МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ «МЭИ»

Кафедра «Вычислительные машины, системы и сети»

А.В. Филатов
Д.А. Орлов

РАЗРАБОТКА ПРИЛОЖЕНИЙ ДЛЯ ВЫСОКОПРОИЗВОДИТЕЛЬНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Практическое пособие
по курсу «Программное обеспечение
высокопроизводительных вычислительных систем»
для студентов, обучающихся по направлению
«Информатика и вычислительная техника»

ISBN 978-5-7046-3461-4

УДК 621.398
ББК 32.965
Ф 517

Утверждено учебным управлением НИУ «МЭИ»
в качестве производственно-практического издания

Подготовлено на кафедре вычислительных машин, систем и сетей

Рецензент: докт. техн. наук, проф. Ш.А. Оцоков

Филатов, А.В.

Ф 517 Разработка приложений для высокопроизводительных вычислительных систем [Электронный ресурс]: практическое пособие / А.В. Филатов, Д.А. Орлов; под ред. А.В. Филатова. – Электрон. дан. – М.: Издательство МЭИ, 2026. – 1 электрон. опт. диск DVD-R.

Пособие подготовлено применительно к программе дисциплины «Программное обеспечение высокопроизводительных вычислительных систем» направления 09.04.01 «Информатика и вычислительная техника». Пособие содержит описание и варианты задания лабораторных работ по данному курсу.

Предназначено для студентов, обучающихся по направлению «Информатика и вычислительная техника».

Минимальные системные требования:

Тип ЭВМ: настольный PC (домашний, офисный), ноутбук, планшет, смартфон.
ОС: MS Windows, Linux, Android, MacOS.
Дополнительное оборудование: дисковод для компакт-дисков, компьютерная мышь.
Дополнительные программные средства: программа демонстрации файлов PDF.

ISBN 978-5-7046-3461-4

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ…… 4

Лабораторная работа №1

СОЗДАНИЕ ПРОГРАММНОГО КОМПЛЕКСА
И ИСПОЛЬЗОВАНИЕ МЕХАНИЗМА СОКЕТОВ
ДЛЯ ВЗАИМОДЕЙСТВИЯ ПРОЦЕССОВ…….. 5

Лабораторная работа №2

ПРОГРАММИРОВАНИЕ ГРАФИЧЕСКИХ ПРОЦЕССОРОВ
СРЕДСТВАМИ *NVIDIA CUDA*. ВВЕДЕНИЕ В ИСПОЛЬЗОВАНИЕ
GPU В КАЧЕСТВЕ СОПРОЦЕССОРА ДЛЯ УСКОРЕНИЯ
ВЫЧИСЛЕНИЙ…….. 12

Лабораторная работа №3

ПРОГРАММИРОВАНИЕ ГРАФИЧЕСКИХ ПРОЦЕССОРОВ
СРЕДСТВАМИ *NVIDIA CUDA*. РАЗМЕЩЕНИЕ ДАННЫХ
В ГЛОБАЛЬНОЙ, РАЗДЕЛЯЕМОЙ И КОНСТАНТНОЙ ПАМЯТИ…. 24

Лабораторная работа № 4

ПРИМЕНЕНИЕ ГРАФИЧЕСКИХ ПРОЦЕССОРОВ ДЛЯ РЕШЕНИЯ
ЗАДАЧ ОБРАБОТКИ ИЗОБРАЖЕНИЙ…….. 31

Лабораторная работа №5

ИЗУЧЕНИЕ ЯЗЫКА ПРОГРАММИРОВАНИЯ *OpenCL*
НА ПРИМЕРЕ РЕШЕНИЯ ЗАДАЧ ОБРАБОТКИ ИЗОБРАЖЕНИЙ..... 41

Лабораторная работа №6

ИЗУЧЕНИЕ МНОГОПОТОЧНОГО ПРОГРАММИРОВАНИЯ
НА ПРИМЕРЕ *POSIX THREADS* И СТАНДАРТНОЙ
БИБЛИОТЕКИ ЯЗЫКА *C++*……..... 49

СПИСОК ЛИТЕРАТУРЫ…….. 54

Приложение А…… 55

Приложение Б…… 57

Приложение В…… 61

Приложение Г…… 63

Приложение Д…… 65

ПРЕДИСЛОВИЕ

Пособие предназначено для студентов НИУ «МЭИ», обучающихся по направлению «Информатика и вычислительная техника».

Пособие содержит описание лабораторных работ по курсу «Программное обеспечение высокопроизводительных вычислительных систем».

Предполагается наличие начальных знаний и опыта в написании многопроцессных приложений, параллельных приложений и приложений с сетевым взаимодействием процессов посредством сокетов.

Лабораторные работы 1–3 разработаны и апробированы старшим преподавателем кафедры ВМСС А.В. Филатовым. Лабораторные работы 4–6 разработаны и апробированы к.т.н, доцентом кафедры ВМСС Д.А. Орловым.

Лабораторная работа №1

СОЗДАНИЕ ПРОГРАММНОГО КОМПЛЕКСА И ИСПОЛЬЗОВАНИЕ МЕХАНИЗМА СОКЕТОВ ДЛЯ ВЗАИМОДЕЙСТВИЯ ПРОЦЕССОВ

Лабораторная работа выполняется на двух компьютерах (назовём их *Host1* и *Host2*) с установленной *UNIX*-подобной операционной системой и соединёнными локальной сетью, поддерживающей протокол *TCP/IP*. В ходе выполнения лабораторной работы студенты пишут программы для трёх типов процессов *Client*, *Server* и *Workers*. Процесс *Client* запускается на компьютере *Host1*, а процессы *Server* и *Workers* на компьютере *Host2*. Процессы типа *Client* и *Server* запускаются в единственном экземпляре, а процессы типа *Workers* в нескольких (рис.1.).

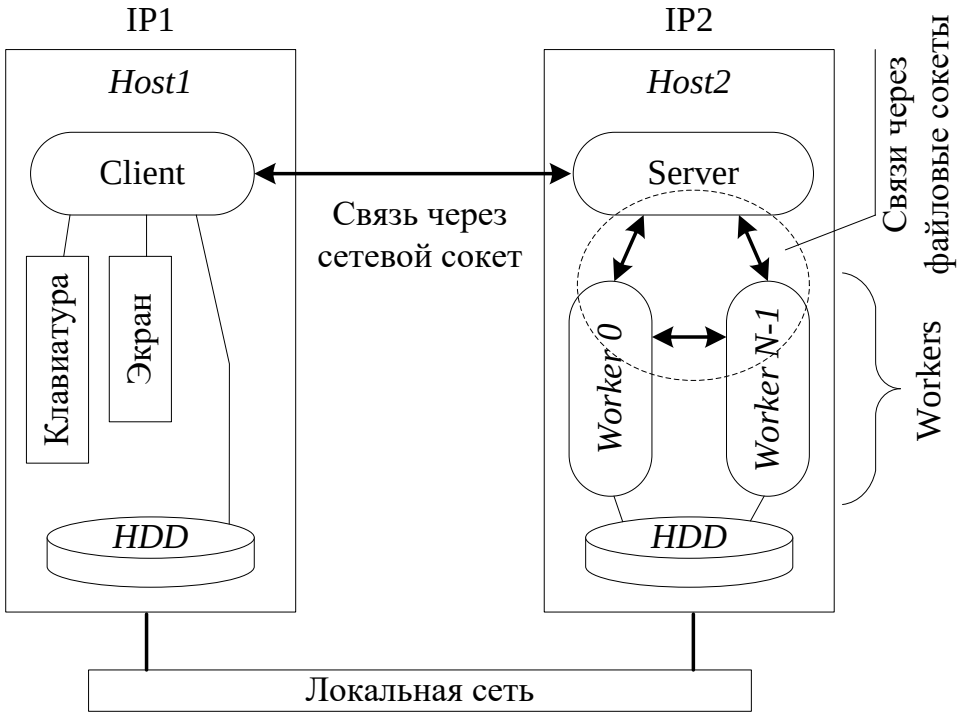


Рис. 1. Схема распределения процессов *Client, Server* и *Workers* по компьютерам *Host1* и *Host2*, а также каналов передачи данных

После запуска, процесс *Server* создаёт на своём компьютере *Host2* сетевой сокет, по которому к нему с компьютера *Host1* подключается процесс *Client*. Процессы *Workers* порождаются процессом *Server* (или, если указанно в задании, они порождаются от одного из уже запущенных процессов типа *Workers*) на компьютере *Host2* и между ними устанавливается связь с помощью файловых сокетов. Весь этот комплекс процессов выполняет функции согласно варианту задания из табл. 1.

Лабораторная работа рассчитана на студентов знакомых с программированием процессов, взаимодействующих друг с другом посредством сокетов. Однако дадим некоторые пояснения.

Для взаимодействия процессов между собой, выполняющихся как на разных сетевых хостах, так и на одном хосте, в данной лабораторной работе используется механизм сокетов. Чтобы организовать это взаимодействие каждый из процессов должен предпринять ряд действий. Особенностью взаимодействия любых двух процессов с использованием сокетов является то, что процессы не равнозначны. Один из процессов выделяется как сервер, а другой как клиент. Поэтому и последовательность действий у них разная.

Чтобы процесс-сервер выполнил свою последовательность действий, в программу надо вставить следующие строчки:

– ***int sockid, chan;*** – объявление целочисленой переменной *sockid*, которая будет использоваться для идентификации сокета, и переменной *chan*, которая будет использоваться для идентификации соединения;

– ***struct sockaddr_in addr;*** – объявление переменной *addr*, которая имеет тип *struct sockaddr_in*. Этот тип описывает структуру сетевого адреса сокета;

– ***sockid=socket(AF_INET,SOCK_STREAM,0);*** – вызов функции обращения к механизму сокетов с указанием семейства сетевых сокетов (*AF_INET*), указать тип соединения как потоковое с программным каналом (*SOCK_STREAM*) и протокол по умолчанию (ноль в третьем параметре). В качестве результата функция вернёт целочисленный идентификатор сокета, который будет присвоен переменной *sockid*;

– ***addr.sin_family = AF_INET;*** – присвоение полю *sin_family* переменной *addr* значения используемого семейства сокетов. Это нужно для правильной итерпретации адреса;

– ***addr.sin_port = htons(3425);*** – присвоение полю *sin_port* переменной *addr* номера сетевого порта (например ***3425***) сокета;

– ***inet_aton("10.4.129.38", &addr.sin_addr);*** – вызов функции присвоения полю *sin_addr* переменной *addr* значения *IP*-адреса хоста (например: "10.4.129.38"), на котором будет выполняться процесс-сервер. Конкретное значение *IP*-адреса хоста, на котором Вы будете запускать на выполнение процесс-сервер, необходимо получить у администратора лаборатории или преподавателя во время занятий;

– ***bind(sockid, &addr, sizeof(addr));*** – вызов функции связывания сокета с адресом (в нашем случае: сетевым адресом). Параметрами функции являются: идентификатор сокета *sockid*, указатель на переменную *addr* и функция *sizeof***,** которая позволяет получить реальный размер переменной *addr*;

– ***listen(sockid, 1);*** – вызов функции установления очереди соединений и активации прослушивания сокета;

– ***chan = accept(sockid, NULL, NULL);*** – вызов функции ожидания соединения (подключения процесса-клиента). На время ожидания процесс-сервер приостановится. Когда процесс-клиент подключится, процесс-сервер возобновит свою работу, а функция *accept* присвоит переменной *chan* идентификатор соединения. Второй и третий параметры функции можно занулить константой *NULL*;

– после установления соединения идут строчки с операторами работы процесса, и среди них вставляются вызовы функций работы с установленным соединением. Среди них могут быть функции ***recv*** – приёма данных от процесса-клиента из соединения (например*: recv(chan, (char*) buf, 1024, 0); –* приём из соединения *chan* 1024 байта данных, с их сохранением в массиве *buf*), и функций ***send*** – посылки данных процессу-клиенту в соединение (например*: send(chan, (char*) buf, 1024, 0); –* посылка в соединение *chan* 1024-х байтов данных из массива *buf*);

– ***close(chan);*** – вызов функции закрытия соединения *chan*.

Чтобы процесс-клиент выполнил свою последовательность действий, в его программу надо вставить другую последовательность строчек:

– ***int sockid;*** – объявление целочисленой переменной *sockid*, которая будет использоваться для идентификации сокета;

– ***struct sockaddr_in addr;*** – аналогично процессу-клиенту;

– ***sockid=socket(AF_INET,SOCK_STREAM,0);*** – также аналогично процессу-клиенту;

– ***addr.sin_family = AF_INET;*** – присвоение полю *sin_family* переменной *addr* значения используемого семейства сокетов;

– ***addr.sin_port = htons(3425); –*** присвоение полю *sin_port* переменной *addr* номера сетевого порта (например ***3425***) серверного сокета;

– ***inet_aton("10.4.129.38", &addr.sin_addr);*** – вызов функции присвоения полю *sin_addr* переменной *addr* значения *IP*-адреса серверного сокета. Необходимо помнить, что задаётся *IP*-адрес хоста, на котором будет выполняться процесс-сервер, а также номер сетевого порта серверного сокета, поскольку именно к нему будет подключаться процесс-клиент;

– ***connect(sockid, &addr, sizeof(addr));*** – вызов функции подключения процесса-клиента к процессу-серверу. На время ожидания процесс-клиент приостановится. Когда процесс-клиент подключится, он продолжит свою работу. Второй и третий параметры функции - это указатель на переменную адреса серверного сокета и вызов функции определения её фактического размера;

– после установления соединения идут строчки с операторами работы процесса, и среди них вставляются вызовы функций работы с установленным соединением, такие как ***recv*** и ***send*** (примеры использования смотрите у процесса-сервера). Единственным отличием является то, что первым параметром функций *recv* и *send* ставится идентификатор сокета *sockid*, а не идентификатор соединения *chan* как у процесса-сервера;

– ***close(sockid);*** – вызов функции закрытия соединения с серверным сокетом.

Если выполнение лабораторного задания подразумевает порождение нового процесса от уже существующего на одном хосте, то последовательность действий в этом случаем простая. Сначала выполняется только один процесс, который называют процесс-предок. Когда появляется необходимость породить новый процесс, процесс-предок вызывает функцию порождения нового процесса – процесса потомка.

Например, так: ***(int) pid=fork();***

Особенностью функции *fork* является то, что вызывает её один процесс – процесс-предок, а завершают уже оба процесса. Поскольку процесс-потомок выполняет ту же самую программу, что и процесс-предок, он является как бы копией процесса-предка. Отличием является то, что функция *fork* по своему окончанию в процессе-предке присвоит переменной *pid* числовой идентификатор процесса-потомка, а в процессе-потомке функция *fork* присвоит переменной *pid* значение ноль. Проверив значение переменной *pid*, процессы «узнают» кто из них предок, а кто потомок и таким образом могут спланировать свою работу.

Домашняя подготовка

1. Разработать программы для комплекса процессов согласно варианту задания из табл. 1.

2. Проверить и отладить программы, разработанные в п. 1.

Задание на лабораторную работу

1. Испытать и, если необходимо, отладить разработанные программы на одном компьютере.

2. Распределить исполняемые файлы программ и исходные данные по двум компьютерам (*Host1* и *Host2*) и выполнить задание в полном объёме.

3. Составить отчёт, содержащий исходные тексты программ с комментариями, результаты выполнения работы и комментарии.

