных языков программирования. Совокупность таких правил для одного языка программирования называется *привязкой к языку.* Технология OpenMP создана и поддерживается для языков C, C++ и Fortran. В этой лабораторной работе предполагается использование языка C/C++ в среде разработки Microsoft Visual Studio (начиная с версии 2005). Для того, чтобы ее компилятор нужным образом реагировал на директивы OpenMP и строил параллельную программу, в командную строку его запуска должна быть включена опция /openmp (например, путем включения флажка «OpenMP support» в свойствах проекта на закладке Configuration properties\C/C++\Language).

Для того, чтобы в программе на языке C/C++ стали доступны возможности технологии OpenMP, в нее нужно включить заголовочный файл omp.h:
#include <omp.h>

Если эту программу транслировать компилятором, не поддерживающим технологию OpenMP, то она будет построена в обычном последовательном (однопоточном) варианте, поскольку все директивы, задающие распараллеливание, оформляются в виде так называемых прагм (рекомендаций), и просто игнорируются такими компиляторами.

В программах на языке C/C++ все прагмы, имена функций и переменных окружения OpenMP начинаются со строки «omp». Формат директивы:
#pragma omp директива [опция_1[, опция_2, ...]]

Каждая директива вместе со всеми ее опциями обязательно должна занимать ровно одну строку текста. Действие некоторых директив распространяется только на один оператор (или блок операторов, заключенных в фигурные скобки), непосредственно следующий за директивой в тексте программы. Для таких директив будет указан
<структурный блок кода>.

Перечень **директив OpenMP** включает в себя:
1. Директива задания параллельно выполняемой секции:
#pragma omp parallel [опция_1[, опция_2, ...]]
<структурный блок кода>

С помощью опций этой директивы можно указать:
– требуемое количество потоков n (*num_threads(n)*);

– условие, при котором параллельная область действительно создается (*if(условие)*);
– список общих переменных для всех потоков данной секции (*shared(список переменных)*);
– список переменных, которые будут локальными в каждом потоке секции (*private(список переменных)*), причем их начальные значения не будут определены;
– список переменных, которые будут локальными в каждом потоке секции (*firstprivate(список переменных)*), причем в качестве их начальных значений будут установлены значения одноименных переменных из главного потока;
– способ назначения класса памяти *default(shared|none)* всем переменным потоков, которым класс не назначен явно с помощью опции *shared* (слово *none* означает, что класс памяти всех локальных переменных должен быть задан явно); в реализациях для языка Fortran могут назначаться классы *private* и *firstprivate*;
– список переменных, объявленных директивой *threadprivate* (см. ниже), которые при входе в параллельную секцию инициализируются значениями соответствующих переменных в потоке-мастере;
– оператор сведения и список общих переменных *reduction(оператор : список переменных)*; для каждой указанной в списке переменной создаются локальные копии в каждом потоке; локальные копии инициализируются соответственно типу оператора (для аддитивных операций ноль или его аналоги, для мультипликативных операций единица или её аналоги); над всеми локальными копиями каждой переменной после завершения параллельной секции будет выполнен заданный оператор сведения, результаты будут занесены в одноименные общие переменные; в качестве оператора можно указывать: +, –, *, &, |, ^, &&, ||.

2. Директива определения цикла, итерации которого нужно распределить между параллельно выполняемыми потоками:

#pragma omp for [опция_1[, опция_2, ...]]
<структурный блок кода>

С использованием опций директивы for можно указать:

– список переменных, которые будут локальными в каждом потоке секции (*private(список переменных)*), причем их начальные значения не будут определены;
– список переменных, которые будут локальными в каждом потоке секции (*firstprivate(список переменных)*), причем в качестве их начальных значений будут установлены значения одноименных переменных из главного потока;
– список переменных главного потока (*lastprivate(список переменных)*), которым будут присвоены значения, полученные при выполнении последней итерации цикла;
– оператор сведения и список общих переменных *reduction(оператор : список переменных)*; для каждой указанной в списке переменной создаются локальные копии в каждом потоке; локальные копии инициализируются соответственно типу оператора (для аддитивных операций ноль или его аналоги,

для мультипликативных операций единица или её аналоги); над всеми локальными копиями каждой переменной после завершения параллельной секции будет выполнен заданный оператор сведения, результаты будут занесены в одноименные общие переменные; в качестве оператора можно указывать: +, –, *, &, |, ^, &&, ||;

- способ распределения итераций цикла между потоками параллельной секции (*schedule(type[, chunk])*); параметр *chunk* этой опции определяет количество итераций на один поток (по умолчанию 1), параметр *type* указывает тип распределения и может иметь значения:
  - *static* – статический, т.е. при компиляции,
  - *dynamic* – динамический, т.е. при выполнении,
  - *guided* - динамический с уменьшением количества итераций на поток от начального, определяемого автоматически, до значения *chunk*,
  - *auto* – способ распределения выбирается компилятором или исполняющей системой,
  - *runtime* – способ распределения задается специальной переменной окружения операционной системы;
- возможность (опция *ordered*) появления в теле цикла директивы *ordered*, которая требует исполнения охваченного ею блока операторов в точности в той же последовательности, какая реализуется в последовательной версии данного цикла;
- отмену (*nowait*) неявной барьерной синхронизации потоков, достигших конца выполнения своей части итераций цикла (при отсутствии этой опции участок программы после цикла будет выполняться только тогда, когда все потоки выполнят все свои итерации;
- глубину n вложенных друг в друга циклов (*collapse(n)*), пространство итераций которых подлежит распределению между параллельными потоками (доступно только в реализации 2.5 технологии OpenMP);

Директивы *parallel* и *for* можно объединять в одну директиву, в которой можно указывать опции как директивы *parallel*, так и директивы *for*:

#pragma omp parallel for [опция_1[, опция_2, ...]]

3. Директива, указывающая на необходимость исполнения охватываемого ею участка кода (части тела цикла) в той последовательности, в которой он выполняется в чисто последовательном режиме

#pragma omp ordered

<структурный блок кода>

4. Директива задания области нециклического параллелизма (участки структурного блока кода, которые могут выполняться параллельно, выделяются с помощью описываемой далее директивы *section*):

#pragma omp sections [опция_1[, опция_2, ...]]

<структурный блок кода>

В качестве опций этой директивы можно указывать *private, firstprivate, lastprivate, reduction* и *nowait*, имеющие в точности такой же синтаксис и тот же смысл, что и у директивы *for.*

5. Директива определения участка нециклического кода для одного потока:

#pragma omp section

С помощью этой директивы внутри участка кода, охваченного директивой *sections*, выделяются отдельные фрагменты для исполнения параллельными потоками. Перед самым первым фрагментом участка нециклического кода директиву *section* можно не указывать.

6. Директива объявления списка локальных переменных потоков

#pragma omp threadprivate(список переменных)

Эта директива позволяет сделать локальные копии для статических переменных языка C/C++ (и COMMON-блоков языка Фортран), которые по умолчанию являются общими.

7. Директива создания отдельной независимой задачи (начиная с версии 2.5 OpenMP):

#pragma omp task [опция_1[, опция_2, ...]]
<структурный блок кода>

Текущий поток создает в качестве задачи ассоциированный с директивой блок операторов. Эта задача может выполняться немедленно после создания или быть отложенной на неопределённое время и выполняться по частям. Размер таких частей, а также порядок выполнения частей разных отложенных задач определяется реализацией OpenMP.

Возможные опции:

– *if*, *default*, *private*, *firstprivate* и *shared* – имеют такой же синтаксис и смысл, как и в предыдущих директивах;
– *untied* – означает, что в случае откладывания задача может быть продолжена любым потоком из числа выполняющих данную параллельную область; если данная опция не указана, то задача может быть продолжена только породившим её потоком;

8. Директива ожидания потоком завершения всех независимых задач, запущенных именно из данного потока:

#pragma omp taskwait

9. Директива, требующая исполнения охваченного ею участка кода в точности одним (любым) потоком:

#pragma omp single [опция_1[, опция_2, ...]]
<структурный блок кода>

Опции директивы позволяют указать:

– списки переменных *private* и *firstprivate*, имеющие такой же синтаксис и семантику, как в ранее описанных директивах;
– список переменных (*copyprivate(список переменных)*), значения которых после выполнения структурного блока, заданного директивой *single*, будут занесены во все одноименные локальные переменные (*private* и *firstprivate*), заданные для охватывающей параллельной секции; эта опция не может использоваться совместно с опцией *nowait*; переменные списка не должны быть перечислены в опциях *private* и *firstprivate* данной директивы *single*;
– отмену (*nowait*) неявной барьерной синхронизации потоков, достигших точки, следующей за блоком операторов, охваченным директивой *single*; при отсутствии этой опции такая синхронизация выполняется.

10. Директива, требующая исполнения охваченного ею участка кода главным потоком программы:
#pragma omp master
<структурный блок кода>

Этот структурный блок будет выполнен только главным потоком программы. Остальные потоки просто пропускают данный участок и продолжают работу с оператора, расположенного следом за ним. Неявной синхронизации данная директива не предполагает.

11. Директива явной барьерной синхронизации:
#pragma omp barrier

Потоки, выполняющие текущую параллельную секцию, дойдя до этой директивы, останавливаются и ждут, пока все потоки не дойдут до этой точки программы, после чего разблокируются и продолжают работать дальше. Кроме того, для разблокировки необходимо, чтобы все синхронизируемые потоки завершили все порождённые ими задачи (директивы *task* и *taskwait*).

12. Директива, объявляющая критическую секцию – участок параллельной области программы, который единовременно может выполняться не более, чем одним потоком:
#pragma omp critical [(<имя_критической_секции>)]
<структурный блок кода>

Если критическая секция уже выполняется каким-либо потоком, то все остальные, выполнившие эту директиву для секции с данным именем, будут заблокированы, пока вошедший в секцию поток не закончит выполнение этого блока кода. Как только это произойдет, один из заблокированных потоков войдет в критическую секцию. Если на входе в критическую секцию стояло несколько потоков, то случайным образом выбирается один из них, а остальные заблокированные продолжают ожидание.

Все неименованные критические секции условно ассоциируются с одним и тем же именем. Все критические секции, имеющие одно и тоже имя, рассматриваются как одна секция, даже если находятся в разных параллельных областях. Побочные входы и выходы из критической секции запрещены.

13. Директива блокировки доступа к общей переменной из левой части оператора присваивания на время выполнения всех действий с этой переменной в данном операторе:
#pragma omp atomic
<оператор присваивания>

Атомарной (блокирующей выполнение остальных потоков) является только работа с переменной из левой части оператора присваивания, при этом вычисления в его правой части, использующие другие переменные, не обязаны быть атомарными.

14. Директива актуализации значений переменных потока:
#pragma omp flush [(список переменных)]

Значения всех переменных (или переменных из списка, если он задан), временно хранящиеся в регистрах и кэш-памяти текущего потока, заносятся в

основную память; следовательно, все изменения переменных, сделанные потоком во время работы, станут видимы остальным потокам; если какая-то информация хранится в буферах вывода, то буферы будут сброшены на внешние носители и т.п. Эти действия производятся только с данными выполнившего директиву потока, а данные, изменявшиеся другими потоками, не затрагиваются. До полного завершения этих действий никакие другие операции с участвующими в директиве *flush* переменными не могут выполняться. Поэтому выполнение директивы *flush* без списка переменных может повлечь значительные накладные расходы. Если в данный момент нужна гарантия согласованного представления не всех, а лишь некоторых переменных, то именно их следует явно перечислить в директиве списком.

Переменные окружения

OMP_NUM_THREADS – количество потоков в новой параллельной области.

OMP_NESTED – возможность создания вложенных параллельных областей.

OMP_MAX_ACTIVE_LEVELS – максимальная глубина вложенности параллельных областей.

OMP_DYNAMIC – возможность динамического определения количества потоков (имеет больший приоритет, чем OMP_NUM_THREADS).

OMP_THREAD_LIMIT – максимальное количество потоков программы.

OMP_SCHEDULE – способ распределения итераций цикла для значения *runtime* опции *schedule* директивы *for*.