Таблица 1

№ бриг.	Задание
1	Разработать отказоустойчивую программу *Program*, которая выполняет некоторые продолжительные вычисления и периодически (в соответствии с контрольными точками программы) сохраняет в файл состояние вычислений и промежуточные данные. В случае возникновения сбоя и возобновления программы после аварийного завершения, она должна возобновить работу с последней контрольной точки. Исполняемый файл программы *Program* выполняется процессом типа *Workers,* порождаемым процессом *Server* на компьютере *Host2*. Процесс *Client* связывается с процессом *Server* (посредством сетевого сокета, создаваемого процессом *Server*), и должен через него иметь возможность удалённо запускать, прерывать и возобновлять выполнение программы *Program,* а также получать промежуточные или окончательные результаты
2	На диске компьютера *Host2* находится распределённая (в трёх файлах) база данных. Процесс *Server* создаёт сетевой сокет, посредством которого к нему с компьютера *Host1* подключается процесс *Client*. Процесс *Client* посылает *Server* запрос на поиск в базе данных строки содержащей заданное слово. Процесс *Server* порождает три процесса *Workers*, каждый из которых осуществляет поиск в своём файле. Результат поиска (строчки с искомым словом) передаются процессу *Server*, который передаёт их *Client*.
3	На диске компьютера *Host2* находятся исполняемые файлы заранее приготовленных программ из некоторого множества *Programs*. Каждая из программ принимает исходные данные для своей работы в определённом формате, и в другом определённом формате их возвращает. На компьютере *Host2* запускается процесс *Server*, который создаёт сетевой сокет. К нему через этот сокет с компьютера *Host1* подключается процесс *Client*. Процесс *Client* посылает *Server* задание на удалённое выполнение любого набора программ из множества *Programs* и исходные данные к ним. *Server* порождает процессы *Works*, каждый из которых выполняет свою программу. Результаты выполнения через *Server* отправляются процессу *Client*

Продолжение табл. 1

№ бриг.	Задание
4	На диске компьютера *Host2* находится распределённая (в трёх файлах) база данных. Процесс *Server* создаёт сетевой сокет, посредством которого к нему с компьютера *Host1* подключается процесс *Client*. Процесс *Client* посылает *Server* запрос на поиск в базе данных строки содержащей заданное слово. Процесс *Server* порождает три процесса *Workers*, каждый из которых осуществляет поиск в своём файле. Результат поиска (строчки с искомым словом), через файловые сокеты, передаются одному из процессов *Workers* (указанному *Client*). Этот процесс передаёт их *Server*-у, который потом передаёт их процессу *Client*
5	На компьютере *Host2* запускается процесс *Server*, который создаёт сетевой сокет. К нему через этот сокет с компьютера *Host1* подключается процесс *Client*. На диске компьютера *Host1* находятся исполняемые файлы некоторых программ. Процесс *Client* передаёт любую из этих программы *Server*, который сохраняет её на диске *Host2*, затем порождает процесс *Workers*, который эту программу выполняет. Результаты выполнения программы забираются *Server* и отправляются процессу *Client*
6	На диске компьютера *Host2* расположен файл *File.txt*. На компьютере *Host2* запускается один из процессов *Workers*, который порождает другие процессы *Workers* и процесс *Server*. Процесс *Server* создаёт на *Host2* сетевой и файловый сокеты, к которым подключаются процессы *Works* и процесс *Client* с компьютера *Host1*. Процессы *Workers* поочерёдно считывают содержимое файла *File.txt* и, через процесс *Server*, передают его *Client*, который выводит его на экран
7	На диске компьютера *Host2* находится (в семи файлах) база данных. На компьютере *Host2* запускается один из процессов *Works*, который порождает два других процесса *Workers* и процесс *Server*. Процесс *Server* создаёт на *Host2* сетевой и файловый сокеты, к которым подключаются процессы *Workers* и процесс *Client* с компьютера *Host1*. Процесс *Client* посылает *Server* запрос на поиск в базе данных строки содержащей заданное слово. Три процесса *Workers* осуществляют поиск в файлах. Результат поиска (строчки с искомым словом) передаются процессу *Server*, который передаёт их *Client*

Окончание табл. 1

№ бриг.	Задание
8	Разработать программу *Program*, которая сохраняет в файл данные, принятые из сокетного канала. На компьютере *Host2* запускается процесс *Server*, который создаёт сетевой сокет. К нему через этот сокет с компьютера *Host1* подключается процесс *Client*. На диске компьютера *Host1* располагается исполняемый файл программы *Program*. Процесс *Client* передаёт исполняемый файл программы *Program* процессу *Server*, который сохраняет её на диске *Host2*, затем порождает процесс *Workers*, который начинает выполнять программу *Program*. В ходе выполнения, процесс *Client* через процесс *Server* передаёт процессу *Workers* данные, которые тот сохраняет в файл
9	На компьютере *Host2* запускается процесс *Server*, который создаёт первый сетевой сокет (и является для него сервером). На компьютере *Host1* запускается процесс *Client*, который создаёт второй сетевой сокет (и также является для него сервером). Процессы *Client* и *Server* подключаются в качестве клиентов к сокетам друг друга. Процесс *Server* создаёт файловый сокет, запускает три процесса *Workers*, каждый из которых подключается к файловому сокету. Процессы *Workers* выполняют некоторые вычисления и могут по требованию передавать *Server* промежуточные результаты. Периодически процесс *Client* через первый сетевой сокет отправляет запрос *Server* на получение промежуточных данных, от какого либо процесса *Workers*, и получает их от него через второй сетевой сокет. Сам процесс *Server* получает промежуточные данные от *Workers* через файловый сокет

Контрольные вопросы

1. Как организуется взаимодействие процессов посредством сетевых сокетов?
2. Что такое процесс *Server* и процесс *Client* клиент?
3. Как строится программа для процесса *Server*?
4. Как строится программа для процесса *Client*?
5. Как происходит порождение нового процесса от уже имеющегося?

Лабораторная работа №2

ПРОГРАММИРОВАНИЕ ГРАФИЧЕСКИХ ПРОЦЕССОРОВ СРЕДСТВАМИ *NVIDIA CUDA*. ВВЕДЕНИЕ В ИСПОЛЬЗОВАНИЕ *GPU* В КАЧЕСТВЕ СОПРОЦЕССОРА ДЛЯ УСКОРЕНИЯ ВЫЧИСЛЕНИЙ

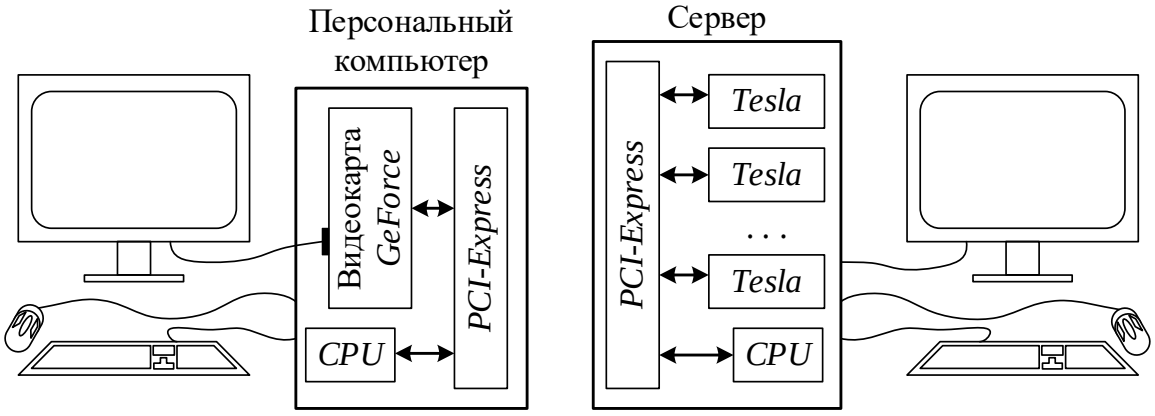
Графические процессоры используются в вычислительной технике уже давно. Основное их предназначение – обработка и вывод графической информации на экран монитора вычислительной системы. Графические процессоры (*GPU*) включались в состав специальных устройств – видеоадаптеров (или графических адаптеров), и, по мере своего развития, сначала помогали основному процессору (*CPU*), а позже взяли на себя основную заботу по подготовке и выводу графики на экран. Развитие функций *GPU* привело к такому их усовершенствованию, что они смогли выполнять многие универсальные вычислительные функции. Если сначала функции *GPU* задавались жёстко на уровне аппаратуры, то потом разработчики графических процессоров позволили эти функции (так называемые шейдеры) программировать. Некоторые продвинутые программисты воспользовались этим, чтобы использовать графические адаптеры с *GPU* в качестве вычислительных сопроцессоров к основному *CPU*. Это оказалось непростым делом и требовало от программиста хороших знаний по программированию и функционированию *GPU*, поэтому использовалось довольно узким кругом специалистов.

Ситуация изменилась, когда известный мировой разработчик графических процессоров – компания *NVIDIA* создала в 2006 г. свой аппаратно-программный комплекс *CUDA* (*Compute Unified Device Architecture*). Этот аппаратно-программный комплекс позволяет упростить использование графических процессоров, разработанных компанией *NVIDIA*, в качестве вычислительных сопроцессоров и сводит их программирование к относительно простому расширению «обычных» программ на языках *C* и *C++*, написанных для *CPU*.

Поскольку, *GPU* изначально создавались для видеоадаптеров, то и технология *CUDA* была сначала внедрена на видеоадаптерах *NVIDIA* серии *GeForce* (начиная с *GeForce 8800 GTX*). Позже, компания *NVIDIA* стала создавать уже специальные вычислительные ускорители на *GPU*, такие как *Tesla*. Как видеоадаптеры *GeForce*, так и ускорители *Tesla* могут быть установлены в вычислительную систему с интерфейсом *PCI-Express* (в перспективе рассматривается использование других, возможно даже

создание более быстрых специализированных) в количестве от одного и более, и, путем применения *CUDA*, использованы для вычислительных задач. На данный момент в мире получают широкое распространение кластерные вычислительные системы, узлы которых снабжаются ускорителями на базе *GPU*. К таким кластерным системам, в частности, относится суперкомпьютеры установленные в НИВЦ МГУ.

На рисунке 2 схематически показаны: настольный персональный компьютер с видеоадаптером *NVIDIA* серии *GeForce* на шине *PCI-Express* (на его месте может быть ноутбук со встроенным *GeForce*) и сервер, с несколькими ускорителями *NVIDIA* серии *Tesla*, также на шине *PCI-Express*.



а) *б*)

Рис. 2. Ускорители на *GPU* (*GeForce* и *Tesla*) в персональном компьютере (а) и сервере (б)

В данном описании будем рассматривать классическую структуру графического ускорителя. Она хоть и устаревшая, но наиболее простая, и позволяет быстрее понять основные принципы построения графических ускорителей *NVIDIA*.

На рисунке 3 схематически показано обобщенное внутреннее устройство одного ускорителя на *GPU*. Этим ускорителем может быть и видеоадаптер, и специализированный ускоритель. У каждого конкретного устройства будет свой набор и количество составных частей. Важным отличием специализированного ускорителя от просто видеоадаптера является наличие у него дополнительных средств контроля, а также устройств, выполняющих операции двойной точности. Ускоритель имеет в своём составе несколько потоковых мультипроцессоров *SM* (*Streaming Multiprocessor*). Каждый мультипроцессор состоит из элементарных ядер (*Core*) – потоковых процессоров *SP* (*Streaming Processor*) работающих под общим управлением по типу *SIMD*. Потоковые мультипроцессоры

группируются (в классической версии устройства их 2–3 шт.) в текстурные процессорные кластеры *TPC* (*Texture Processor Cluster*). В их рамках они имеют общую текстурную память (используется для хранения графических текстур). На рисунке 3 содержимое одного из *TPC* детализировано, а в нём детализированы два из трёх имеющихся в нем *SM*. Каждому из потоковых процессоров *SP* аппаратный планировщик выделяет (в момент вычислений) в его распоряжение часть регистрового массива *RM*. При решении задачи, в каждый момент времени, на каждом *SP* выполняется один поток вычислений – *thread*, однако в отличие от классических потоков, потоки всех *SP*, входящих в *SM*, выполняются под общим управлением, единого для всего *SM*, устройства управления (т.е. по принципу *SIMD*, как было отмечено ранее).

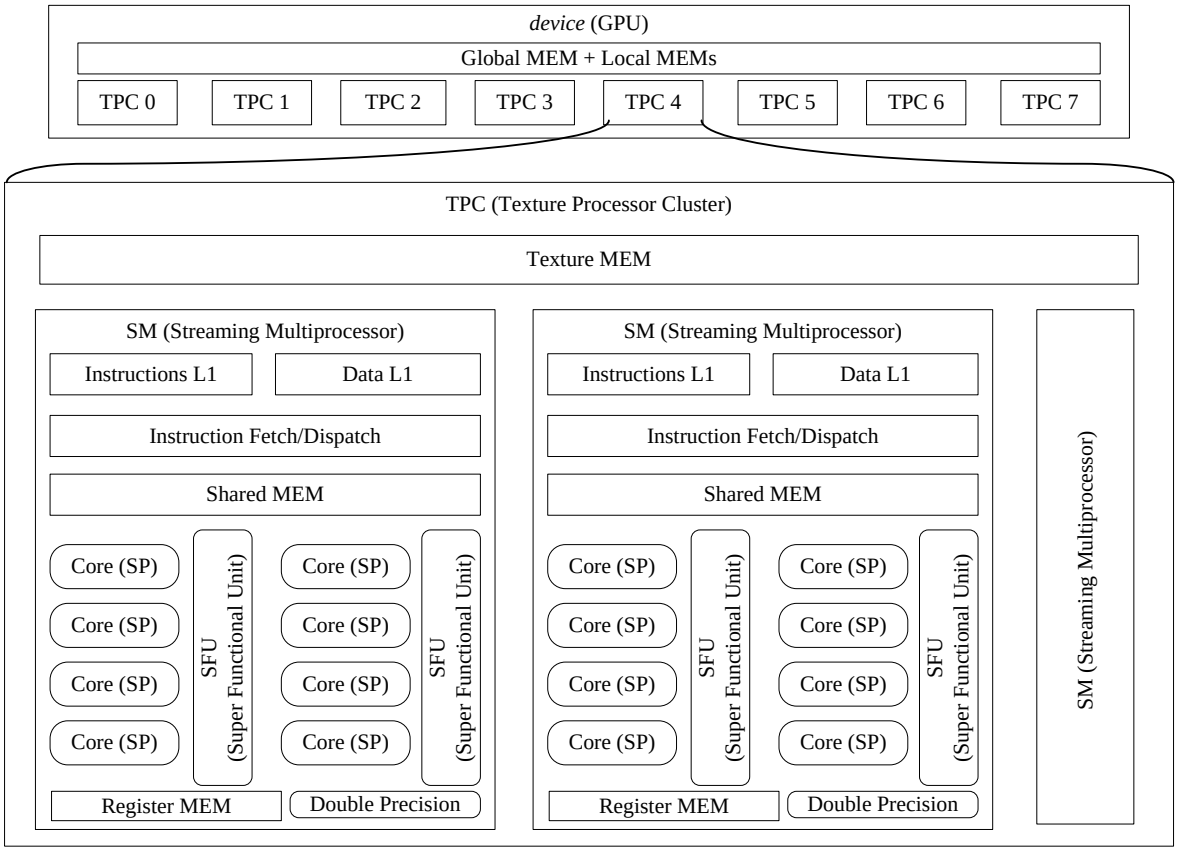


Рис. 3. Внутреннее устройство ускорителя на *GPU*

На каждый *SM* может быть назначено любое количество потоков. Потоки, назначенные для выполнения на один *SM* делятся на варпы. Варп (*warp*) – это совокупность потоков, единовременно выполняемых в рамках одного потокового мультипроцессора *SM*. На момент создания данного описания, типичный размер варпа равен 32. Кроме *SP*, в *SM* ещё имеются блоки специальных функций (*SFU*).

Выполняющиеся в рамках *SM* потоки, могут взаимодействовать друг с другом посредством небольшой (ёмкостью, измеряемой килобайтами, например в 48 Кб) быстродействующей (поскольку она интегрирована на кристалл вместе с *SM*) общей памяти *Shared MEM*. Также у *SM* имеются КЭШи *L1* и дополнительные устройства для вычислений с двойной точностью. Вне *SM* имеются ёмкие, но довольно медленные устройства глобальной *Global MEM* (общей для всего ускорителя *GPU*) и локальной *Local MEM* (для каждого мультипроцессора) памяти. На рисунке 3 не указана имеющаяся у каждого *SM* константная память, но о ней коротко будет сказано ниже.

На рисунке 4 показана связь между основным процессором вычислительной системы *CPU* со своей памятью и ускорителем на *GPU* с его памятью. В терминах *CUDA* одно устройство (с *CPU*) называется *host*, а другое (с *GPU*) – *device*.

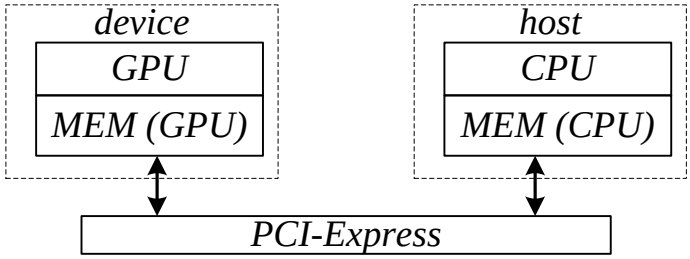


Рис. 4. Связь устройств *host* и *device*

Первым важным моментом является то, что программа, написанная в стиле *CUDA*, будет выполняться и на *host* и на *device*. Отсюда следует **второй важный момент**, что часть исходного текста программы должна быть скомпилирована для *host* стандартным компилятором *CPU*, а другая часть исходного текста программы для *device* должна быть скомпилирована компилятором *NVIDIA* для *GPU*. Такая компиляция производится автоматически специальными инструментариями *NVIDIA CUDA Toolkit*, разрабатываемыми под конкретные аппаратные и программные платформы. **Третьим важным моментом** на сегодня является программная (на уровне пользователя) взаимная недоступность памяти устройств *host* и *device*. Для этого необходимо средствами *CUDA* осуществлять явное перемещение данных из памяти одного устройства в память другого.

На рисунке 5 приведен схематический пример программы на языке *C* в стиле *CUDA*. В тексте программы имеются две функции: *main* и *FUN_KERNEL*. Функция *main* является стандартной, будет скомпилирована под *CPU* и запущена для выполнения на *host*. Функция *FUN_KERNEL* (предположим, мы так её решили назвать) будет скомпилирована

под *GPU*, и запущена на *device* после того как её специальным образом вызовет функция *main*. Функции, предназначенные для запуска на *device* из *host*-а, в терминах *CUDA* называются ядрами. То, что эта функция предназначена для выполнения на *device*, указывает специальный префикс *__global__*, который обозначает, что функция с её данными будет размещена в глобальной памяти ускорителя *GPU*.

```
#include<stdio.h>

//Функция-ядро, которая будет выполняться на device (GPU)
__global__ void FUN_KERNEL (формальные параметры функции-ядра)
{
        //Тело функции-ядра, Которое будет по умолчанию выполняться всеми
          запущенными потоками на GPU
}

// Функция main, которая будет выполняться на host (CPU)
int main ( int argc, char *  argv [] )
{
        ПОДГОТОВКА ДАНЫХ НА host

        //Выделение памяти на GPU
        cudaMalloc ( указатель памяти device, размер);
        // Копирование данных в память device из памяти host
        cudaMemcpy ( указатель памяти device, указатель памяти host, размер данных,
                        cudaMemcpyHostToDevice);

        // ЗАПУСК ЯДРА НА device
        FUN_KERNEL<<< параметры исполнения >>> (параметры функции-ядра);

        //Копирование данных в память host из памяти device
        cudaMemcpy ( указатель памяти CPU, указатель памяти GPU, размер данных,
                        cudaMemcpyDeviceToHost );

        ИСПОЛЬЗОВАНИЕ РЕЗУЛЬТАТОВ РАБОТЫ device НА host

        cudaFree   (указатель памяти device); //Очистка памяти на устройстве
        return 0;
}
```

Рис. 5. Схематический пример *CUDA*-программы

В функции main могут выполняться любые действия. Предположим, что результатом этих действий являются некоторые данные, которые надо обработать на *GPU*. Для этого с помощью функции *cudaMalloc* необходимо на *device* выделить память заданного размера и ассоциировать её с указателем. Потом с помощью функции *cudaMemcpy* осуществляется копирование

данных из памяти *host* в память *device* (указатели ставятся во второй и первый параметры функции соответственно), а также размер данных и ключевое слово *cudaMemcpyHostToDevice*, задающее направление копирования.

Ключевым моментом является вызов функции-ядра. Сначала пишется имя функции-ядра, потом в тройных угловых скобках параметры запуска (о них позже) и далее в круглых скобках фактические параметры самой функции. На время выполнения функции-ядра, функция *main* на *host* приостанавливает свою работу.

После исполнения функции-ядра, результаты работы копируются из памяти *device* в память *host*. Для этого снова используется функция *cudaMemcpy*, но с ключевым словом *cudaMemcpyDeviceToHost* в четвёртом параметре. Далее функция *main* может использовать результаты полученные с *GPU*.

Каким образом функция-ядро выполняется на *GPU*? Как уже упоминалось, ядро выполняется множеством потоков. Вся совокупность потоков, запускаемых для выполнения ядра, образует так называемую сетку – *grid*. Сетка (g*rid*) блоков поскольку вся совокупность потоков разбивается на блоки для выполнения на разных *SM*. Следует помнить, что потоки одного блока назначаются на один мультипроцессор *SM*, и, следовательно, у них есть возможность пользоваться для взаимодействия общей (s*hared*) памятью мультипроцессора.

На рисунке 6 схематически представлено выполнение ядра на *GPU*.

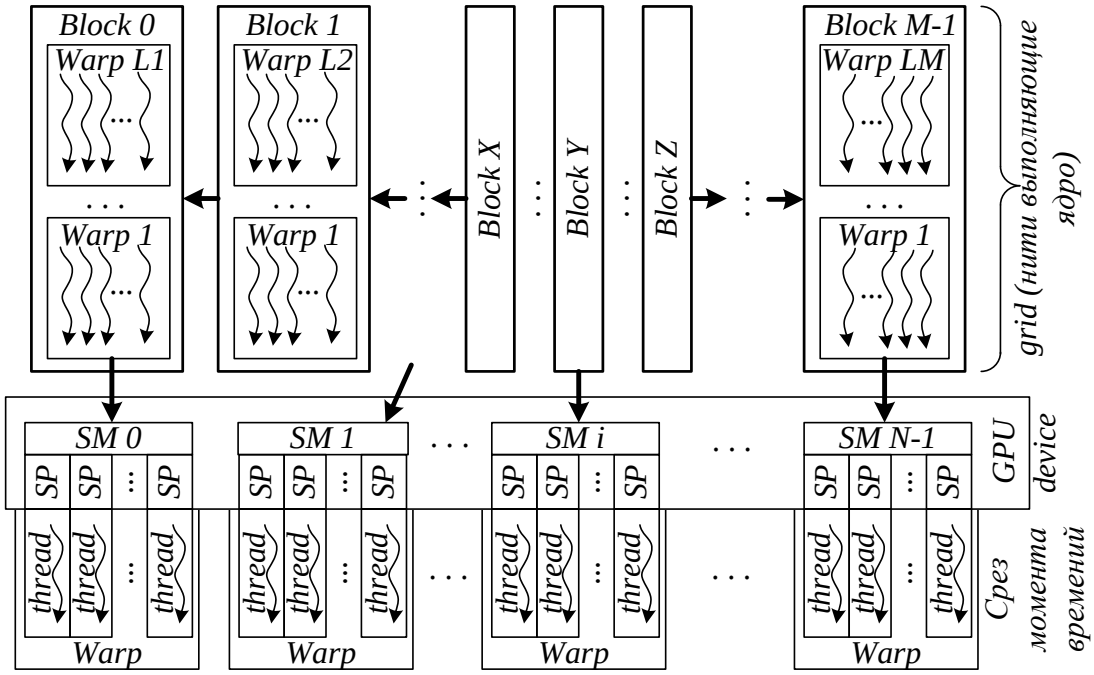


Рис. 6. Выполнение ядра на *GPU*

Для выполнения некоторого ядра запускается M блоков, в каждом блоке запускается некоторое количество потоков. Потоки каждого блока выполняются независимо от потоков других блоков. Потоки одного блока выполняются на SM по схеме SIMD. Как было отмечено, потоки выполняются на мультипроцессорах варпами. В классическом устройстве, одновременно на GPU может выполняться один варп потоков на одном SM, а всего число варпов равно числу единовременно задействованных SM.

На рисунке 7 приведён пример программы подсчёта суммы векторов.

```
#include<stdio.h>
#define N 4096

__global__ void FUN_KERNEL ( float * A, float * B, int S)
{
  int k;
  int idx_thread = blockIdx.x * blockDim.x + threadIdx.x;

  for (k=0;k<S;k++) A[idx_thread*S+k]=A[idx_thread*S+k]+B[idx_thread*S+k];
}

int main ( int argc, char *  argv [] )
{
      int         i, blocks, blocksize, steps;
      float       vect1[N],vect2[N];
      float       * devA, *devB;

  for (i=0; i<N; i++) { vect1[i]=i; vect2[i]=i;}

  cudaMalloc ( (void**)&devA, N * sizeof ( float ) ) ;
  cudaMalloc ( (void**)&devB, N * sizeof ( float ) ) ;

  blocks=4; blocksize=64;
  steps=(int)N/(blocks*blocksize);

  cudaMemcpy ( devA, vect1, N * sizeof ( float ), cudaMemcpyHostToDevice);
  cudaMemcpy ( devB, vect2, N*sizeof ( float ), cudaMemcpyHostToDevice);

  FUN_KERNEL<<<blocks,blocksize>>> ( devA, devB, steps);

  cudaMemcpy ( vect1, devA, N * sizeof ( float ), cudaMemcpyDeviceToHost );
  cudaFree   ( devA  );

  for (i = 0; i < N; i++)  printf("vect1[%d] = %.5f\n", i, vect1[i]);
  return 0; }
```

Рис. 7. Пример программы сложения двух векторов

На устройстве *host*, в функции *main* осуществляется формирование содержимого векторов *vect1* и *vect2*. С помощью функций *cudaMalloc* на устройстве *device* выделяется память для обоих векторов, после чего *host* возвращаются указатели на эти области (*devA* и *devB*). Следует иметь в виду, что части программы, выполняемые на *host*, не могут (об этом уже говорилось) адресовать память на *device* и наоборот, части программы, выполняемые на *device*, не могут адресовать память на *host*. Поэтому данные должны быть скопированы средствами *CUDA* (функция *cudaMemcpy*) в память *device*. По окончании вычислений, с помощью этой же функции, результаты копируются в память *host*.

При вызове ядра (*FUN_KERNEL*) были установлены следующие параметры исполнения: количество блоков потоков (переменная *blocks* = 4) и количество потоков в каждом блоке (переменная *blocksize* = 64). В качестве параметров, функции-ядру передаются указатели на вектора в памяти *device* (переменные *devA* и *devB*) и количество элементов векторов, которые должен обработать один поток (значение переменной *steps*).

Сама функция-ядро *FUN_KERNEL* будет выполняться всеми запущенными на *device* потоками по принципу *SPMD*. Для каждого потока будут созданы свои экземпляры переменных, объявленных внутри функции (*k* и *idx_kernel*). Складываемые вектора будут находиться в глобальной памяти *device*, и к ним будут иметь доступ все потоки. Указатели на эти вектора, функции *FUN_KERNEL* передаются через параметры *A* и *B* при её вызове. Через параметр *S* передаётся количество элементов, которое должен обработать каждый поток.

Для определения номера потока в задаче в ядре используются следующие собственные структурные переменные среды *CUDA*: *blockIdx.x* – номер блока в задаче; *blockDim.x* – размер блока (т.е. сколько в нем потоков); *threadIdx.x* – номер потока в текущем блоке. Блоки в задаче могут быть многомерными. Пока мы используем только одномерный вариант, поэтому берём значения лишь по измерению *x*.

На рисунке 8 приведён пример функции *main*, в которой использованы средства *CUDA* для замера времени. С помощью функций *cudaEventRecord* фиксируются события начала (*start*) и конца (*stop*) процедуры вычислений. Предварительно события необходимо инициализировать функцией *cudaEventCreate*. К сожалению, *host* не всегда дожидается окончания функционирования *device*, поэтому для правильной фиксации события необходимо произвести синхронизацию функцией *cudaEventSynchronize*. Определить время между событиями позволяет функция *cudaEventElapsedTime*. Эта функция сохраняет время в переменной *elapsedTime* (тип *float*).

```
int main ( int argc, char *  argv [] )
{
      int i,blocks,blocksize,steps;
                  float vect1[N],vect2[N];
                  FILE *f;
                  float * devA, *devB, elapsedTime;

                  cudaEvent_t start, stop; //Идентификаторы событий

 cudaEventCreate(&start);  //Инициализация события start
 cudaEventCreate(&stop);  //Инициализация события stop

 for (i=0; i<N; i++) { vect1[i]=i; vect2[i]=i;}

 cudaMalloc ( (void**)&devA, N * sizeof ( float ) ) ;
 cudaMalloc ( (void**)&devB, N * sizeof ( float ) ) ;

 blocks=16; blocksize=128;
 steps=(int)N/(blocks*blocksize);

 cudaEventRecord(start,0); // Фиксация события start
 cudaMemcpy ( devA, vect1, N * sizeof ( float ), cudaMemcpyHostToDevice);
 cudaMemcpy ( devB, vect2, N*sizeof ( float ), cudaMemcpyHostToDevice);

 FUN_KERNEL<<<blocks,blocksize>>> ( devA, devB, steps);

 cudaMemcpy ( vect1, devA, N * sizeof ( float ), cudaMemcpyDeviceToHost );
 cudaFree   ( devA  );

 cudaEventRecord(stop,0);  // Фиксация события stop
 cudaEventSynchronize(stop); // Синхронизация host и device по событию stop

 // Определение времени (в миллисекундах)  между событиями start и stop
 cudaEventElapsedTime(&elapsedTime,start,stop);

 printf("time1 = %f\n", elapsedTime); //Вывод времени
 for (i = 0; i < N; i++)  printf("vect1[%d] = %.5f\n", i, vect1[i]);

 return 0;
}
```

Рис. 8. Пример функции *main* с замером времени

Теперь о памяти *GPU*. Всего в *GPU* имеется шесть видов памяти:

– регистровая (*register*), это быстродействующая, программно недоступная память;

– константная (*constant*), это быстродействующая память, запись в которую можно осуществлять только с *host*. Значения данных, загруженных в константную память, потоки на *device* могут использовать, но не могут изменять;

– общая (*shared*), это небольшая быстродействующая внутрикристальная память у каждого *SM*, доступная для записи и чтения всем потоковым процессорам *SP* мультипроцессора *SM*;

– текстурная (*texture*), это общая для мультипроцессоров *SM* одного *TPC* память. Она предназначена для хранения текстур графического процессора, но при вычислениях может быть использована как особый вид константной памяти;

– глобальная (*global*), это общедоступная для всего *GPU* память. Её ёмкость может измеряться гигабайтами (именно это значение указывается в качестве ёмкости видеопамяти). Время доступа к ней достаточно велико, поэтому при вычислениях она часто используется как некий базовый накопитель, откуда инструкции и данные считываются в кэши и другие модули памяти, а после в ней сохраняется результат работы для передачи его в память *host*;

– локальная (*local*), это внешняя, достаточного ёмкая, но медленная память каждого отдельного мультипроцессора, в которой каждому потоку выделяются разделы для хранения его локальных данных (в частности, содержимое переменных локальных функций).

Домашняя подготовка

1. Прочитать задание на лабораторную работу.
2. Разработать и написать программы согласно заданию.

Задание на лабораторную работу

1. Используя специальную программу (входящую в состав ОС или предоставленную преподавателем), определить параметры имеющегося у Вас ускорителя *GPU* (*device*).

2. Разработать, опираясь на примеры рис. 7 и рис. 8, программу на *CUDA*, реализующую вычисления из табл. 2 (вариант задания берётся на основе номера из журнала лабораторных работ).

3. Разработать аналогичный п. 2. вариант программы только для *CPU*, предусматривающий чисто последовательную обработку. Для замера времени выполнения вычислений использовать средства *CUDA***.**

4. Выполнить программу п. 3. два раза для размеров векторов 16384 и 65536 элементов. Время выполнения вычислений занести в протокол. Будем называть эти времена – временами последовательного выполнения на *host*.

5. Выполнить программу п. 2. два раза для размеров векторов 16384 и 65536 элементов. Для выполнения ядра запустить один блок с одним потоком. Времена выполнения вычислений занести в протокол как времена последовательного выполнения на *device*.

6. Провести для каждого из двух размеров векторов (16384 и 65536 элементов) серию из 35 запусков программы п. 2, запуская для выполнения ядра 1, 2, 4, 8 и 16 блоков, с 1 (кроме случая, когда один блок), 4, 32, 64, 256, 2048 нитями в блоке. Все времена занести в протокол.

7. Составить отчёт, содержащий:

– параметры ускорителя (*device*), который был использован для экспериментов;

– вариант задания на ЛР;

– исходные тексты программ п. 2 и п. 3.;

– для каждого из двух размеров векторов привести по две таблицы. Таблицу с коэффициентами ускорения многопоточных (параллельных) вариантов исполнения вычислений к последовательному на *device*. Таблицу с коэффициентами ускорения многопоточных (параллельных) вариантов исполнения вычислений к последовательному на *host*. Для формирования каждой из четырёх таблиц выбрать любые 8 наиболее представительных (по Вашему мнению) результатов п. 6;

– на основе таблиц коэффициентов ускорений построить графики.

Примечание: Для ускорения работы, ядро в функции *main* можно запускать несколько раз с разными параметрами исполнения.

Таблица 2

№ бригады	Задание
1	$\mathbf{Y}_i=\mathbf{A}_i\mathbf{B}_i+\mathbf{A}^m{}_i\mathbf{C}_i/\mathbf{B}_i$
2	$\mathbf{Y}_i=(\mathbf{A}_i+\mathbf{C}_i)/(\mathbf{B}^m{}_i-\mathbf{D}_i)$
3	$\mathbf{Y}_i=\mathbf{A}_i(\mathbf{B}_i\mathbf{C}_i+13)/S1^m$
4	$\mathbf{Y}_i=\mathbf{C}_i/(\mathbf{A}_i+\mathbf{D}_i)+S1\mathbf{B}^m{}_i$
5	$\mathbf{Y}_i=(S1^m-3\mathbf{B}_i)/(\mathbf{A}_i+S2)$
6	$\mathbf{Y}_i=\mathbf{B}_i\mathbf{D}_i/(\mathbf{D}_i+\mathbf{E}_i)^m+\mathbf{A}_i\mathbf{C}_i$
7	$\mathbf{Y}_i=(\mathbf{C}^m{}_i-S2)/(\mathbf{A}_i+\mathbf{D}_i\mathbf{D}_i)+S1$
8	$\mathbf{Y}_i=\mathbf{A}_i(\mathbf{B}_i\mathbf{C}_i+34*\mathbf{A}^m{}_i)/S1^m$

Примечания к табл. 2:

1) **A**, **B**, **C**, **D**, **E** и **Y** – векторы, S1 и S2 – скаляры;

2) *m* – степень (от 2 до 5), конкретное значение которой задаёт преподаватель

Контрольные вопросы

1. Что такое *CUDA*?
2. В чём состоят отличия *GPU* от *CPU*? Что такое *host* и *device*?
3. В чем заключается структура и состав GPU?
4. Какие виды памяти имеются у *GPU*? Каково их назначение?
5. Что такое ядро? Каким образом можно вызвать ядро?
6. Что такое сетка (*grid*), блок (*block*), варп (*warp*) и нить (*thread*)? Как выполняются потоки на *GPU*?
7. Как идентифицируются потоки и блоки?

Лабораторная работа №3

ПРОГРАММИРОВАНИЕ ГРАФИЧЕСКИХ ПРОЦЕССОРОВ СРЕДСТВАМИ *NVIDIA CUDA*. РАЗМЕЩЕНИЕ ДАННЫХ В ГЛОБАЛЬНОЙ, РАЗДЕЛЯЕМОЙ И КОНСТАНТНОЙ ПАМЯТИ, ВРЕМЯ ДОСУПА К ДАННЫМ

Цель лабораторной работы заключается в приобретении навыков работы с памятью *GPU* средствами *CUDA*, и экспериментов с доступом к данным. Предыдущая лабораторная работа познакомила с основами программирования графических ускорителей *NVIDIA* с технологией *CUDA*. Данная лабораторная работа посвящена изучению трёх видов памяти графического ускорителя – глобальной, разделяемой и константной.