OMP_STACKSIZE – размер стека каждого потока.

OMP_WAIT_POLICY – выделять или нет кванты процессорного времени ждущим потокам.

Библиотека функций OpenMP.

double omp_get_wtime(void); – возвращает текущее время.

double omp_get_wtick(void); – возвращает размер тика таймера.

int omp_get_thread_num(void); – возвращает номер потока.

int omp_get_max_threads(void); – возвращает максимально возможное количество потоков для следующей параллельной области.

int omp_get_thread_limit(void); – возвращает значение OMP_THREAD_LIMIT.

void omp_set_num_threads(int num); – устанавливает новое значение переменной OMP_NUM_THREADS.

int omp_get_num_procs(void); – возвращает количество процессоров/ядер.

int omp_get_dynamic(void); – возвращает значение OMP_DYNAMIC.

void omp_set_dynamic(int num); – устанавливает новое значение переменной OMP_DYNAMIC.

int omp_get_nested(void); – возвращает значение OMP_ NESTED.

void omp_set_nested(int nested); – устанавливает новое значение переменной OMP_ NESTED.

int omp_in_parallel(void); – возвращает 0, если функция вызвана из последовательной области и 1, если из параллельной.

int omp_get_max_active_levels(void); – возвращает значение переменной OMP_MAX_ACTIVE_LEVELS

void omp_set_max_active_levels(int max);); – устанавливает новое значение OMP_MAX_ACTIVE_LEVELS

int omp_get_level(void); – возвращает глубину вложенности параллельных областей.

int omp_get_ancestor_thread_num(int level); – возвращает номер потока, породившего текущую параллельную область.

int omp_get_team_size(int level); – возвращает для заданного параметром level уровня вложенности параллельных областей количество потоков, порождённых одним родительским потоком.

int omp_get_active_level(void); – возвращает количество вложенных параллельных областей, обрабатываемых более чем одним потоком.

Следующая группа функций используется для синхронизации параллельно выполняющихся потоков с помощью так называемых замков (lock).

*void omp_init_lock(omp_lock_t *lock);* – создать (инициализировать) простой замок.

*void omp_init_nest_lock(omp_nest_lock_t *lock);* – создать замок с множественными захватами.

*void omp_destroy_lock(omp_lock_t *lock);* – уничтожить простой замок (перевести в неинициализированное состояние).

*void omp_destroy_nest_lock(omp_nest_lock_t *lock);* – уничтожить множественный замок (перевести в неинициализированное состояние).

*void omp_set_lock(omp_lock_t *lock);* – захватить простой замок (если замок уже захвачен другим потоком, то данный поток переводится в ждущее состояние).

*void omp_set_nest_lock(omp_nest_lock_t *lock);* – захватить множественный замок (если замок уже захвачен другим потоком, то данный поток переводится в ждущее состояние; если замок захвачен этим же потоком, то увеличивается счетчик захватов).

*void omp_unset_lock(omp_lock_t *lock);* – освободить простой замок (если есть потоки, ждущие его освобождения, то один из них, выбираемый случайным образом, захватывает этот замок и переходит в состояние выполнения).

*void omp_unset_nest_lock(omp_lock_t *lock);* – уменьшить на 1 количество захватов множественного замка (если количество захватов стало равно нулю и есть потоки, ждущие его освобождения, то один из них, выбираемый случайным образом, захватывает этот замок и переходит в состояние выполнения).

*int omp_test_lock(omp_lock_t *lock);* – попытаться захватить простой замок (если попытка не удалась, т.е. замок уже захвачен другим потоком, то возвращается 0, иначе возвращается 1).

*int omp_test_nest_lock(omp_lock_t *lock);* – попытаться захватить множественный замок (если попытка не удалась, т.е. замок уже захвачен другим потоком, то возвращается 0, иначе возвращается новое значение счетчика захватов).

Этапы выполнения работы

1. Изучить технологию OpenMP и средства среды разработки (наприме – Microsoft Visual Studio или Intel Parallel Studio) используемые при разработке и отладке параллельных программ на основе этой технологии.
2. Разработать последовательный алгоритм решения задачи, написать и отладить последовательную программу.
3. Модифицировать последовательную программу путем вставки директив и вызовов функций OpenMP, получить и отладить параллельную программу решения задачи.
4. Проанализировать результаты решения задачи для последовательного и параллельного вариантов программы, оценить отклонения полученных решений друг от друга и от эталонного (если оно известно), объяснить причины отклонений.
5. Измерить и оценить временные характеристики последовательной и параллельной программы, объяснить полученные соотношения.
6. Подготовить в электронном виде, сдать преподавателю и защитить отчет по работе.

Требования к содержанию отчета

Отчет по данной лабораторной работе включается в общий отчет по лабораторному практикуму (см. лабораторную работу 4, п. 4.3.). Отчет должен содержать:

- цель работы;
- краткое описание технологии OpenMP и многопроцессорной (многоядерной) системы с общей памятью, на которой решается поставленная задача;
- описание используемого метода распараллеливания;
- описание параллельного алгоритма решения задачи;
- листинг программы;
- результаты анализа решения задачи последовательным и параллельным вариантом программы;
- результаты измерений временных характеристик для разного количества процессоров (ядер) и анализ эффективности параллельной программы.

Контрольные вопросы

1. Перечислите составные части технологии OpenMP.
2. С помощью какой директивы (директив) создаются новые параллельные области программы?
3. Что такое критическая секция программы?
4. Каким образом можно установить нужное количество потоков для создания очередной параллельной области?
5. Как обеспечить выполнение фрагмента параллельной области только главным потоком?