Глобальная память самая ёмкая из всех, она доступна всем потоковым мультипроцессорам (*SMP*), т.е. всем потокам задачи, выполняющимся на потоковых процессорах (*SP*) – ядрах мультипроцессоров. При этом глобальная память является и самой медленной. В отличие от глобальной памяти (назовём её *globa*), разделяемая память (назовём её *shared*) разбита на модули, доступные только потокам одного блока задачи. Это происходит потому, что каждый блок потоков выполняется на одном *SMP,* и доступ к встроенному модулю *Shared MEM* имеют только его потоки, которые выполняются на ядрах *SP* данного мультипроцессора. *Shared MEM* достаточно быстрая, но небольшая память. Типичная её ёмкость составляет всего 48–64 Кбайт **на один модуль.** Главной особенностью константной памяти (назовём её *constant*) является то, что её содержимое не может быть изменено потоками задачи. Она тоже разделена на модули, но они хранят идентичное, доступное потокам всех блоков содержимое.

Схема доступа к рассматриваемой нами памяти графического ускорителя показана на рис. 9. Здесь собственные процессоры и память основного модуля компьютера обозначены как *host*, а графический ускоритель как *device*. Чтобы загрузить данные в *global* или выгрузить из неё результат, используется функция *cudaMemcpy*. Чтобы загрузить данные в *constant*, используется функция *cudaMemcpyToSymbol*. Прямого доступа с *host* в *shared* нет, поэтому данные сначала загружаются в глобальную память, а потом переносятся в разделяемую простым копированием.

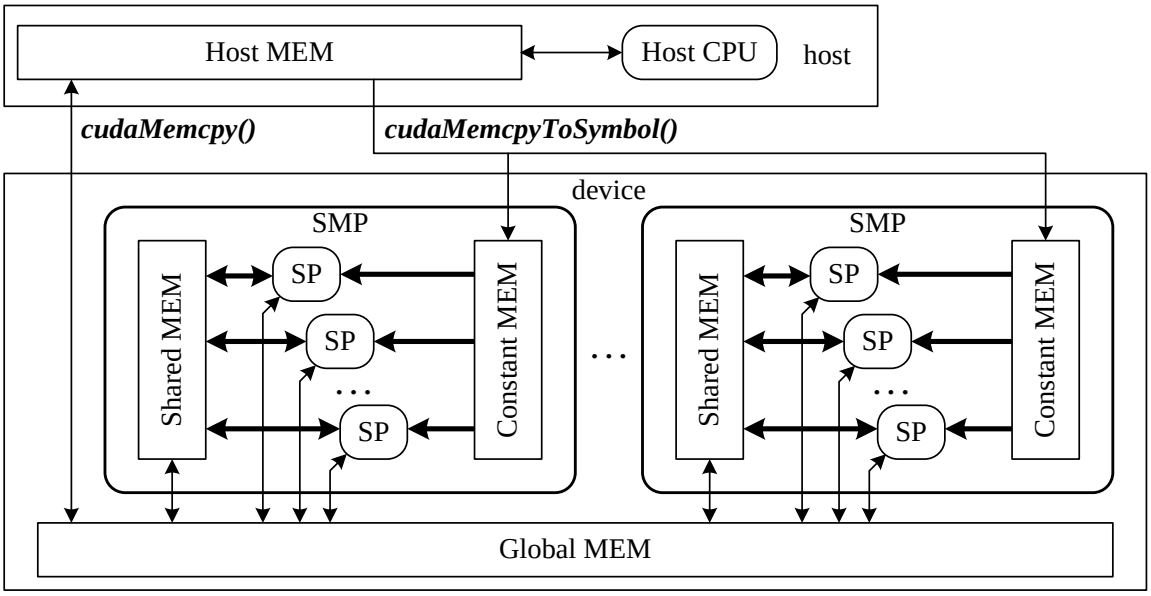


Рис. 9. Схема доступа к памяти

На рисунке 10 приведён фрагмент ядра *CUDA* – программы с простым поэлементным копированием потоком *threadIdx.x* своих *s* элементов данных из глобальной памяти (массив *A*) в распределённую (массив *as*) и наоборот.

```
__global__ void fun_kernel( float * a, int s)
{
        __shared__ float as [NS];  //где NS*sizeof(float) < 48 Кб

for ( int k = s*threadIdx.x; k < (s+1)*threadIdx.x; k++ )
   as [k]=A[k];

        __syncthreads();
        // Работа с данными в разделяемой памяти
        __syncthreads();

for ( int k = s*threadIdx.x; k < (s+1)*threadIdx.x; k++ )
   A[k]=as[k];
}
```

Рис. 10. Пример ядра *CUDA*, работающего с разделяемой памятью

Важно отметить две детали. Чтобы массив *as* был создан в разделяемой памяти, его объявление необходимо снабдить префиксом *__shared__*. Чтобы не было рассогласований между потоками при работе с разделяемой памятью, часто бывает необходимо производить синхронизацию потоков функцией *__syncthreads*.

Выполняя данную лабораторную работу, будем считать, что нам доступно только 16 Кб в каждом модуле *shared*, поэтому, работая с большими массивами, необходимо будет их делить на части, чтобы они вместились в указанную ёмкость. При этом, чтобы выборка из памяти была наиболее эффективна, необходимо соблюдать правила блочного запроса – *coalescing* (см. лекционный материал или литературу [1–3]).

Чтобы загрузить данные в константную память, надо сначала напрямую (глобально, вне функций) в программе объявить массив, например *ac,* с префиксом *__constant__*, а затем с помощью функции *cudaMemcpyToSymbol* скопировать данные из памяти *host*-а (*Host MEM*) в *constant*. На рисунке 11 приведён фрагмент программы, демонстрирующий загрузку данных в константную память.

```
__constant__ float ac[N];
…
__global__ void fun_kernel() // Функция-ядро
…
int main()
{
   float Ahost[N]; //массив данных на host-е
   …
   cudaMemcpyToSymbol(ac, Ahost, sizeof(float)*N);
   …
  <вызов ядра fun_kernel() >
   …
}
```

Рис. 11. Код загрузки данных в константную память

Домашняя подготовка

1. Изучить материалы данного пособия, касающиеся работы с памятью *global*, *shared* и *constant*;

2. Заготовить протокол с табл. 4 и 5.

Задание на лабораторную работу

1. Написать на языке *C* и отладить исходную программу для *host*, реализующую обработку данных согласно варианту из табл. 3.

2. Задать число $N = 512$. Для записи результатов работы подготовить таблицу протокола по шаблону (табл. 4).

3. Переписать исходную программу в программу для *CUDA* так, чтобы в процессе вычислений на *device* все массивы хранились в глобальной памяти.

4. Выполнить вычисления программы п. 3 одним, двумя и четырьмя блоками, по 1, 4, 32, 128 потоков в каждом. Замерить времена выполнения и занести их в таблицу протокола.

5. Переписать исходную программу для *CUDA* так, чтобы в процессе вычислений на *device* все массивы хранились в разделяемой (*shared*) памяти (не забывайте про *coalescing*).

6. Выполнить вычисления программы п. 5 одним, двумя и четырьмя блоками, по 1, 4, 32, 128 потоков в каждом. Замерить времена выполнения и занести их в таблицу протокола.

7. Переписать исходную программу для *CUDA* так, чтобы в процессе вычислений на *device* все массивы кроме *X* хранились в константной (*constant*) памяти, а сам *X* – в разделяемой (*shared*) памяти.

8. Выполнить вычисления программы п. 7 одним, двумя и четырьмя блоками, по 1, 4, 32, 128 потоков в каждом. Замерить времена выполнения и занести их в таблицу протокола.

9. Задать число $N = 65536$. Для записи результатов работы подготовить ещё одну таблицу протокола по шаблону (табл. 4).

10. Переписать программы пп. 5 и 7 таким образом, чтобы, считая доступной для хранения массивов ёмкость разделяемой (*shared*) памяти по 16 Кб на блок, можно было разместить и эффективно использовать данные из массивов. Для этого надо рассчитать, сколько элементов массивов одновременно может храниться в памяти и организовать загрузку и выгрузку данных и результатов между глобальной памятью и разделяемой памятью.

11. Выполнить программы пп. 3 и 10 (при $N = 65536$) одним, двумя и четырьмя блоками, по 1, 4, 32, 256 потоков в каждом. Замерить времена выполнения и занести их в таблицу протокола.

12. Составить сравнительную таблицу времён выполнения вычислений (табл. 5).

Таблица 3

№ бригады	Вариант задания

1	#define M=**45*< номер вашей студенческой группы >**
	
	int A[N];
	float B[N],X[N]; char C[N];
	. . .
	for (i=0; i<M; i++)
	  for (j=0; j<N; j++)
	  X[j]=*(float)* A[j]*X[j]+(B[j]-X[j])/C[j];

2	#define M=**60*< номер вашей студенческой группы >**
	
	int A[N], B[N];
	double X[N], C[N];
	. . .
	for (i=0; i<N; i++)
	  for (j=0; j<M; j++)
	  X[i]=*(double)* A[i]*(X[i]+B[i])/C[i];

3	#define M=**31*< номер вашей студенческой группы >**
	
	char A[N], B[N];
	float X[N], C[N], D[N];
	. . .
	for (i=0; i<N; i++)
	  for (j=0; j<M; j++)
	  X[i]=*(float)* A[i]*(X[i]+B[i])+X[i]/(C[i]+ D[i]);

4	#define M=**< номер вашей студенческой группы >+879**
	
	int A[N], B[N], C[N];
	double X[N];
	. . .
	for (i=0; i<3*M; i++)
	  for (j=0; j<N; j++)
	  X[j]=*(double)* (A[j]*X[j]+C[j]*X[j])/B[j];

5	#define M=**35*< номер вашей студенческой группы >-12**
	
	int X[N], B[N];
	char A[N], C[N], D[N];
	. . .
	for (i=0; i<N; i++)
	  for (j=0; j<M; j++)
	  X[i]=B[i]*(X[i]+A[i]+C[i]) + X[i]*D[i]);

Окончание табл. 3

6	#define M=**21*< номер вашей студенческой группы >+234**
	
	char A[N], B[N], C[N];
	double X[N], D[N];
	. . .
	for (i=0; i<N; i++)
	  for (j=0; j<2*M; j++)
	  X[i]=*(double)* A[i]*X[i]*(X[i]*C[i]+B[i])/D[i];

7	#define M=**< номер вашей студенческой группы >+956**
	
	int A[N], B[N], X[N];
	float C[N]; char D[N];
	. . .
	for (i=0; i<N; i++)
	  for (j=0; j<4*M; j++)
	  X[i]=*(int)* X[i]*((*float)* (C[i]+B[i])/(X[i]*A[i]+D[i]));

8	#define M=**87*< номер вашей студенческой группы >**
	
	int B[N], C[N];
	float X[N], A[N]; char D[N];
	. . .
	for (i=0; i<M; i++)
	  for (j=0; j<N; j++)
	  X[j]=*(float)* (A[j]*X[j]+C[j]*X[j] /(B[j]- D[j]);

Таблица 4

Времена выполнения вычислений на *GPU* при *N*=…												
	(число блоков / число потоков в блоке)											
	(1/1)	(1/4)	(1/32)	(1/256)	(2/1)	(2/4)	(2/32)	(2/256)	(4/1)	(4/4)	(4/32)	(4/256)
Только глобальная память												
Разделяемая память												
Разделяемая и константная память												

Таблица 5

		(число блоков / число потоков в блоке)											
N	Память	(1/1)	(1/4)	(1/32)	(1/256)	(2/1)	(2/4)	(2/32)	(2/256)	(4/1)	(4/4)	(4/32)	(4/256)
N=512	Glob	1	1	1	1	1	1	1	1	1	1	1	1
	Shar												
	S&C												
N = 65536	Glob	1	1	1	1	1	1	1	1	1	1	1	1
	Shar												
	S&C												

Контрольные вопросы

1. Какие виды памяти в *CUDA* Вы знаете, чем и как они характеризуются?

2. Где находится *global* память и для чего она используется?

3. Где находится *shared* память и для чего она используется?

4. Где находится *constant* память и для чего она используется?

5. Режимы доступа к разным типам памяти в *CUDA*. Каким потокам доступны *global*, *shared* и *constant* память?

6. Что такое *coalescing* и как его организовать?

Лабораторная работа № 4

ПРИМЕНЕНИЕ ГРАФИЧЕСКИХ ПРОЦЕССОРОВ ДЛЯ РЕШЕНИЯ ЗАДАЧ ОБРАБОТКИ ИЗОБРАЖЕНИЙ

Предыдущие лабораторные работы были посвящены изучению основ программирования графических ускорителей *NVIDIA* с технологией *CUDA*. Данная лабораторная работа посвящена применению *GPU* для решения прикладных задач. В качестве примеров подобных задач были выбраны задачи обработки изображений. Сделанный выбор обусловлен тем, что задачи обработки изображений принадлежат к классу задач, для ускорения решения которых применяются графические процессоры (*GPU*).

В данной лабораторной работе студентам предлагается изучить алгоритмы обработки полутоновых двумерных изображений и реализовать их в программах для центральных процессоров (*CPU*) и графических процессорах (*GPU*) архитектуры *NVIDIA CUDA*.

Программы, которые должны быть разработаны в ходе выполнения данной лабораторной работы, должны загружать входное изображение из файла, провести обработку загруженного изображения алгоритмом согласно варианту задания, и записать полученный результат в другой файл. Таким образом, результаты работы программы можно будет просмотреть с помощью графического редактора.

Общие сведения о форматах файлов

В данной лабораторной работе входными данными должны быть полутоновые чёрно-белые изображения. Значение яркости каждого пикселя представлено беззнаковым целым восьмибитным числом (следовательно, диапазон изменения яркости пикселя от 0 до 255).

Обозначим через $I(x, y)$ яркость пикселя с координатами (x, y). Координаты изменяются соответственно от 0 до W-1 и от 0 до H-1, где W и H – соответственно ширина и высота изображения в пикселях.

Изображение хранится в памяти ЭВМ в виде двумерного массива. Поэтому, если `data` – указатель на массив, содержащий данные изображения, то элемент этого массива с индексом $x + yw$ (или `data[x+y*w]`, в нотации языка *C*) содержит значение пикселя с координатами (x, y).

Для обмена данными можно использовать формат Targa. Функции чтения и записи в файл такого формата приведены в приложении.

Домашняя подготовка

1. Выбрать алгоритм согласно варианту задания. Изучить выбранный алгоритм.

2. Разработать программы для *CPU* и *GPU*, реализующие выбранный алгоритм. Программы должны считывать входное изображение из файла и записывать результат в другой файл. Разработанные программы должны работать с изображениями произвольных разрешений до *FullHD* (1920x1080 пикселей) включительно. Программа для *GPU* также должна принимать на вход в качестве аргумента количество нитей и блоков, которые следует запустить на *GPU*.

Задание на лабораторную работу

1. Запустить разработанные программы. Измерить времена выполнения обработки изображений на *GPU* и на *CPU* (при замерах следует учитывать только время выполнения алгоритма, времена загрузки данных из файла и записи данных в файл не учитываются). Для программы, реализующей вычисления на *GPU*, необходимо измерить не только время выполнения ядра, но и время выполнения обработки с учётом обмена данными между *GPU* (*device*) и *CPU* (*host*).

2. Добиться ускорения обработки изображения разрешения *FullHD* на *GPU* (с учётом обмена данными между *device* и *host*) по сравнению с обработкой на *CPU*. Для этого может потребоваться как модификация программы, разработанной для *GPU* (например, использование разделяемой памяти или константной памяти, рассмотренных в предыдущей лабораторной работе), так и подбор количества нитей и блоков, на котором запускается ядро.

3. Измерить времена выполнения обработки изображений на *GPU* и на *CPU* для файлов разных размеров. Также следует вычислить коэффициент ускорения выполнения программы на *GPU* относительно *CPU*, а также долю времени, которую занимает обмен данными при решении задачи на *GPU*. Результаты измерений привести в таблице.

Указания к выполнению задания

1. Некоторые алгоритмы для вычисления значения яркости пикселя результирующего изображения требуют также значения яркости соседних пикселей исходного изображения. Чтобы получить результат для пикселей, находящихся на границах исходного изображения, надо

доопределить значения яркости для координат, выходящих за границы изображения. Будем считать, что яркость такой точки равна яркости ближайшей точки изображения. Например, $I(-1,1) = I(0,1)$; $I(W,1) = I(W-1,1)$.

2. Поскольку яркость пикселя задаётся беззнаковым целым восьмибитным числом (тип данных `unsigned char` в C/C++), промежуточные вычисления следует проводить, используя типы `float` или `int` (в зависимости от задачи).

3. При записи вычисленных значений в результирующее изображение следует помнить, о том, что яркость пикселей результирующего изображения имеет тип `unsigned char`, а значит, может принимать значения только от 0 до 255. Поэтому перед записью результата отрицательные значения надо заменить на 0, а значения, большие 255 – на 255.

Варианты задания

1. Свёртка.

Свёртка изображения с заданным ядром задаётся формулой:

$$R(x,y) = \sum_{j=-n}^{n} \sum_{i=-n}^{n} I(x+i, y+j) \cdot k(i,j),$$

где x, y – координаты пикселя;

$R(x,y)$ – значение яркости пикселя результирующего изображения;

$I(x,y)$ – значение яркости пикселя исходного изображения;

k – ядро свёртки, как правило задаётся квадратной матрицей K, размерностью $N = 2n+1$, причём, $k(i,j) = K(i+n, j+n)$.

Например, чтобы заменить значение яркости пикселя на средние значения яркости соседних пикселей, надо применить свёртку со следующим ядром: $\begin{pmatrix} 0{,}125 & 0{,}125 & 0{,}125 \\ 0{,}125 & 0 & 0{,}125 \\ 0{,}125 & 0{,}125 & 0{,}125 \end{pmatrix}$.

Задание: запрограммировать алгоритм свёртки исходного изображения со следующим ядром: $\begin{pmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{pmatrix}$. Это один из наиболее простых алгоритмов повышения чёткости изображения.

2. Свёртка с разделяемым ядром.

Определение свёртки (см. в описании варианта 1). Если матрицу K можно представить в следующем виде:

$$K(i, j) = F_1(i) \cdot F_2(j),$$

то говорят, что ядро K разделимо. Подставим эту формулу в определение свёртки:

$$R(x, y) = \sum_{j=-n}^{n} \sum_{i=-n}^{n} I(x+i, y+j) \cdot k(i, j) = \sum_{j=-n}^{n} \sum_{i=-n}^{n} I(x+i, y+j) \cdot f_1(i) \cdot f_2(j) =$$

$$= \sum_{j=-n}^{n} f_2(j) \sum_{i=-n}^{n} I(x+i, y+j) \cdot f_1(i),$$

где $f_1(i) = F_1(i+n)$, $f_2(j) = F_2(j+n)$.

Это означает, что вычисление свёртки можно разбить на два этапа: свёртка по строкам и свёртка по столбцам. На первом этапе вычисляются промежуточные значения

$$M(x, y) = \sum_{i=-n}^{n} I(x+i, y) \cdot f_1(i),$$

на втором – вычисляется свёртка:

$$R(x, y) = \sum_{j=-n}^{n} M(x, y+j) \cdot f_2(j).$$

Такое разбиение позволяет ускорить процесс вычисления свёртки, так как этом случае потребуется всего $2N$ умножений на пиксель. В случае неразделимого ядра количество умножений составило бы N^2 на пиксель.

Задание: запрограммировать алгоритм Гауссова размытия изображения. Этот фильтр используется для подавления шума.

Данный фильтр реализуется свёрткой с разделяемым ядром следующего вида:

$$k(i, j) = g(i) \cdot g(j),$$

где $g(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$ – функция Гаусса; σ – радиус размытия.

Для практических целей $g(x)$ считают равной нулю при $|x| > 6\sigma$. Поэтому, если для реализации алгоритма взять $\sigma = 3$, то размер ядра составит 19 на 19 (так как $3\sigma + 3\sigma + 1 = 19$).

3. Нерезкое маскирование (*unsharp masking*).

Нерезкое маскирование – это способ визуально сделать изображение более чётким. Основная идея состоит в вычитании из исходного изображения нерезкой составляющей (т.е. размытого изображения, умноженного на некоторый коэффициент). Яркость пикселя результирующего изображения вычисляется по следующей формуле:

$$I_U = I_O + a(I_O - I_B),$$

где I_U – яркость пикселя результирующего изображения (от *unsharp*);
I_O – яркость пикселя исходного изображения (от *original*);
I_B – яркость пикселя размытого изображения (от *blurred*);
a – коэффициент «силы» размытой составляющей (от *amount*), как правило $0 \le a \le 1$.

Таким образом, в результирующем изображении нерезкая составляющая будет замаскирована.

Задание: реализовать алгоритм нерезкого маскирования. В качестве алгоритма размытия изображения использовать усреднение пикселей в квадратной окрестности размером 5 на 5 пикселей. Значение коэффициента *a* должно задаваться из командной строки.

4. Дискретное косинусное преобразование.

Пусть F – матрица исходных пикселей изображения, размером N на N, C – результат дискретного косинусного преобразования, матрица частот, размером N на N. Тогда:

$$C = D \cdot F \cdot D^T,$$

где D – матрица коэффициентов дискретного косинусного преобразования, размером N на N. Коэффициенты матрицы D определяются по следующей формуле:

$$D_{ij} = k(j)\cos\left(j(i+0{,}5)\frac{\pi}{N}\right),$$

где $k(j) = \begin{cases} \frac{1}{\sqrt{N}}, j = 0 \\ \frac{\sqrt{2}}{\sqrt{N}}, j \neq 0 \end{cases}$.

Обратное преобразование осуществляется по следующей формуле:

$$F = D^T \cdot C \cdot D.$$

Дискретное косинусное преобразование применяется, например, в алгоритме сжатия *JPEG*. Коэффициенты, полученные после дискретного косинусного преобразования, подвергаются квантованию, т.е., делятся на коэффициенты матрицы квантования и округляются до ближайшего целого числа. За счёт этого происходит обнуление коэффициентов, отвечающих за высокочастотные составляющие сигнала, что позволяет достичь значительного сжатия данных при хранении таких коэффициентов. При декодировании сохранённые коэффициенты умножаются на коэффициенты матрицы квантования, а затем проводится обратное дискретное косинусное преобразование.

Если значение яркости каждого пикселя представлено беззнаковым целым числом, то перед выполнением преобразования из значения яркости вычитают половину максимального значения для данного диапазона (128 для восьмибитного значения яркости). Соответственно, после выполнения обратного преобразования, к полученному результату следует прибавить 128.

Задание: провести моделирование кодирования и декодирования по алгоритму *JPEG*. Для каждого блока изображения 8x8 пикселей:

– провести дискретное косинусное преобразование;

– провести квантование полученных коэффициентов согласно матрице:

$$\begin{pmatrix} 16, & 11, & 10, & 16, & 24, & 40, & 51, & 61 \\ 12, & 12, & 14, & 19, & 26, & 58, & 60, & 55 \\ 14, & 13, & 16, & 24, & 40, & 57, & 69, & 56 \\ 14, & 17, & 22, & 29, & 51, & 87, & 80, & 62 \\ 18, & 22, & 37, & 56, & 68, & 109, & 103, & 77 \\ 24, & 35, & 55, & 64, & 81, & 104, & 113, & 92 \\ 49, & 64, & 78, & 87, & 103, & 121, & 120, & 101 \\ 72, & 92, & 95, & 98, & 112, & 100, & 103, & 99 \end{pmatrix};$$

– провести обратное дискретное косинусное преобразование.

5. Выделение границ: оператор Робертса.

Выделение границ на изображении применяется, например, для решения задач машинного зрения. Граница объекта на изображении характеризуется резким изменением значения яркости пикселя. Поэтому методы выделения границ в изображении основываются на вычислении градиента изображения.

Одним из самых простых методов выделения границ на изображении является оператор Робертса. Яркость пикселя результирующего изображения вычисляется по формуле:

$$R(x,y)=\sqrt{\left(I(x,y)-I(x+1,y+1)\right)^2+\left(I(x+1,y)-I(x,y+1)\right)^2},$$

где x, y – координаты пикселя;

$R(x,y)$ – значение яркости пикселя результирующего изображения;

$I(x,y)$ – значение яркости пикселя исходного изображения.

В результирующем изображении пиксели, находящиеся на границах областей с плавным изменением яркости, будут белыми. Остальные пиксели будут чёрными.

Задание: реализовать алгоритм выделения границ, который использует оператор Робертса.

6. Увеличение изображения: билинейная интерполяция.

Билинейная интерполяция – это расширение алгоритма линейной интерполяции на двумерный случай, когда интерполяция производится по одному измерению, а затем по другому. Схема билинейной интерполяции показана на рис. 12.

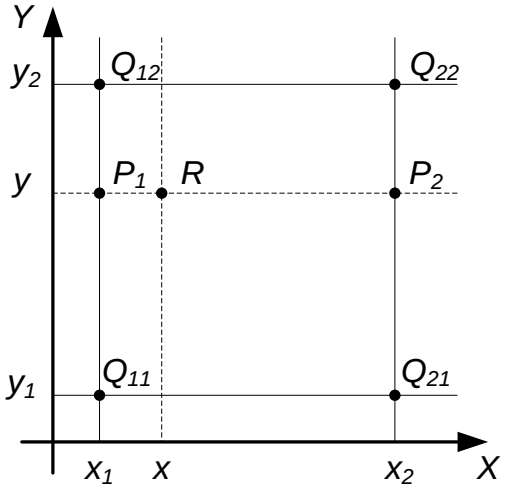


Рис. 12. Схема билинейной интерполяции

Для того, чтобы узнать приближённое значение $F(R) = F(x, y)$, надо сначала вычислить приближённые значения $F(P_1)$ и $F(P_2)$. Они вычисляются методом линейной интерполяции:

$$F(P_1) = F(x_1, y) \approx \frac{y_2 - y}{y_2 - y_1} F(Q_{11}) + \frac{y - y_1}{y_2 - y_1} F(Q_{12});$$

$$F(P_2) = F(x_2, y) \approx \frac{y_2 - y}{y_2 - y_1} F(Q_{21}) + \frac{y - y_1}{y_2 - y_1} F(Q_{22}).$$

Затем $F(R)$ вычисляется линейной интерполяцией значений $F(P_1)$ и $F(P_2)$:

$$F(R) = F(x, y) \approx \frac{x_2 - x}{x_2 - x_1} F(P_1) + \frac{x - x_1}{x_2 - x_1} F(P_2).$$

В простейшем случае увеличение изображения в N раз алгоритмом билинейной интерполяции проводится следующим образом: для каждой пары соседних пикселей в столбце исходного изображения вычисляются N-1 пикселей увеличенного изображения, затем, аналогичным способом вычисляются значения пикселей в строках. Таким образом, для исходного изображения размером W пикселей в ширину и H пикселей в высоту, интерполированное изображение будет иметь размер $(W-1)N+1$ пикселей в ширину и $(H-1)N+1$ пикселей в высоту.

Задание: реализовать алгоритм увеличения изображения в целое количество раз методом билинейной интерполяции.

7. Масштабирование Ланцоша (*Lanczos*).

Этот алгоритм масштабирования основан на фильтре Ланцоша, который применяется для передискретизации сигналов.

Пусть одномерный сигнал $f(x)$, известен в точках с целыми координатами. Тогда промежуточное значение $f(x+t), 0 \le t < 1$ можно получить из значений функции в нескольких соседних точках:

$$f(x+t) \approx \sum_{i=-n+1}^{n} f(x+i)h(t-i),$$

где $h(t)$ – функция Ланцоша.

$$h(t)=\begin{cases}1, t=0\\ \sin c(t)\sin c\left(\dfrac{t}{n}\right), -n<t<n,\\ 0, t\geq n\cup t\leq -n\end{cases}$$

где $\sin c(t)=\dfrac{\sin(\pi t)}{\pi t}$.

Такой алгоритм интерполяции называют *Lanczos-n*. Как правило *n* выбирают равным 3. Интерполяция *Lanczos*3 используется в современных графических и видеоредакторах.

Не трудно показать, что при $n\to\infty$ выражение для $f(x+t)$ превращается в ряд Котельникова.

На двумерный случай фильтр обобщается следующим образом: сначала передискретизация происходит по одному измерению, затем – по другому.

Задание: реализовать алгоритм увеличения изображения в $N = 4$ раза методом интерполяции *Lanczos*3.

8. Автоматическая регулировка яркости и контрастности

Пусть m и M – соответственно минимальное и максимальное значения яркости пикселя для данного изображения. Автоматическая регулировка позволит увеличить контраст изображения. Тогда, если для записи значения яркости отводится один байт, то диапазон изменения значений яркости пикселей будет $0\leq I\leq 255$. Тогда яркость пикселей результирующего изображения будет вычислена по формуле:

$$I=\frac{(I_0-m)255}{M},$$

где I_0 – яркость пикселя исходного изображения.

Задание: реализовать алгоритм автоматической регулировки контрастности, при этом в программе для графического процессора поиск максимального и минимального значений яркости пикселя также должен выполняться на графическом процессоре.

1. Масштабирование методом «до ближайшей соседней точки».

Пусть исходное изображение имеет размеры W_0 пикселей в ширину и H_0 пикселей в высоту. Пусть новое изображение имеет размеры W_1 пикселей в ширину и H_1 пикселей в высоту. Тогда пиксель нового изображения с координатами (i, j) будет иметь то же значение яркости, что пиксель исходного изображения с координатами $\left(\left[\frac{iH_0}{H_1}\right],\left[\frac{jW_0}{W_1}\right]\right)$.

Задание: реализовать алгоритм масштабирования изображения методом «до ближайшей соседней точки». На вход алгоритма помимо исходного изображения должны поступать размеры нового изображения. Размеры нового изображения могут быть произвольными: как больше, так и меньше размеров исходного изображения.

Требования к отчёту

Отчёт по лабораторной работе должен содержать:

- исходные тексты разработанных программ;
- результаты проведённых измерений;
- меры, которые были предприняты при выполнении п. 2 задания, благодаря которым достигнуто ускорение программы на *GPU*;
- выводы по лабораторной работе.

Контрольные вопросы

1. В чем заключается работа алгоритма, указанного в варианте задания?
2. Объясните, в каких случаях программа для *GPU* будет выполняться за большее время, чем аналогичная программа для *CPU*?
3. Какие виды памяти использовались при разработке программы обработки изображений для *GPU*?

Лабораторная работа №5

ИЗУЧЕНИЕ ЯЗЫКА ПРОГРАММИРОВАНИЯ *OpenCL* НА ПРИМЕРЕ РЕШЕНИЯ ЗАДАЧ ОБРАБОТКИ ИЗОБРАЖЕНИЙ

Целью данной работы является получение навыков создания параллельных программ, используя язык программирования *OpenCL*.

Общие сведения об *OpenCL*

По мере распространения гетерогенных вычислительных систем, всё более актуальной становится задача эффективного использования всех доступных вычислительных ресурсов.

OpenCL – это открытый стандарт для параллельного программирования, предлагающий эффективный и переносимый способ использования возможностей разнородных вычислительных многоядерных платформ (*CPU*, *GPU* и другие).

Он включает:

1) *API* – программный интерфейс для координирования параллельных вычислений в среде разнородных процессоров;

2) кроссплатформенный язык *OpenCL C*, используемый в определённом вычислительном окружении.

Стандартизацией *OpenCL* занимается *Khronos Group*. Актуальная версия стандарта *OpenCL* 3.1, но в лабораторной работе будет использоваться версия 2.0 как более распространённая. Спецификации *OpenCL* для всех версий стандарта доступны по адресу: https://www.khronos.org/registry/OpenCL/.

Архитектура *OpenCL*

OpenCL определяет следующую иерархию моделей:

- модель платформы;
- модель выполнения;
- модель памяти;
- программная модель.

Модель платформы определяет архитектуру вычислительной системы. Вычислительная система состоит из хоста (*host*), соединённого с одним или более устройствами (*OpenCL devices*). Устройство *OpenCL* состоит из нескольких вычислительных устройств (*compute units*, *CU*),

которые, в свою очередь, состоят из одного или более обрабатывающих элементов (*processing elements*, *PE*). Вычисления на устройстве происходят на обрабатывающих элементах (рис. 13). Пользователю может быть доступно несколько платформ одновременно.

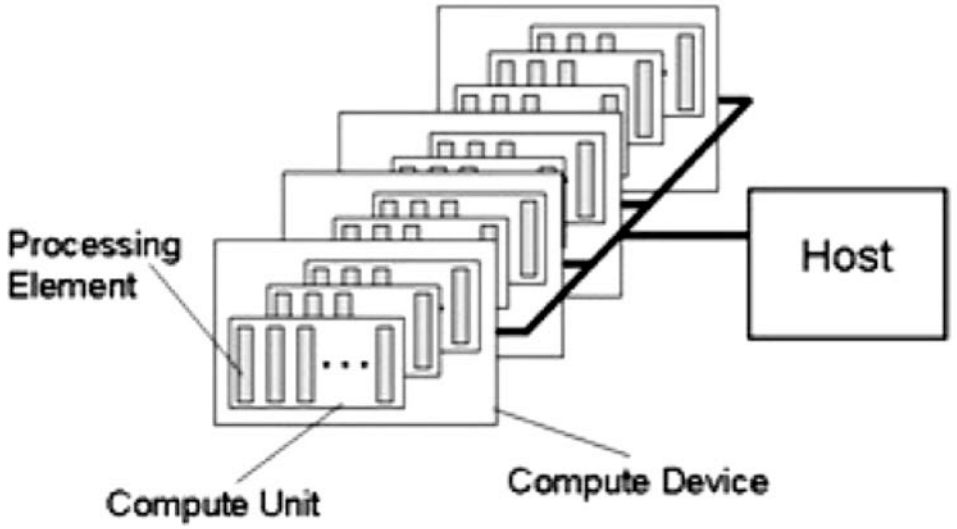


Рис. 13. Модель платформы *OpenCL*

Модель выполнения определяет следующие единицы выполнения: ядра (*kernel*), которые выполняются на одном или более устройствах *OpenCL* и программу для хоста (*host program*), которая выполняется на хосте. Объём вычислений, которые надо выполнить ядру (*"work"* в терминах *OpenCL*) состоит из элементов (*work-items*), исполняемых внутри рабочих групп (*work-groups*).

*Я*дра *OpenCL* могут быть заданы строками, содержащими исходный код на *OpenCL C*, на промежуточном языке *SPIR-V*, а также в виде бинарных объектов, определяемых используемой реализацией *OpenCL*. *Я*дро выполняется внутри контекста, управляемого хостом. Контекст включает следующие ресурсы:

– устройства;

– ядра (*krnel objects*) – функции *OpenCL* вместе с их аргументами, которые выполняются на устройствах *OpenCL*;

– программы (*program objects*) – исходные тексты и двоичные коды программ, которые реализуют ядра;

– объекты памяти (*memory objects*) – переменные, видимые хосту и устройствам. Запускаемые ядра работают именно с этими объектами.

Программа для хоста должна использовать *OpenCL API*, чтобы создать контекст и управлять им. Хост взаимодействует с устройством через очередь команд (*command queue*). С каждым устройством *OpenCL*

связана своя очередь команд. Команды, которые помещаются в очередь, могут быть трёх типов:

– вызов ядра (*kernel-enqueue commands*);

– команды доступа к памяти (*memory commands*);

– команды синхронизации (*synchronization commands*).