6. Какая опция директивы OpenMP *for* используется для указания способа распределения итераций цикла между потоками параллельной области?
7. Что такое *deadlock*? Каким правилам нужно следовать, чтобы избежать возможности попадания параллельной программы в *deadlock*?
8. Что такое сведение данных? Какие опции и в каких директивах используются для выполнения сведения?
9. Что делает директива OpenMP *threadprivate*?
10. Как обеспечить выполнение фрагмента параллельной области потоком с максимальным номером в данной параллельной области?
11. Для чего используется опция *firstprivate*? Чем она отличается от опций *private* и *lastprivate*?
12. Что такое вложенная параллельная область программы? В каких случаях ее нельзя создать?
13. Что такое неявная барьерная синхронизация? С помощью каких средств ее можно отменить?
14. Для чего используются директивы OpenMP *sections* и *section*? Что делает каждая из этих директив?
15. Перечислите все средства синхронизации потоков в OpenMP.
16. Перечислите возможные способы распределения итераций цикла между потоками.
17. Что делает директива OpenMP *atomic*?
18. В чем состоит различие между общими и локальными переменными потока?
19. С помощью каких средств можно ограничить глубину вложенности параллельных областей программы?
20. От чего зависит равномерность загрузки процессоров/ядер системы с общей памятью?
21. Каким образом функция, вызываемая из параллельной программы, может выяснить, в последовательной или параллельной области она выполняется?
22. Как обеспечить выполнение фрагмента параллельной области в точности одним потоком?
23. Какое значение будет иметь переменная *count* в результате выполнения фрагмента параллельной программы:

```
int count = 0;
#pragma omp parallel for
for(int i = 0; i<10; i++){
        count++;
}
```

24. Могут ли два потока одновременно захватить один множественный замок?
25. Для чего может использоваться опция *reduction* в некоторых директивах OpenMP?

ЛАБОРАТОРНАЯ РАБОТА 2

Тема: «Параллельное программирование в среде NVidia CUDA»

Цель работы: – изучить архитектуру CUDA и основы разработки параллельных программ для совместного использования CPU и GPU, написать и отладить на языке С параллельную программу решения поставленной задачи на графическом процессоре.

Основные сведения об архитектуре CUDA

CUDA (Compute Unified Device Architecture) [12-14] представляет собой совокупность аппаратного (графический процессор – GPU) и программного обеспечения, предоставляющего возможность подготовки и исполнения программ с очень высокой степенью параллелизма.

GPU есть специализированное вычислительное устройство, которое является сопроцессором к основному процессору компьютера (CPU), обладает собственной памятью и возможностью параллельного выполнения огромного количества (тысячи и десятки тысяч) отдельных нитей (потоков) обработки данных согласно модели ОКМД. Логическая структура связанных двухядерного центрального и графического процессора показана на рис. 3.

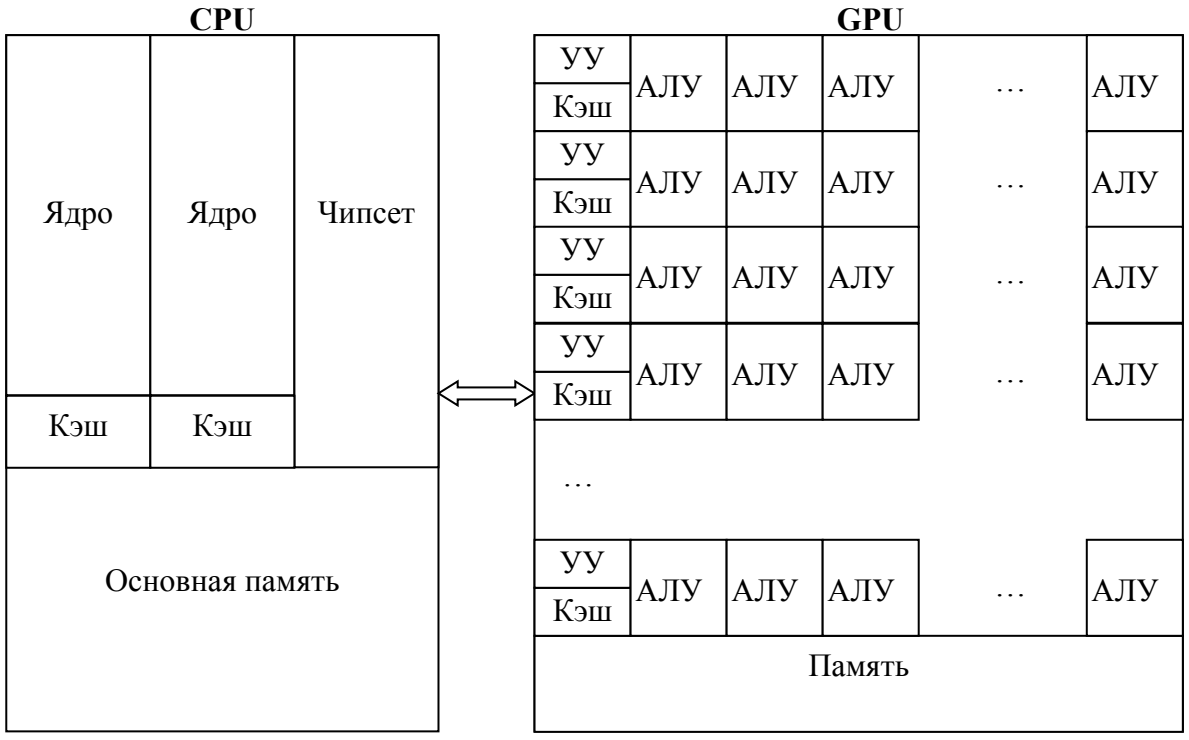

Рис. 3. Двухядерный CPU и подключенный к нему GPU

Технология OpenMP, рассмотренная в работе 1 и CUDA вполне совместимы и могут использоваться в одной параллельной программе для достижения максимально возможного ускорения.

Между потоками, выполняемыми на CPU и потоками, выполняемыми графическим процессором, есть принципиальные различия:

- нити, выполняемые на GPU, обладают крайне низкой «стоимостью» – их создание и управление ими требует минимальных ресурсов (в отличии от потоков CPU)
- для эффективной утилизации возможностей GPU нужно использовать многие тысячи отдельных нитей (для CPU обычно бывает трудно организовывать более, чем 10-20 потоков)

Приложения, использующие возможности CUDA для параллельной обработки данных, взаимодействуют с GPU через программные интерфейсы, называемые CUDA-runtime или CUDA-driver (обычно интерфейсы CUDA-runtime и CUDA-driver одновременно не используются, хотя это в принципе возможно).