Ядро, запущенное на устройстве, также может помещать команды в очередь команд устройства. Таким образом, ядро может запускать дочерние ядра (*child kernel*). Каждая команда может быть в одном из 6 состояний:

1) поставлена в очередь команд (*queued*);

2) отправлена на устройство для исполнения (*submitted*);

3) готова (*ready*) – все условия, необходимые для запуска команды, выполнены;

4) выполняется (*running*);

5) выполнение завершено (*ended*);

6) выполнено (*complete*) – команда и её дочерние команды завершили выполнение.

Состояние команды можно узнать, используя объекты событий (*event objects*). События также могут быть использованы для синхронизации команд.

По умолчанию команды начинают выполнение на устройстве в том же порядке, в котором они помещаются в очередь команд. Однако можно создать очередь команд, которая поддерживает изменение порядка выполнения (*out-of-order execution*), при использовании такой очереди для каждой команды следует указывать события, которые должны завершиться, чтобы команда могла быть выполнена.

Можно создать несколько очередей команд для одного устройства. Между командами, помещёнными в разные очереди, не будет производиться синхронизация. Такой приём полезен для программирования независимых операций на устройстве.

Каждая рабочая группа ядра, а также каждый элемент имеют свои индексы, в N-мерном пространстве (*NDRange* в терминологии *OpenCL*), где $N \in \{1,2,3\}$ (рис. 14).

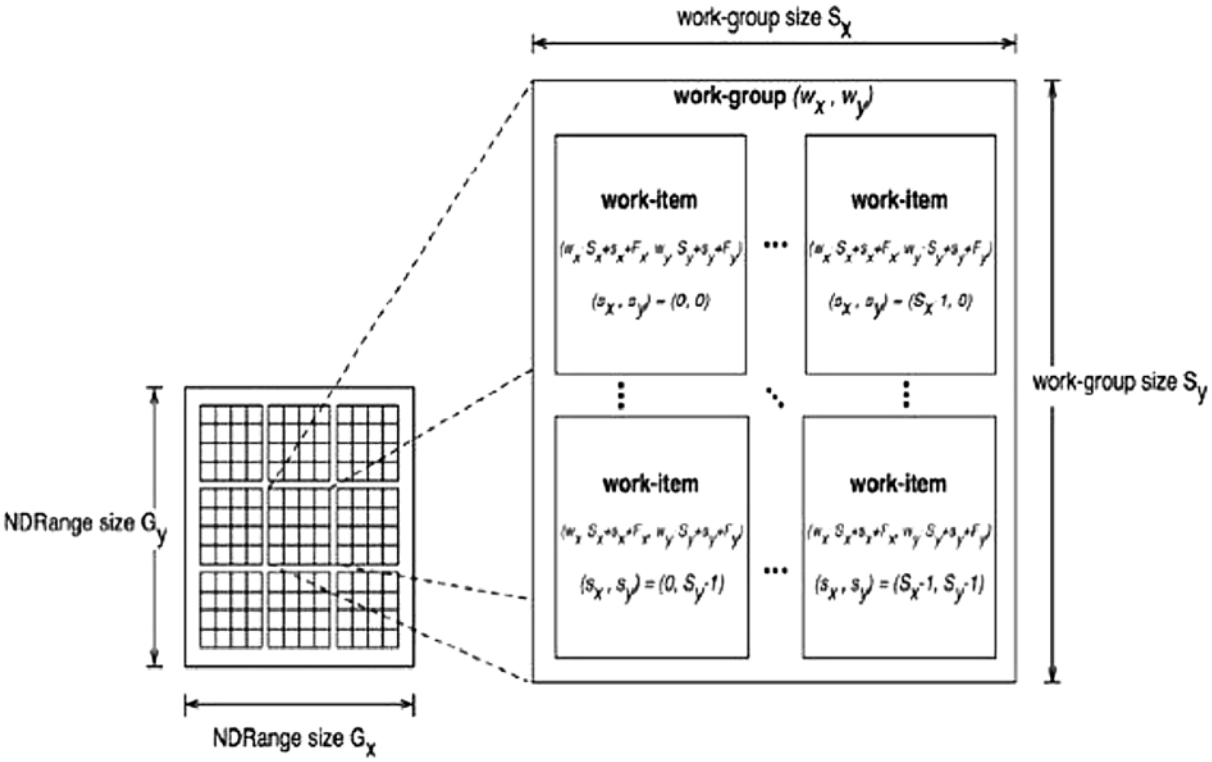


Рис. 14. Пример *N*-мерного пространства индексов, в котором выполняется ядро

Модель памяти определяет состав, структуру и поведение памяти, доступной программе *OpenCL*. Память делится на память хоста и память устройства. Память устройства делится на следующие адресные пространства (рис. 15):

– глобальная память (*global memory*) – доступна на чтение и запись всем элементам всех рабочих групп всех ядер, выполняющихся внутри контекста. Может кэшироваться, в зависимости от возможностей устройства. Доступна всем устройствам контекста;

– константная память (*constant memory*) – остаётся неизменной во время выполнения ядра. Выделение и инициализация объектов в этой памяти производится хостом. Доступна всем устройствам контекста;

– локальная память (*local memory*) – область памяти, выделенная рабочей группе. Эта область памяти используется для хранения переменных, разделяемых между всеми элементами в рабочей группе;

– приватная память (*private memory*) – область памяти, доступная только элементу.

Начиная с *OpenCL* 2.0 программисту доступно обобщённое адресное пространство, включающее в себя глобальную, локальную и приватную память.

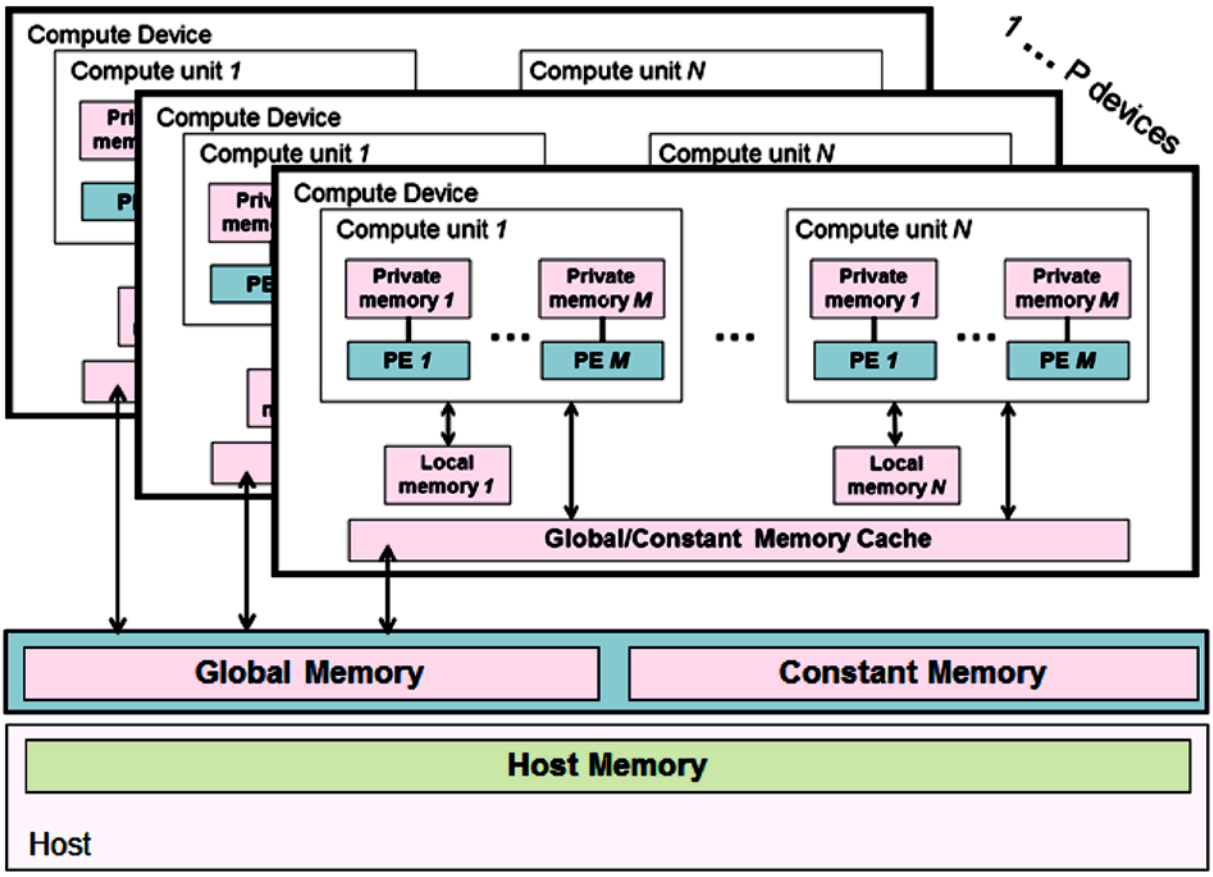


Рис. 15. Адресные пространства, доступные платформе *OpenCL*

Обзор *OpenCL API*

Приложение *OpenCL* представляет собой совокупность программного кода, выполняемого на хосте и устройствах *OpenCL*. Хостом обычно выступает *CPU*. При этом программная модель у такого приложения *SIMD* – код ядра может быть исполнен одновременно на множестве вычислительных единиц, работающих каждая со своими данными. Ресурсы и другое (память, контекст, ядро) в *OpenCL* представляются в виде объектов.

Ядро (*kernel*) – функция, выполняемая на устройствах. Помечается специальным ключевым словом `__kernel`, является «точкой входа» в исполняемый на устройствах код. Компилируется специальным компилятором по запросу в части приложения на хосте.

Команда (*command*) – операция *OpenCL*, предназначенная для исполнения (исполнение ядра на устройстве, манипуляции с памятью и команды синхронизации).

Очередь команд (*command queue*) – объект, содержащий команды для исполнения на устройстве.

Объект ядра (*kernel object*) – хранит отдельную функцию ядра программы вместе со значениями аргументов.

Объект события (*event object*) – хранит состояние команды, предназначен для синхронизации и профилирования.

Контекст (*context*) – среда, в которой выполняются ядра, а также область определения синхронизации и управления памятью. Включает:

– набор устройств;

– память, доступную устройствам;

– свойства памяти;

– одну или несколько очередей команд.

Для запуска ядра *OpenCL* нужно:

– получить информацию о доступных платформах и устройствах *OpenCL*;

– создать контекст, в котором будет выполняться ядро *OpenCL*;

– скомпилировать ядро *OpenCL*;

– создать очередь команд для заданного контекста и устройства;

– создать объекты памяти и скопировать исходные данные в память устройства;

– создать объект ядра, связать его с объектами памяти;

– поместить объект ядра в очередь команд.

Полная документация по *OpenCL API* доступна в [4].

Обзор *OpenCL C*

*Я*дра для программ *OpenCL* могут быть написаны на языке *OpenCL C* – языке программирования, основанном на языке *C* (*C*99, стандарт *ISO*/*IEC*9899:1999).

Основные отличия:

– ключевое слово `__kernel` для определения функции ядра;

– для переменных можно задать тип памяти - `__global`, `__local`, `__private`, `__constant`;

– векторные типы данных `charn`, `intn`, `floatn` и т.д., где $n \in \{2,3,4,8,16\}$;

– типы данных для текстур `image2d_t` и т.д.

Полная документация по *OpenCL C* доступна в [5]. Также необходимо ознакомиться с примером программы для *OpenCL*, приведённом в приложении к данному описанию.

Инструментальные средства

Существует несколько реализаций *OpenCL*, на одной машине может быть одновременно установлено несколько реализаций. Реализация *OpenCL*, как правило, поставляется вместе с драйверами устройства. Этого достаточно для запуска программ. Однако для разработки необходимы заголовочные файлы и библиотеки импорта, их надо установить отдельно. Достаточно установить один из предлагаемых комплектов для разработчиков, главное, чтобы были установлены драйвера от всех устройств.

Наиболее распространены реализации *OpenCL*, разработанные:

– *NVIDIA* (https://developer.nvidia.com/cuda-toolkit);

– *AMD* (https://github.com/GPUOpen-LibrariesAndSDKs/OCL-SDK/releases);

– *Intel* (https://github.com/GPUOpen-LibrariesAndSDKs/OCL-SDK/releases).

Домашняя подготовка

1. Изучить материалы данного пособия.

2. Изучить и скомпилировать тестовую программу для *OpenCL* из Приложения Б.

Задание на лабораторную работу

1. Получить сведения о всех платформах *OpenCL* и всех устройствах *OpenCL*, доступных в Вашей системе.

2. Переписать программу, разработанную в предыдущей лабораторной работе, используя язык *OpenCL C* и *OpenCL API*.

3. Для каждого доступного устройства *OpenCL* измерить времена выполнения обработки изображений разрешения *FullHD* (1920x1080). Подобрать такое количество рабочих групп и элементов, чтобы время обработки было минимальным. Измерить время обработки с учётом передачи данных между хостом и устройством, а также время выполнения ядра.

4. Вычислить коэффициент ускорения выполнения программы на *GPU* относительно *CPU* (программы, разработанной в Лабораторной работе 4), а также долю времени, которую занимает обмен данными при решении задачи на *GPU*. Результаты измерений привести в таблице.

5. Сравнить времена выполнения задачи на *OpenCL* и *CUDA*.

Контрольные вопросы

1. Что такое *OpenCL*?

2. Что конкретно *OpenCL* подразумевает под «платформой»?

3. Запуск функции-ядра блокирует ход выполнения части программы на хосте или нет?

4. В очередь команд поместили операцию копирования исходных данных для ядра, а затем запуск ядра, использующего эти данные. Необходимо ли ожидание завершения копирования на стороне устройства?

5. Необходимо ли ожидание завершения копирования результатов выполнения ядра в память хоста?

6. Какие действия необходимы для запуска ядра?

7. Что такое контекст?

8. Возможно ли использовать несколько очередей запросов в рамках одного контекста? Как в этом случае выполняются команды в различных очередях?

9. Какие типы синхронизации поддерживаются стандартом *OpenCL*?

10. В чем заключатся модель памяти в *OpenCL*?

Лабораторная работа №6

ИЗУЧЕНИЕ МНОГОПОТОЧНОГО ПРОГРАММИРОВАНИЯ НА ПРИМЕРЕ *POSIX Threads* И СТАНДАРТНОЙ БИБЛИОТЕКИ ЯЗЫКА *C++*

Целью данной работы является получение навыков создания многопоточных программ, используя библиотеки *POSIX Threads* и стандартной библиотеки языка *C++*.

Общие сведения *POSIX Threads*

POSIX Threads (*pthreads*) – стандарт реализации нитей (потоков выполнения) (*IEEE Std 1003.1c-1995*). Стандарт определяет интерфейс программирования (*API*) и модель выполнения, которая зависит от языка программирования. Получить краткий обзор информации о *pthreads* можно в системе справки *man*, вызвав команду `man pthreads`.

Согласно стандарту, процесс может состоять из нескольких нитей. Нити процесса имеют общую память данных (сегменты *data* и *heap*), но каждая нить имеет собственный стек. Кроме того, у нитей общие *pid*, *ppid*, терминал, дескрипторы открытых файлов, а также некоторые другие объекты. Каждая нить имеет собственный *threadID*, переменную *errno* и некоторые другие объекты (подробнее см. `man pthread`).

Стандарт определяет функции работы с нитями (создания, завершения, ожидания завершения), функции работы с мутексами (создания, удаления, блокировки, разблокировки), функции работы с переменными условия (*condition variables*). В данной лабораторной работе рассмотрим работу с нитями и мутексами.

Стандарт также определяет список потокобезопасных функций, т.е., функций, вызов которых приводит к тем же результатам, вне зависимости от того, из скольких нитей одновременно вызвана функция.

Реализации *pthreads* существуют для большинства *UNIX*-подобных систем, а также для *Windows*.

Краткие сведения о *POSIX Threads API*

Чтобы использовать *API POSIX Threads*, в исходный текст программы необходимо включить файл `pthread.h`. При линковке необходимо указать библиотеку *pthreads* (аргумент командной строки `-lpthread` для *gcc* или *clang*).

Нить создаётся функцией

int pthread_create(pthread_t *restrict *thread***,**
const pthread_attr_t *restrict *attr***,**
void *(* *start_routine***)(void *),**
void *restrict *arg***);**

функция имеет следующие параметры:

– *thread* – адрес переменной, в которую будет записан дескриптор потока;

– *attr* – указатель на атрибуты нити (атрибутами нити можно управлять, используя функции с именами, начинающимися с `pthread_attr_`), если *attr* имеет значение *NULL*, то нить создаётся с атрибутами по умолчанию;

– *strart_routine* – указатель на функцию, которая будет запущена в отдельной нити (функция нити), функция должна иметь следующую сигнатуру `void* f(void* arg);`

– *arg* – аргумент, который будет передан функции нити.

Нить может быть как присоединяемой (*joinable*), так и отсоединённой (*detached*). Отличия между типами нитей состоят в следующем. Завершения присоединяемой нити может ожидать другая нить, при этом присоединяемая нить освобождает свои ресурсы только после вызова этой функции. Отсоединённая нить освобождает свои ресурсы сразу после завершения. По умолчанию функция `pthread_create` создаёт присоединяемую нить, чтобы создать *detached*-нить, при создании нити надо установить соответствующий атрибут (при помощи `pthread_attr_setdetachstate`).