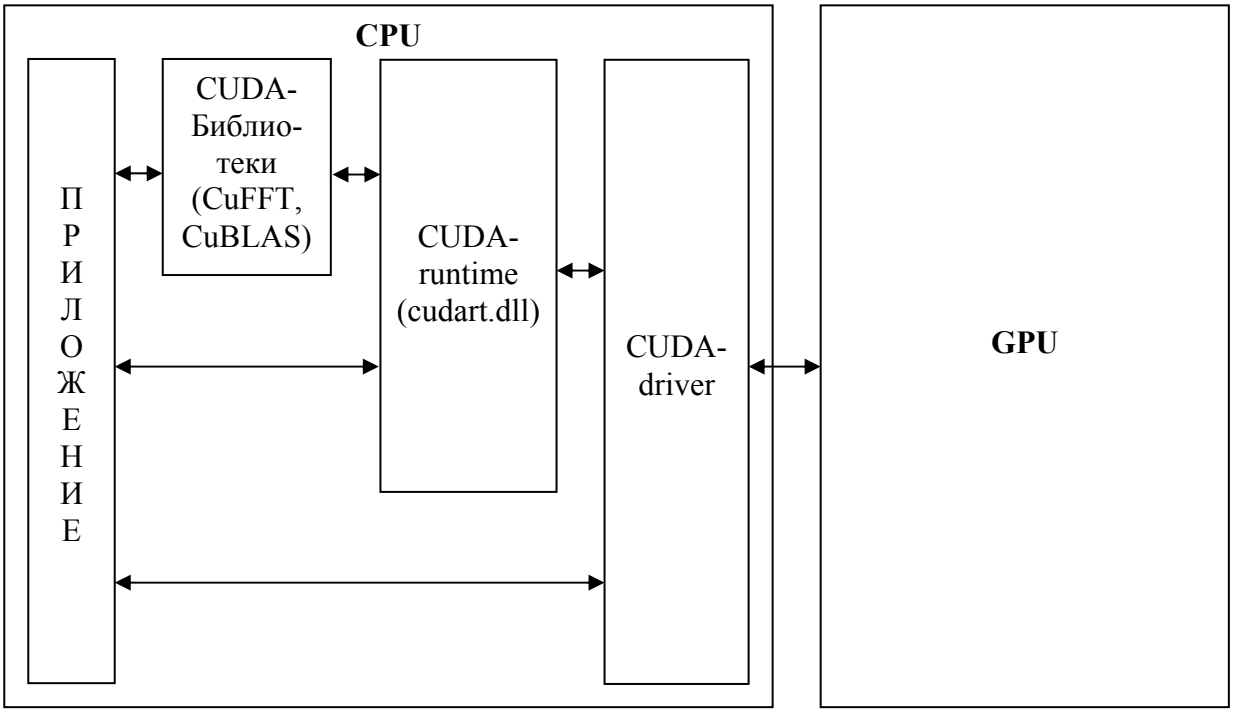

Рис. 4. Состав программного обеспечения CUDA

Программы для CUDA пишутся на "расширенном" языке C, при этом их параллельная часть (так называемые «ядра») выполняется на GPU, а обычная последовательная часть – на CPU. Компоненты архитектуры CUDA, принимающие участие в подготовке и исполнении приложения, автоматически осуществляют разделение частей и управление их запуском.

С точки зрения разработчика программы для CUDA графический процессор представляет собой одно устройство управления и огромное количество арифметико-логических устройств (АЛУ), каждое из которых обладает собственной регистровой памятью (распределением регистров между АЛУ управляет программное обеспечение CUDA) и имеет доступ к нескольким уровням памяти различного объема и быстродействия. Все арифметико-логические

устройства в любой данный момент времени исполняют одну команду, выбранную и декодированную устройством управления. Реально устройств управления несколько (их количество зависит от модели графического процессора). Совокупность из одного устройства управления, связанных с ним арифметико-логических устройств, регистров, быстрой разделяемой памяти, доступной из каждого АЛУ, и более медленной локальной памяти каждого АЛУ образуют так называемый мультипроцессор. Все мультипроцессоры имеют доступ к еще более медленным кэшу текстур, кэшу констант и глобальной памяти устройства, через которую осуществляется передача данных в/из основную память компьютера. Внутренние элементы памяти мультипроцессора недоступны из CPU. Структура памяти графического процессора и возможность доступа к памяти разного уровня из программы центрального процессора показаны на рис. 5.

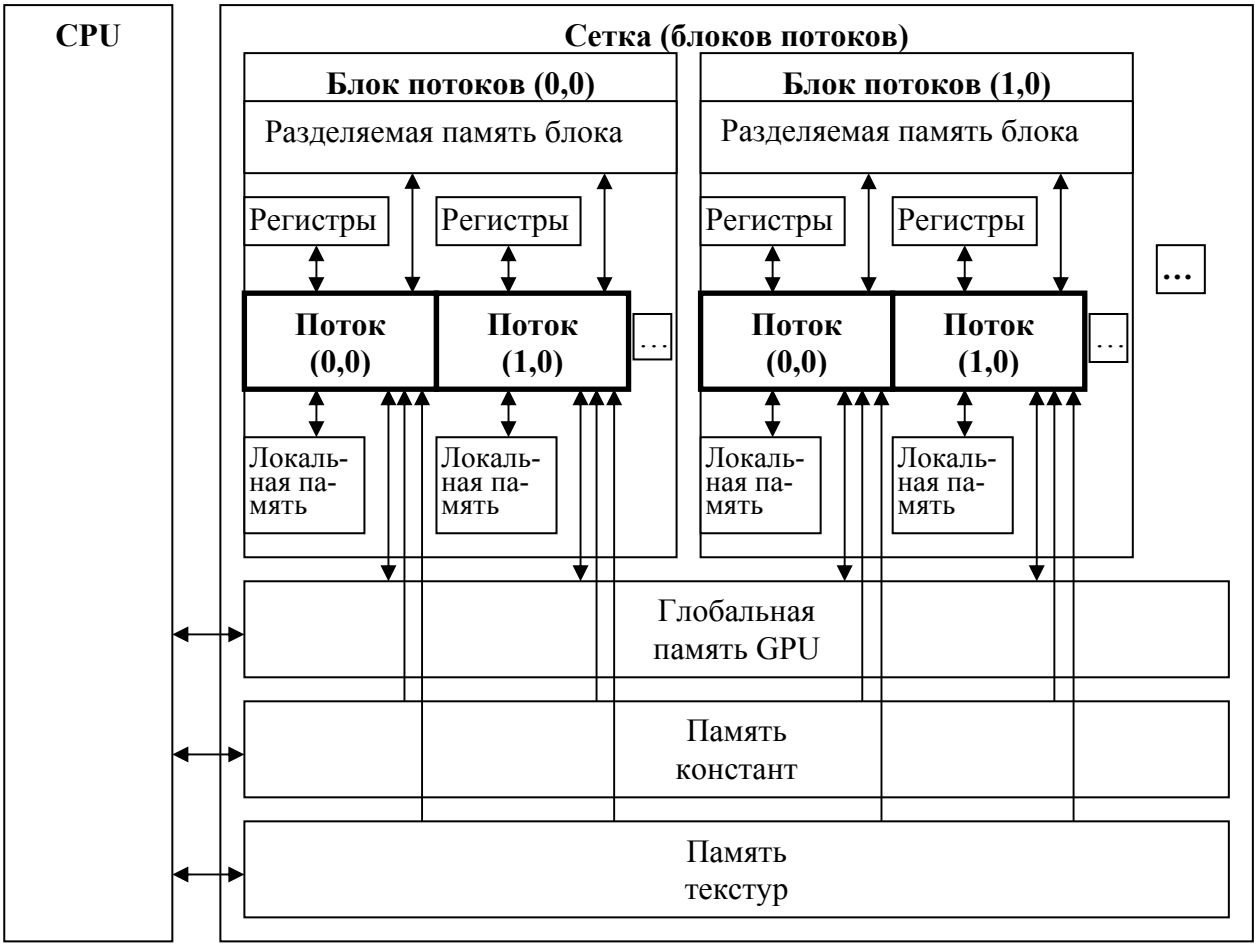

Рис.5. Структура памяти графического процессора

Отображение одной или нескольких параллельных областей программы на совокупность мультипроцессоров осуществляет программное обеспечение CUDA и является «прозрачным» для программиста. Ему нужно знать только о том, что с логической точки зрения параллельная область программы («ядро» в терминологии CUDA) представляет собой сетку (grid) блоков (block) потоков (thread). Блок потоков – это набор потоков, выполняемых одним мультипроцессором. Потоки одного блока могут взаимодействовать через разделяемую па-

мять и синхронизироваться между собой. Сетка и (независимо от нее) каждый из блоков представляют собой одно-, дву- или трехмерную структуру блоков и потоков соответственно. Количество размерностей и размеры каждой из них задает программист. Каждый поток после запуска имеет доступ к переменным (структурам), хранящим его собственные координаты внутри блока и координаты охватывающего блока внутри сетки.

Архитектура CUDA предусматривает широкий набор возможностей и поддерживается в видеокартах нескольких линеек (GEforce, Quadro, NVS, Tesla, ION). Разные модели графических процессоров, функционирующих в таких видеокартах, имеют каждая собственный набор параметров (например – количество регистров, объем разделяемой памяти мультипроцессора) и поддерживают разные наборы возможностей, сгруппированные в версии архитектуры от 1.0 до 2.1 (по состоянию на конец 2011 года). Для каждого устройства производитель указывает поддерживаемую им версию CUDA, по номеру версии можно определить значения параметров и доступные возможности. Соответствие поддерживаемых возможностей версии CUDA показано в таблице 1.

Таблица 1.

<table>
<tr><th rowspan="2">Поддерживаемые возможности (не указанные здесь возможности поддерживаются во всех версиях)</th><th colspan="5">Версия CUDA</th></tr>
<tr><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.x</th></tr>
<tr><td>Целочисленные atomic-функции, работающие над 32-разрядными словами в глобальной памяти</td><td>Нет</td><td colspan="4">Да</td></tr>
<tr><td>Целочисленные atomic-функции, работающие над 64-разрядными словами в глобальной памяти</td><td colspan="2" rowspan="3">Нет</td><td colspan="3" rowspan="3">Да</td></tr>
<tr><td>Целочисленные atomic-функции, работающие над 32-разрядными словами в разделяемой памяти</td></tr>
<tr><td>Warp-функции голосования</td></tr>
<tr><td>Операции с плавающей точкой двойной точности</td><td colspan="3">Нет</td><td colspan="2">Да</td></tr>
<tr><td>Дополнительные atomic-функции, работающие над 32-разрядными числами с плавающей точкой в глобальной и разделяемой памяти</td><td colspan="4" rowspan="2">Нет</td><td rowspan="2">Да</td></tr>
<tr><td>Дополнительные функции синхронизации</td></tr>
</table>

В таблице 2 приведено соответствие технических характеристик, на которые нужно ориентироваться при разработке параллельных программ, версии CUDA, реализованной в конкретной видеокарте. Для того, чтобы узнать, какую версию поддерживает подключенная к компьютеру видеокарта, нужно воспользоваться функцией

cudaGetDeviceProperties(...);

Формат ее аргументов и возвращаемых результатов можно найти в руководстве по программированию для CUDA [15].