Ожидание завершения присоединяемой нити осуществляется при помощи вызова:

int pthread_join(pthread_t *thread***, void **** *retval***);**

функция ожидает завершения нити *thread*. Если нить уже завершена, функция возвращается немедленно. Если *retval* не *NULL*, в него копируется значение, возвращённое нитью. Нить *thread* должна быть присоединяемой.

Создание мутекса осуществляется вызовом:

int pthread_mutex_init(pthread_mutex_t *restrict *mutex***,**
const pthread_mutexattr_t *restrict *attr***);**

аргументы функции:

– *mutex* – адрес переменной, в которую будет записана структура данных мутекса, перед вызовом функции переменная должна быть инициализирована при помощи PTHREAD_MUTEX_INITIALIZER;

– *attr* – атрибуты мутекса (атрибутами можно управлять, используя функции с именами, начинающимися с pthread_mutexattr_), если attr имеет значение NULL, то муткес создаётся с атрибутами по умолчанию.

Чтобы заблокировать мутекс, надо вызвать функцию:

int pthread_mutex_lock(pthread_mutex_t **mutex*);

Если мутекс уже заблокирован, то нить будет остановлена до тех пор, пока мутекс не разблокируется.

Чтобы осуществить попытку блокировки мутекса, надо вызвать фнкцию:

int pthread_mutex_trylock(pthread_mutex_t **mutex*);

В отличие от pthread_mutex_lock, если мутекс заблокирован, эта функция сразу завершится с ошибкой.

Чтобы разблокировать мутекс, необходимо вызвать:

int pthread_mutex_unlock(pthread_mutex_t **mutex*);

Удаление мутекса осуществляется при помощи вызова:

int pthread_mutex_destroy(pthread_mutex_t **mutex*);

При удалении мутекс должен быть разблокирован, иначе функция завершится с ошибкой.

Общие сведения о поддержке многопоточности в *C++*

Начиная со стандарта *C++*11, в языке *C++* существует поддержка многопоточности. Поддерживаются следующие объекты: нити, мутексы, переменные условия, атомарные переменные, ожидание асинхронной установки значений (`std::future`) и т.д.

Стандартная библиотека *C++* кросс-платформенная, для реализации указанных объектов может быть использован *WinAPI* (на *Windows*) или *POSIX Threads* (на *Linux*).

Объект класса `std::thread` представляет нить (https://en.cppreference.com/w/cpp/thread/thread). Аргументами конструктора является функтор, т.е. объект, для которого определён `operator()`, а также аргументы для вызова этого функтора. В качестве такого аргумента подойдёт лямбда-выражение, объект `std::function` или указатель на функцию.

Для `std::thread` может быть вызвана функция `join` (аналогичная `pthread_join`). Также нить можно сделать отсоединённой, вызвав функцию `detach`.

Объект класса `std::mutex` представляет мутекс (https://en.cppreference.com/w/cpp/thread/mutex).

Чтобы осуществлять блокировку, попытку блокировки, разблокировку, можно воспользоваться, соответственно, функциями `lock` (аналогична `pthread_mutex_lock`), `unlock` (аналогична `pthread_mutex_unlock`), `try_lock` (аналогична `pthread_mutex_trylock`), однако удобнее использовать объекты `std::lock_guard` или `std::unique_lock`.

В конструкторах этих объектов происходит блокировка мутекса, а в деструкторах – разблокировка. Таким образом, отсутствует необходимость вручную писать код разблокировки мутекса.

C++11 также поддерживает атомарные переменные (`std::atomic`) (https://en.cppreference.com/w/cpp/atomic/atomic). Если одна нить производит запись в атомарную переменную, а другая читает из неё, то результат будет определён. Можно считать, что для программиста доступ к такой переменной является атомарным.

Задание на лабораторную работу

1. Написать многопоточную программу согласно варианту задания. Для управления нитями и их синхронизации использовать интерфейс *POSIX Threads*. Для синхронизации нитей использовать `pthread_mutex`, для обмена данными между нитями использовать общую память процесса:

1) разработайте программу из двух нитей. Первая нить передаёт второй нити содержимое некоторого файла поблочно. Если нить не может открыть файл, она должна передать сообщение на вывод *STDERR*. Вторая нить получает блоки файла и их выводит на экран, используя *STDOUT*;

2) разработайте программу из трёх нитей: одна поблочно читает файл, а две других выводят его на экран. Помимо информации из файла, нити должны выводить свой *id*;

3) разработайте программу из трёх нитей: каждая из двух первых нитей читает свой файл поблочно, а третья выводит на экран содержимое файлов, а также *id* нити, от которой получены эти данные;

4) решите задачу для варианта 1, при условии, что нить, выводящая блок на *STDOUT*, завершается после вывода блока, т.е., для вывода следующего блока необходимо создавать нить снова;

5) решите задачу для варианта 1, при условии, что нить, читающая блок из файла, завершается после передачи блока второй нити, т.е., для чтения следующего блока необходимо создавать нить снова;

6) разработайте программу поиска строки в тексте файла. Главная нить процесса читает файл в память, затем запускаются 4 нити, каждая из которых ищет строку в своём фрагменте файла. По завершении поиска главная нить выводит результат. Программа должна корректно искать подстроку, оказавшуюся на границе частей файла.

7) разработайте программу поиска минимального и максимального значений в файле, состоящем из 32-битных целых чисел со знаком. Главная нить процесса читает файл в память, затем запускаются 4 нити, каждая из которых ищет указанные значения в своём фрагменте файла. По завершении поиска главная нить выводит результат.

2. Переписать программу из п. 1 задания, используя `std::thread` и `std::mutex`. Допускается использование атомарных переменных.

Контрольные вопросы

1. Что такое *POSIX Threads*?
2. Что такое нить?
3. Что такое муткес?
4. Приведите пример многонитевой программы с ошибкой взаимной блокировки.
5. Приведите пример многонитевой программы с ошибкой, связанной с гонками.
6. Можно ли удалить заблокированный муткес?
7. Можно ли запустить новую нить из одной нити, а ожидать завершения в другой?
8. Почему `pthread_join` для некоторой нити нужно вызывать из другой нити?
9. Что такое атомарная переменная?
10. Как при помощи мутекса реализовать атомарную переменную?

СПИСОК ЛИТЕРАТУРЫ

1. Сандерс, Дж. Технология CUDA в примерах: Введение в программирование графических процессоров / Дж. Сандерс, Э. Кэндрот; пер. с англ. А.А. Слинкина, науч. ред. А.В. Боресков. – М.: ДМК Пресс, 2013.

2. Боресков, А.В. Основы работы с технологией CUDA / А.В. Боресков, А.А. Харламов. – М.: ДМК Пресс, 2011.

3. Казённов, А.М. Основы технологии CUDA И OpenCL / А.М. Казённов. – М., 2013 г.

4. The OpenCL Specification: Version v3.1.1 / Khronos OpenCL Working Group. – 2020. – URL: https://registry.khronos.org/OpenCL/specs/unified/pdf/OpenCL_API.pdf (дата обращения 28.06.2026)

5. The OpenCL C Specification: Version v3.1.1 / Khronos OpenCL Working Group. – 2026. – URL: https://registry.khronos.org/OpenCL/specs/unified/pdf/OpenCL_C.pdf

Приложение А

Функции записи и чтения данных из файлов формата *Targa*

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

//Сохраняет изображение в файл .tga
// path - путь к файлу
// data - указатель на данные изображения
// w - ширина изображения
// h - высота изображения
// bpp - количество бит на пиксель
// возвращает true, если удалось записать данные в файл, иначе - false
bool Array2Targa(const char *path, const unsigned char *data, unsigned w, unsigned h,
unsigned bpp){
    unsigned char TargaMagic[12] = {0,0,0,0,0,0,0,0,0,0,0,0};
    if(bpp==8)
        TargaMagic[2]=3;
    else
        TargaMagic[2]=2;
    FILE *File = fopen(path, "wb");
    if(File==NULL){
        return false;
    }
    if(fwrite(TargaMagic,1,sizeof(TargaMagic),File)!= sizeof(TargaMagic)){
        fclose(File);
        return false;
    }
    unsigned char Header[6]={0};
    Header[0]=w&0xFF; Header[1]=(w>>8)&0xFF;
    Header[2]=h&0xFF; Header[3]=(h>>8)&0xFF;
    Header[4]=bpp;
    unsigned int ImageSize=w*h*(bpp)/8;
    if(fwrite(Header,1,sizeof(Header),File)!=sizeof(Header)){
        fclose(File);
        return false;
    }
    if(fwrite(data,1,ImageSize,File)!=ImageSize){
        fclose(File);
        return false;
    }
    fclose(File);
    return true;
}

//Загружает изображение из несжатого файла .tga
// path - путь к файлу
// pdata - указатель на переменную, в которую будет записан указатель на считанные данные
//         обращаю внимание, что памать для хранения данных надо освобождать вручную
// pw - указатель на переменную, в которую будет записана ширина изображения
// ph - указатель на переменную, в которую будет записана высота изображения
// pbpp - указатель на переменную, в которую будет записано количество бит на пиксель
// возвращает true, если удалось считать данные из файла, иначе - false
bool Targa2Array(const char *path, unsigned char **pdata, unsigned *pw, unsigned *ph,
unsigned *pbpp){
    const unsigned char TargaMagic[12] = {0,0,0,0,0,0,0,0,0,0,0,0};
    unsigned char FileMagic[12];
    unsigned char Header[6];
    //char *data;
    FILE *File = fopen(path, "rb");
    if(File==NULL){
        return false;
    }
    if(fread(FileMagic,1,sizeof(FileMagic),File)!= sizeof(FileMagic)){
        fclose(File);
        return false;
    }
    unsigned char ImageType=FileMagic[2];
    FileMagic[2]=0;
    if(memcmp(TargaMagic,FileMagic,sizeof(TargaMagic))!= 0||
       fread(Header,1,sizeof(Header),File)!= sizeof(Header)){
```

```
        fclose(File);
        return false;
    }
    // Определяем размеры изображения
    *pw=Header[1]*256+Header[0];
    *ph=Header[3]*256+Header[2];
    // Определяем глубину цвета используемую в изображении
    *pbpp=Header[4];
    unsigned int Bpp=*pbpp/8;
    //Поддерживаются только изображения с 1, 3 или 4 байта на пиксель
    if(*pw<=0 || *ph<=0 || (ImageType==2&&Bpp!=3&&Bpp!=4) || (ImageType==3&&Bpp!=1)){
        fclose(File);
        return false;
    }
    unsigned int ImageSize=*pw**ph*Bpp;
    unsigned char *data=(unsigned char *)malloc(ImageSize);
    // Читаем данные
    if(data==NULL||fread(data,1,ImageSize,File)!=ImageSize){
        free(data);
        fclose(File);
        return false;
    }
    // Закрываем файл
    fclose (File);
    *pdata=data;
    return true;
}
```

Пример использования функций записи и чтения данных из файлов формата *Targa*

```
int main(int argc, char **argv){
    if(argc!=3)
        return 1;
    unsigned char *data;
    unsigned w=0, h=0, bpp=0;
    //загрузка данных из файла, заданного в первом аргументе командной строки
    if(!Targa2Array(argv[1], &data, &w, &h, &bpp))
        return 1;

    //здесь производится обработка данных

    //запись данных в файл, заданного в первом аргументе командной строки
    Array2Image(argv[2], data2, w, h, bpp);
    //data надо освобождать вручную
    free(data);
    return 0;
}
```

Приложение Б

Пример программы на *OpenCL*

```
#include <CL/cl.h>

#include <string>
#include <vector>
#include <iostream>
#include <cstdlib>

// кросс-платформенный вариант изменения времени
// OpenCL предоставляет функции для получения таймеров, начиная с версии 2.1, поэтому
воспользуемся этой реализацией
#ifdef _WIN32
#include <windows.h>
#else
#include <stdlib.h>
#include <sys/time.h>
long long GetTickCount64()
{
    struct timeval ts;
    gettimeofday(&ts, NULL);
    return ts.tv_sec * 1000 + ts.tv_usec / 1000;
}
#endif

int main(int argc, char **argv){
    // строка, содержащая исходный код программы на OpenCL
    const std::string source =
 R"(__kernel void sumKernel(__global float *a, __global float *b, __global float *res,
unsigned size){
    int workItemsCount = get_global_size(0); // количество элементов в NDRange
(соответствует количеству нитей grid для CUDA)
    int globalWorkItemId = get_global_id(0); // глобальный идентификатор элемента

    unsigned startIdx = (size * globalWorkItemId) / workItemsCount;
    unsigned endIdx = (size * (globalWorkItemId + 1)) / workItemsCount;

    for(unsigned i = startIdx; i < endIdx; ++i)
        res[i] = a[i] + b[i];
})";

    constexpr cl_uint vectorSize = 1024*1024; // 1 MB

    // массивы исходных данных
    std::vector<float> a(vectorSize, 1.0f);
    std::vector<float> b(vectorSize, 2.0f);
    // массив результата
    std::vector<float> res(vectorSize);

    // проверка аргументов командной строки
    if(argc!=4){
        std::cout << "Usage: " << argv[0] << "platform_index work_group_size
work_groups_count" << std::endl;
        return 1;
    }
    const int workgroupSize = std::atoi(argv[2]);
    if(workgroupSize <= 0){
        std::cerr << "Invalid work_group_size" << std::endl;
        return 1;
    }
    const int workgroupsCount = std::atoi(argv[3]);
    if(workgroupsCount <= 0){
        std::cerr << "Invalid work_groups_count" << std::endl;
        return 1;
    }

    // получение информации о платформах
    cl_uint platformCount; // количетсво платформ в системе
    if(clGetPlatformIDs(0,NULL, &platformCount) != CL_SUCCESS){ // получение значения
количества платформ в системе
```

```
        std::cerr << "Cannot get platforms!" << std::endl;
        return 1;
    }
    if(platformCount == 0){
        std::cerr << "No platforms found!" << std::endl;
        return 1;
    }
    std::vector<cl_platform_id> platforms(platformCount); // массив объектов платформ
    if(clGetPlatformIDs(platformCount, platforms.data(), NULL) != CL_SUCCESS){ // заполнение
массива platform объектами типа "платформа"
        std::cerr << "Cannot get platforms!" << std::endl;
        return 1;
    }
    std::cout << "Found " << platforms.size() << " platforms" << std::endl;

    const int platformIndex = std::atoi(argv[1]);
    if(platformIndex >= platforms.size() || platformIndex < 0){
        std::cerr << "Invalid platform index" << std::endl;
        return 1;
    }
    const cl_platform_id platformId = platforms[platformIndex];

    // получение информации об устройствах GPU для заданной платформы
    cl_uint deviceCount = 0;
    if(clGetDeviceIDs(platformId, CL_DEVICE_TYPE_GPU, 0, NULL, &deviceCount) != CL_SUCCESS){
        std::cerr << "Cannot get devices!" << std::endl;
        return 1;
    }
    if(deviceCount == 0){
        std::cerr << "No devices found!" << std::endl;
        return 1;
    }
    std::vector<cl_device_id> devices(deviceCount);
    if(clGetDeviceIDs(platformId, CL_DEVICE_TYPE_GPU, deviceCount, devices.data(), NULL) !=
CL_SUCCESS){
        std::cerr << "Cannot get devices!" << std::endl;
        return 1;
    }
    std::cout << "Found " << devices.size() << " devices" << std::endl;

    // создание контекста
    cl_int ret = 0; // переменная для хранения кода ошибки
    cl_context context = clCreateContext(NULL, 1, devices.data(), NULL, NULL, &ret); //
создаём контекст для всех GPU-устройств платформы
    if(ret != 0){
        std::cerr << "Cannot create context!" << std::endl;
        return 1;
    }

    const auto startAll = GetTickCount64();

    // создание программы из исходного текста
    const char *sourcePtr = source.data();
    size_t sourceSize = source.size();
    cl_program program = clCreateProgramWithSource(context, 1, &sourcePtr, &sourceSize,
&ret); // создание объекта - программы
    if(clBuildProgram(program, 1, devices.data(), NULL, NULL, NULL) != CL_SUCCESS){
        // вывод отчёта об ошибках компиляции
        std::cerr << "clBuildProgram for deivce id=" << devices[0] << " failed, error: " <<
ret << std::endl;
        size_t buildLogLen = 0;
        if(clGetProgramBuildInfo(program, devices[0], CL_PROGRAM_BUILD_LOG, 0, NULL,
&buildLogLen) == CL_SUCCESS){
            std::vector<char> buildLog(buildLogLen);
            if(clGetProgramBuildInfo(program, devices[0], CL_PROGRAM_BUILD_LOG, buildLogLen,
buildLog.data(), NULL) == CL_SUCCESS){
                std::cerr << "Build log" << std::endl;
                std::cerr << "----------------------------------------" << std::endl;
                std::cerr << buildLog.data() << std::endl;
                std::cerr << "----------------------------------------" << std::endl;
            }
        }
        clReleaseContext(context); // освобождение контекста
        return 1;
    }
```

```
    // создание объекта очереди команд для заданного контекста и устройства (платформы, вида
устройств)
    // функция clCreateCommandQueue объявлена устаревшей, начиная с OpenCL 2.0, использовать
эту функцию для версий OpenCL, предшествующих 2.0
    // cl_command_queue commandQueue = clCreateCommandQueue(context, devices[0],
CL_QUEUE_PROFILING_ENABLE, &ret);
    const std::vector<cl_queue_properties> props = { CL_QUEUE_PROPERTIES,
CL_QUEUE_PROFILING_ENABLE, 0 }; // включение этой опции необходимо для измерения времени
выполнения операций
    // если изменение времени не нужно, вместо props.data() можно передать NULL
    cl_command_queue commandQueue = clCreateCommandQueueWithProperties(context, devices[0],
props.data(), &ret);

    if(ret != 0){
        std::cerr << "Cannot create command queue!" << std::endl;
        clReleaseProgram(program); // удаление объекта - программы
        clReleaseContext(context); // удаление контекста
        return 1;
    }