Таблица 2

Технические характеристики	**Версия (compute capability)**				
	1.0	**1.1**	**1.2**	**1.3**	**2.x**
Максимальные x- или y- размерности сетки блоков	65535				
Максимальное количество нитей в блоке	512				1024
Максимальные x- или y- размерности блока	512				1024
Максимальная z- размерность блока	64				
Размер Warp	32				
Максимальное количество резидентных блоков в мультипроцессоре	8				
Максимальное количество резидентных warp-ов в мультипроцессоре	24		32		48
Максимальное количество резидентных нитей в мультипроцессоре	768		1024		1536
Количество 32-битных регистров в мультипроцессоре	8 K		16 K		32 K
Максимальное объем разделяемой памяти в мультипроцессоре	16 KB				48 KB
Количество банков разделяемой памяти	16				32
Объем локальной памяти одной нити	16 KB				512 KB
Размер памяти констант	64 KB				
Размер кэша памяти констант одного мультипроцессора	8 KB				
Размер кэша памяти текстур одного мультипроцессора	От 6 KB до 8 KB				
Максимальная ширина ссылки на 1D текстуру, связанную с CUDA массивом	8192				32768
Максимальная ширина ссылки на 1D текстуру, связанную с линейной памятью	2^{27}				
Максимальная ширина и высота ссылки на 2D текстуру, связанную с линейной памятью или CUDA массивом	65536 x 32768				65536 x 65535
Максимальная ширина, высота и глубина ссылки на 3D текстуру, связанную с линейной памятью или CUDA массивом	2048 x 2048 x 2048				
Максимальное количество текстур, которые могут быть связаны с ядром	128				

Максимальная ширина ссылки на 1D поверхность, связанную с CUDA массивом	Not supported	8192
Максимальная ширина и высота ссылки на 1D поверхность, связанную с CUDA массивом	Not supported	8192 x 8192
Максимальное количество поверхностей, которые могут быть связаны с ядром		8
Максимальное количество инструкций на ядро	2×10^6	

Рекомендуемая разработчиками архитектуры технология параллельного программирования состоит в следующем:

1 Спланировать разбиение обрабатываемых данных на фрагменты, целиком помещающиеся в разделяемую память графического процессора.

2 Обеспечить загрузку данных в глобальную память GPU.

3 Запустить ядро, в программе каждого блока:

3.1. Выполнить перемещение нужных фрагментов данных из глобальной памяти в локальную память блока.

3.2. Выполнить требуемые вычисления.

3.3. Скопировать сформированные результаты обработки из разделяемой памяти в глобальную память устройства.

4 Перенести полученные результаты из памяти графического процессора в основную память компьютера, выполнить их окончательную обработку или вывести на внешний носитель.

Расширенная среда программирования для языка C включает:

– Расширения синтаксиса для написания кода для GPU, реализуемые дополнительным препроцессором nvcc.

– Run-time библиотеки:

- Общая часть – встроенные векторные типы данных и набор функций, исполняемых и на CPU, и на GPU.
- CPU-компонента, для доступа и управления одним или несколькими GPU (драйвер).
- GPU-компонента, предоставляющая функции, специфические только для GPU.

Расширения синтаксиса языка включают в себя:

1. Дополнительные виды функций:

	Исполняется на	Вызывается из
__host__ float HostFunc()	CPU	CPU
__global__ void KernelFunc()	GPU	CPU
__device__ float DeviceFunc()	GPU	GPU

Особенности дополнительных видов функций:

Функция, объявленная как __global__, является функцией-ядром, она не может возвращать никакого значения (всегда void).

Функция вида __device__ и выполняется и запускается из функций, исполняемых на GPU, поэтому нельзя получить указатель на такую функцию.

В функциях, исполняемых на GPU:

- Недопустима рекурсия.
- Не может быть статических переменных внутри функций.
- Количество аргументов не может быть переменным.

2. Дополнительные виды переменных:

	Тип памяти	Область видимости	Время жизни
__device__ __shared__ int	разделяемая	блок	блок
__device__ int	глобальная	сетка	ядро
__device__ __constant__ int	кэш констант	сетка	ядро

Ключевое слово *__device__* необязательно, если используется *__shared__* или *__constant__* . Локальные переменные без идентификатора вида хранятся в регистрах, за исключением больших структур или массивов, действительно располагающихся в локальной памяти. Тип памяти указателя адаптируется к типу присвоенного ему выражения

3. Встроенные векторные типы объявляются так:

[u]char[1..4], [u]short[1..4], [u]int[1..4], [u]long[1..4], float[1..4]

Буква *u* в этих объявлениях необязательна, является сокращением от слова *unsigned* и в случае ее указания означает, что все поля данного типа являются беззнаковыми целыми.

Доступ к полям этих типов осуществляется по именам: *x, y, z, w,* например:

```
uint4 param;
int y = param.y;
int abc = param.w;
```

Тип *dim3* является синонимом *uint3*. Обычно тип *dim3* используется для задания параметров сетки.

4. Ядро (собственно программа для GPU) и его запуск:

При запуске ядру должны быть переданы обязательные параметры конфигурации сетки. Для этого в программе для CPU описываются и создаются следующие функции и переменные:

__global__ void KernelFunc(...); – прототип функции, являющейся ядром.

dim3 DimGrid(100, 50); – переменная CPU, определяющая двумерность сетки блоков потоков и ее размеры 100 x 50, т.е. 5000 блоков.

dim3 DimBlock(4, 8, 8); – переменная CPU, определяющая трехмерность совокупности потоков в каждом блок и размеры этой совокупности 4 x 8 x 8, т.е. 256 потоков на блок.

size_t SharedMemBytes = 64; – переменная CPU, определяющая размер разделяемой памяти в 64 байта, выделяемой каждому потоку.

После этих объявлений можно запустить ядро, для этого тоже реализован специальный синтаксис:

KernelFunc<<<DimGrid, DimBlock, SharedMemBytes>>>(...);

Опциональные *SharedMemBytes* байт:

– выделяются в дополнение к статически объявленным разделяемым переменным в ядре;

– отображаются на любую переменную вида: *extern __shared__ float DynamicSharedMem[]*.

Запуск ядра асинхронен, сразу после вызова функции управление немедленно возвращается к следующему оператору программы для CPU.

5. Встроенные переменные, создаваемые и инициализируемые автоматически при запуске ядра:

dim3 gridDim; – размеры сетки в блоках (*gridDim.w* не используется).

dim3 blockDim; – размеры одного блока в потоках.

dim3 blockIdx; – индекс блока внутри сетки.

dim3 threadIdx; – индекс потока внутри блока.

Эти переменные доступны только для кода параллельных ветвей ядра, исполняемого на GPU.

Общая часть Run-time библиотек содержит большое количество функций, необходимых (или опциональных) при написании параллельной программы. Таких функций слишком много для того, чтобы приводить их описание в данном методическом пособии, поэтому здесь приводится описание только тех функций, которые действительно необходимы. Все библиотечные функции можно разделить на несколько групп:

1. Управление потоками CPU и GPU, синхронизация.
2. Обработка ошибок.
3. Управление устройством.
4. Управление событиями.
5. Управление глобальной памятью.
6. Интерфейс с OpenGL и Direct3D версий 9, 10 и 11.
7. Интерфейс с видеоадаптером.
8. Управление текстурами и поверхностями.
9. Интерфейс с драйвером CUDA.

При разработке любой параллельной программы невозможно обойтись без использования функций управления глобальной памятью GPU. Таких функций более 40, из них наиболее употребительны следующие:

cudaError_t cudaMalloc (void ∗∗devPtr, size_t size);

Выделяет блок глобальной памяти устройства размером *size* и заносит адрес этого блока в указатель *devPtr*. Возвращает либо код успешного завершения *cudaSuccess*, либо код ошибки, если блок памяти распределить не удалось.

cudaError_t cudaFree (void ∗devPtr);

Возвращает ранее выделенный блок глобальной памяти в пул свободной памяти. Возвращает либо код успешного завершения, либо код ошибки (например, если этот блок ранее не выделялся).

*cudaError_t cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind);*

Копирует блок памяти размером *count* байт, расположенный по адресу *src*, в память по адресу *dst*. Один из этих блоков размещается в основной памяти компьютера, второй – в глобальной памяти графического процессора, какой из них где – определяется значением аргумента *kind* (допустимы *cudaMemcpyHostToDevice* и *cudaMemcpyDeviceToHost*). Возвращает либо код успешного завершения, либо код ошибки.

При необходимости использования других функций можно использовать руководства по программированию для CUDA и справочные материалы.

Этапы выполнения работы

1. Изучить основы архитектуры CUDA и настройку среды разработки Microsoft Visual Studio для разработки и отладки параллельных программ графического процессора.
2. Разработать план размещения данных решаемой задачи в разделяемой памяти GPU.
3. Разработать ядро (функцию для GPU, выполняющую массовую параллельную обработку матрицы) в соответствии с этим планом.
4. Разработать последовательную программу для CPU, обеспечивающую считывание исходных данных, перенос их в глобальную память GPU, запуск ядра, ожидание его завершения, считывание и окончательную обработку результатов. Отладить совокупность программ для CPU и GPU.
5. Проанализировать результаты решения задачи, оценить отклонение полученных значений от ожидаемых, объяснить причины отклонений.
6. Подготовить в электронном виде, сдать преподавателю и защитить отчет по работе.

Требования к содержанию отчета

Отчет по данной лабораторной работе включается в общий отчет по лабораторному практикуму (см. лабораторную работу 4, п. 4.3.). Отчет должен содержать:

- цель работы;
- краткое описание архитектуры CUDA и графического процессора, с использованием которого решается поставленная задача;
- описание используемого метода распараллеливания;
- описание параллельного алгоритма решения задачи;
- листинг программы;
- результаты измерений временных характеристик для разных размеров сетки и блока потоков, анализ эффективности параллельной программы.

Контрольные вопросы

1. Что такое ядро в терминологии CUDA?
2. Как задается размерность и количество потоков ядра, выполняемого графическим процессором?
3. Перечислите виды и характеристики памяти, доступной из программы GPU.
4. Как указать требуемое размещение переменной в памяти GPU (регистровой, разделяемой, глобальной, памяти текстур, …)?
5. Перечислите виды и характеристики памяти, доступной из программы CPU при использовании архитектуры CUDA.
6. С помощью функций какой группы осуществляется копирование блоков данных из основной памяти CPU в глобальную память GPU?
7. Что такое тип данных ulong2? Какие поля содержит этот тип данных?
8. Какова максимальная размерность блока в потоках?
9. Перечислите встроенные векторные типы данных расширения языка C и объясните смысл их наименований.
10. Как в программе для CPU обеспечить синхронизацию с программой для GPU?
11. Если к одному CPU подключено несколько видеокарт NVidia архитектуры CUDA, то каким образом можно обеспечить их одновременную загрузку?
12. Перечислите виды и характеристики памяти GPU, доступной из программы основного процессора.
13. Какие библиотеки функций можно использовать при параллельном программировании для CUDA?
14. Какие ограничения наложены на функции, которые должны выполняться на GPU?
15. Что такое мультипроцессор в терминологии CUDA?
16. Может ли поток GPU прямо записать результаты своей работы в основную память CPU?
17. Как в программе для GPU определить переменную, которая должна размещаться в регистровой памяти?
18. Какие ограничения наложены на функции, выполняемые в графическом процессоре?
19. Каким образом программа для GPU (ядро) может получить свои координаты в блоке и в сетке блоков?
20. Как определить версию и технические характеристики графического процессора NVIDIA?
21. Перечислите и охарактеризуйте группы библиотечных функций Run-time библиотеки CUDA.
22. Что такое сетка, блок, поток в терминологии CUDA?
23. Что такое тип данных dim3? Синонимом названия какого типа данных он является?
24. Какова максимальная размерность сетки в блоках?

25. Перечислите и охарактеризуйте функции компонент программного обеспечения CUDA.
26. Можно ли (и если да – то как) в программе для CPU выяснить, какую версию спецификации CUDA реализует видеокарта?
27. Какую модель параллелизма реализует архитектура CUDA?
28. Каковы дополнительные виды функций в расширении языка C для CUDA?
29. Что делает препроцессор nvcc?
30. Можно ли получить указатель на функцию, выполняемую графическим процессором:
31. С помощью какого расширения синтаксиса из программы для CPU запускается ядро CUDA?
32. Перечислите основные группы функций CUDA и охарактеризуйте их назначение.

Список литературы

1. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. – СПб.: БХВ-Петербург, 2004.
2. Богачев К. Ю. Основы параллельных вычислений. – М., Бином. Лаборатория знаний, 2010.
3. Антонов А.С. Параллельное программирование с использованием технологии OpenMP. – М.: Изд-во МГУ, 2009.
4. Левин М.П. Параллельное программирование с использованием OpenMP – М., Бином. Лаборатория знаний, 2008
5. http://openmp.org/
6. http://cgm.computergraphics.ru