    // создание объекта функции-ядра
    cl_kernel kernel = clCreateKernel(program, "sumKernel", &ret);
        if(ret != 0){
        std::cerr << "Cannot create kernel!" << std::endl;
        clReleaseCommandQueue(commandQueue); // удаление очереди команд
        clReleaseProgram(program); // удаление объекта - программы
        clReleaseContext(context); // удаление контекста
        return 1;
    }

    // выделение буферов на устройстве под исходные данные и результат
    cl_mem buf_a = clCreateBuffer(context, CL_MEM_READ_WRITE, vectorSize * sizeof(float),
nullptr, &ret);
    cl_mem buf_b = clCreateBuffer(context, CL_MEM_READ_WRITE, vectorSize * sizeof(float),
nullptr, &ret);
    cl_mem buf_res = clCreateBuffer(context, CL_MEM_READ_WRITE, vectorSize * sizeof(float),
nullptr, &ret);

    // события, в данном случае они нужно только для замеров времени
    cl_event startMemEvent; // событие, с которым связана операция копирования (начало
копирования входных данных)
    cl_event kernelEvent; // событие, соответствующее выполнению ядра
    cl_event endMemEvent; // событие, с которым связана операция копирования результата

    // постановка в очередь операций копирования
    clEnqueueWriteBuffer(commandQueue, buf_a, CL_TRUE, 0, vectorSize * sizeof(float),
a.data(), 0, NULL, &startMemEvent);
    clEnqueueWriteBuffer(commandQueue, buf_b, CL_TRUE, 0, vectorSize * sizeof(float),
b.data(), 0, NULL, NULL);

    // установка аргументов ядра
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &buf_a); // связываем 0-й аргумент с объектом
памяти buf_a
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &buf_b); // связываем 1-й аргумент с объектом
памяти buf_b
    clSetKernelArg(kernel, 2, sizeof(cl_mem), &buf_res); // связываем 2-й аргумент с
объектом памяти buf_res
    clSetKernelArg(kernel, 3, sizeof(cl_uint), &vectorSize); // связываем 2-й аргумент с
размером массивов

    // запуск ядра
    cl_uint workDim = 1; // размерность пространства-множества исполняющих устройств
    size_t globalWorkSize = workgroupSize * workgroupsCount;
    size_t localWorkSize = workgroupSize;
    // постановка ядра в очередь
    clEnqueueNDRangeKernel(commandQueue, kernel, workDim, NULL, &globalWorkSize,
&localWorkSize, 0, NULL, &kernelEvent);

    // постановка в очередь операции копирования результатов на хост
    clEnqueueReadBuffer(commandQueue, buf_res, CL_TRUE, 0, vectorSize * sizeof(float),
res.data(), 0, NULL, &endMemEvent);

    // ожидание завершения выполнения всех операций в командной очереди
    clFinish(commandQueue);

    const auto endAll = GetTickCount64();
```

```
    std::cout << "Processing time, including compilation: " << endAll - startAll << " ms" <<
std::endl;

    // измерение времён работы
    cl_ulong enqueueTime;
    clGetEventProfilingInfo(startMemEvent, CL_PROFILING_COMMAND_QUEUED, sizeof(cl_ulong),
&enqueueTime, NULL); // время постановки в очередь команды на запись
    cl_ulong transferStartTime;
    clGetEventProfilingInfo(startMemEvent, CL_PROFILING_COMMAND_START, sizeof(cl_ulong),
&transferStartTime, NULL); // время начала передачи исходных данных
    cl_ulong kernelStartTime;
    clGetEventProfilingInfo(kernelEvent, CL_PROFILING_COMMAND_START, sizeof(cl_ulong),
&kernelStartTime, NULL); // время начала выполнения ядра
    cl_ulong kernelEndTime;
    clGetEventProfilingInfo(kernelEvent, CL_PROFILING_COMMAND_END, sizeof(cl_ulong),
&kernelEndTime, NULL); // время окончания выполнения ядра
    cl_ulong transferEndTime;
    clGetEventProfilingInfo(endMemEvent, CL_PROFILING_COMMAND_END, sizeof(cl_ulong),
&transferEndTime, NULL); // время начала передачи исходных данных

    std::cout << "Time in queue: " << transferStartTime - enqueueTime << " ns" << std::endl;
// счётчики в наносекундах
    std::cout << "Processing time: " << transferEndTime - transferStartTime << " ns " <<
std::endl;
    std::cout << "Kernel execution time: " << kernelEndTime - kernelStartTime << " ns " <<
std::endl;

    // после завершения clFinish можно использовать res - массив, в который сохранены данные

    // удаление объектов-событий
    clReleaseEvent(endMemEvent);
    clReleaseEvent(kernelEvent);
    clReleaseEvent(startMemEvent);

    // освобождение буферов на устройстве
    clReleaseMemObject(buf_res);
    clReleaseMemObject(buf_b);
    clReleaseMemObject(buf_a);

    clReleaseKernel(kernel); // удаление объекта функции-ядра
    clReleaseCommandQueue(commandQueue); // удаление очереди команд
    clReleaseProgram(program); // удаление объекта - программы
    clReleaseContext(context); // удаление контекста
    return 0;
}
```

Приложение В

Пример многопоточной программы, использующей *POSIX Threads*

```
/* Демонстрация многопоточной программы, использующей POSIX Threads*/

/* главная нить программы */
/* генерирует строки, состоящие из 256 одинаковых символов и записывает их в массив buf */
/* завершает работу после того, как флаг finished устанавливается в 1 */

/* нить printingThread */
/* в течение 3 с считывает из buf строку и выводит её в stdout; строки разделяются символом
перевода строки */
/* по окончании 3 с устанавливает флаг finished */

/* в этом варианте программы синхронизация доступа к buf и finished производится при помощи
мутекса */

#include <errno.h>
#include <signal.h>
#include <pthread.h>
#include <sys/time.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define BUF_SIZE 256

/* мутекс для синхронизации доступа к памяти */
static pthread_mutex_t memGuard;
/* переменная, которая устанавливается в 1, когда обработка звавершена, доступ к ней должен
быть синхронизирован мутексом memGuard */
static int finished = 0;

/* эта функция будет исполняться в отдельной нити */
/* в течение 3 с считывает из buf строку и выводит её на экран; строки разделяются символом
перевода строки */
void* threadFunc(void* area){
    char* buf = (char*)area;
    struct timeval tStart;
    gettimeofday(&tStart, NULL);

    for(;;){
        /* цикл будет повторяться 3 с */
        struct timeval tEnd;
        gettimeofday(&tEnd, NULL);
        if((tEnd.tv_sec - tStart.tv_sec)*1000000 +(tEnd.tv_usec - tStart.tv_usec) >=
3000000)
            break;

        /* блокируем мутекс */
        pthread_mutex_lock(&memGuard);

        ssize_t count = 0;
        while(count < BUF_SIZE){
            /* вывод данных, write обёрнута в цикл, т.к. может записать меньше байт, чем
запросил пользователь */
            ssize_t res = write(1, buf+count, BUF_SIZE-count);
            if(res < 0){ /* обработка ошибок */
                if(errno == EINTR)
                    continue;
                int err = errno;
                fprintf(stderr, "Write error: %d, %s\n", err, strerror(err));
                break;
            }
            count += res;
        }

        /* разблокируем мутекс */
        pthread_mutex_unlock(&memGuard);
```

```
        write(1, "\n", 1); /* перевод строки, чтобы отделять записи */
    }

    /* устанавливаем флаг завершения в 1, доступ синхронизирован мутексом */
    pthread_mutex_lock(&memGuard);
    finished = 1;
    pthread_mutex_unlock(&memGuard);

    return NULL;
}

int main(){
    /* выделяем буфер для массива, доступ к нему должен быть синхронизирован мутексом
memGuard */
    char* buf = (char*)malloc(BUF_SIZE);
    /* заполняем память нулями, в принципе, это не обязательно, но для пример без
синхронизации может вывести текст до того момента, как в него будут записаны данные, таким
образом, в терминале могут появиться непечантые символы. исключим эту ситуацию */
    memset(buf, '0', BUF_SIZE);

    /* инициализируем мутекс */
    pthread_mutex_init(&memGuard, NULL);

    /* создаём новую нить */
    pthread_t printingThread;
    pthread_create(&printingThread, NULL, &threadFunc, buf);


    /* генерирует строки, состоящие из 256 одинаковых символов и записывает их в buf
       buf доступен нити printingThread, т.к. нити одного процесса имеют общую память
       если чтение не синхронизировать, то на экране будут выводиться строки, содержащие
разные символы */
    char table[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
    int index = 0;
    for(;;){
        /* блокируем мутекс */
        pthread_mutex_lock(&memGuard);

        if(finished){
            /* если установлен флаг завершения, разблокируем мутекс и выходим из цикла */
            pthread_mutex_unlock(&memGuard);
            break;
        }
        /* заполняем содержимое памяти */
        memset(buf, table[index], BUF_SIZE);

        /* разблокируем мутекс */
        pthread_mutex_unlock(&memGuard);

        ++index;
        if(index == sizeof(table)-1)
            index = 0;
    }

    /* ожидаем завершения нити */
    pthread_join(printingThread, NULL);

    /* удаляем мутекс */
    pthread_mutex_destroy(&memGuard);

    /* освобождаем память */
    free(buf);

    return 0;
}
```

Приложение Г

Пример многопоточной программы, использующей стандартную библиотеку *C*++11

```
// Демонстрация многопоточной программы, использующей стандартную библиотеку C++
// для работы с нитями используется std::thread, для синхронизации используется std::mutex

// главная нить программы
// генерирует строки, состоящие из 256 одинаковых символов и записывает их в массив buf
// завершает работу после того, как флаг finished устанавливается в 1

// нить printingThread
// в течение 3 с считывает из массива buf  строку и выводит её в stdout; строки разделяются
символом перевода строки
// по окончании 3 с устанавливает флаг finished

// в этом варианте программы для синхронизации доступа к buf и finished используется мутекс

#include <thread>
#include <mutex>
#include <iostream>
#include <vector>
#include <string>
#include <memory>
#include <chrono>

static constexpr int bufSize = 256;

// эта функция будет исполняться в отдельной нити
// в течение 3 с считывает из buf строку и выводит её на экран; строки разделяются символом
перевода строки
void threadFunc(std::vector<char>& buf, std::mutex& memGuard, bool& finished){
    const auto s = std::chrono::steady_clock::now();

    for(;;){
        // цикл будет повторяться 3 с
        const auto e = std::chrono::steady_clock::now();
        if(std::chrono::duration_cast<std::chrono::seconds>(e-s) >= std::chrono::seconds(3))
            break;

        // блокируем мутекс
        std::unique_lock<std::mutex> lock(memGuard);
        // вывод данных
        std::cout.write(buf.data(), buf.size());

        // разблокируем мутекс
        lock.unlock();

        std::cout << std::endl; // перевод строки, чтобы отделять записи
    }

    // устанавливаем флаг завершения в 1, доступ синхронизирован мутексом
    std::unique_lock<std::mutex> lock(memGuard);
    finished = 1;
    lock.unlock();
}

int main(){
    // выделяем буфер для массива, доступ к нему должен быть синхронизирован мутексом
memGuard
    std::vector<char> buf(bufSize);
    // заполняем память нулями, в принципе, это не обязательно, но для пример без синхронизации
может вывести текст до того момента, как в него будут записаны данные, таким образом, в тер-
минале могут появиться непечантые символы. исключим эту ситуацию
    std::fill(buf.begin(), buf.end(), '0');

    // переменная, которая устанавливается в 1, когда обработка звавершена, доступ к ней
должен быть синхронизирован мутексом memGuard
    bool finished = false;
```

```
    // создаём мутекс
    std::mutex memGuard;

    // создаём новую нить
    std::thread printingThread([&buf, &finished, &memGuard]{threadFunc(buf, memGuard,
finished);});

    // генерирует строки, состоящие из 256 одинаковых символов и записывает их в buf
    // buf доступен нити printingThread, т.к. нити одного процесса имеют общую память
    // если чтение не синхронизировать, то на экране будут выводиться строки, содержащие
разные символы
    char table[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
    int index = 0;
    for(;;){
        {
            // блокируем мутекс
            std::unique_lock<std::mutex> lock(memGuard);
            // деструктор объекта будет вызван при выходе из блока операторов (scope),
поэтому явная разблокировка мутекса не требуется

            if(finished){
                // если установлен флаг завершения, выходим из цикла
                break;
            }
            // заполняем содержимое памяти
            std::fill(buf.begin(), buf.end(), table[index]);
        }

        ++index;
        if(index == sizeof(table)-1)
            index = 0;
    }

    // ожидаем завершения нити
    printingThread.join();

    return 0;
}
```

Приложение Д

Пример многопоточной программы, использующей стандартную библиотеку *C*++11 и атомарные переменные

```
// Демонстрация многопоточной программы, использующей стандартную библиотеку C++
// для работы с нитями используется std::thread, для синхронизации используется std::mutex
// также используются атомарные переменные

// главная нить программы
// генерирует строки, состоящие из 256 одинаковых символов и записывает их в массив buf
// завершает работу после того, как флаг finished устанавливается в 1

// нить printingThread
// в течение 3 с считывает из массива buf  строку и выводит её в stdout;
// строки разделяются символом перевода строки
// по окончании 3 с устанавливает флаг finished

// в этом варианте программы для синхронизации доступа к buf, используется мутекс
// переменная finished является атомарной

#include <thread>
#include <mutex>
#include <atomic>
#include <iostream>
#include <vector>
#include <string>
#include <memory>
#include <chrono>

static constexpr int bufSize = 256;

// эта функция будет исполняться в отдельной нити
// в течение 3 с считывает из buf строку и выводит её на экран; строки разделяются символом
перевода строки
void threadFunc(std::vector<char>& buf, std::mutex& memGuard, std::atomic<bool>& finished){
    const auto s = std::chrono::steady_clock::now();

    for(;;){
        // цикл будет повторяться 3 с
        const auto e = std::chrono::steady_clock::now();
        if(std::chrono::duration_cast<std::chrono::seconds>(e-s) >= std::chrono::seconds(3))
            break;

        // блокируем мутекс
        std::unique_lock<std::mutex> lock(memGuard);
        // вывод данных
        std::cout.write(buf.data(), buf.size());

        // разблокируем мутекс
        lock.unlock();

        std::cout << std::endl; // перевод строки, чтобы отделять записи
    }

    // устанавливаем флаг завершения в 1
    finished = 1;
}

int main(){
    // выделяем буфер для массива, доступ к нему должен быть синхронизирован мутексом
memGuard
    std::vector<char> buf(bufSize);
    // заполняем память нулями, в принципе, это не обязательно, но для пример без синхронизации
может вывести текст до того момента, как в него будут записаны данные, таким образом, в тер-
минале могут появиться непечантые символы. исключим эту ситуацию
    std::fill(buf.begin(), buf.end(), '0');

    // переменная, которая устанавливается в true, когда обработка звавершена, доступ к
переменной считается атомарной операцией, синхронизация при помощи мутекса не требуется
    std::atomic<bool> finished{false};
```

```
    // создаём мутекс
    std::mutex memGuard;

    // создаём новую нить
    std::thread printingThread([&buf, &finished, &memGuard]{threadFunc(buf, memGuard,
finished);});

    // генерирует строки, состоящие из 256 одинаковых символов и записывает их в buf
    // buf доступен нити printingThread, т.к. нити одного процесса имеют общую память
    // если чтение не синхронизировать, то на экране будут выводиться строки, содержащие
разные символы
    char table[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
    int index = 0;
    while(!finished){
        {
            // блокируем мутекс
            std::unique_lock<std::mutex> lock(memGuard);
            // деструктор объекта будет вызван при выходе из блока операторов (scope),
поэтому явная разблокировка мутекса не требуется

            // заполняем содержимое памяти
            std::fill(buf.begin(), buf.end(), table[index]);
        }

        ++index;
        if(index == sizeof(table)-1)
            index = 0;
    }

    // ожидаем завершения нити
    printingThread.join();

    return 0;
}
```

Производственно-практическое электронное издание

Филатов Александр Викторович
Орлов Дмитрий Александрович

Редактор Е.Б. Бурдюкова
Компьютерная верстка О.А. Копыловой

Для чтения электронного методического пособия требуется устройство (типа: настольный персональный компьютер PC (домашний, офисный); ноутбук; планшет (Android или MacOS); смартфон) с установленной операционной системой (типа: MS Windows; Linux; Android; MacOS) и программой позволяющей просматривать (читать) файлы формата PDF.

Для выполнения лабораторных работ используются персональные компьютеры или ноутбуки с установленными операционными системами MS Windows и Linux, и имеющими в своём составе графический видеоадаптер (или ускоритель) поддерживающий выполнение программ созданных по технологиям *NVIDEA* CUDA и OpenCL. Также требуются необходимые компиляторы языков C и C++, и программное обеспечение для использования *NVIDEA* CUDA и OpenCL.

Для индивидуального выполнения лабораторных работ необходимое программное обеспечение или опционально должно быть на устройстве (настольном компьютере или ноутбуке) обучающегося, или скачивается и устанавливается по ссылкам, указанным преподавателем. Для проведения занятий в рамках учебных лабораторий вуза, оборудование и ПО предоставляется вузом.

Дата подписания – 03.08.2026
Объём издания – 1,00 Мбайт
Тираж – 10 электронных оптических дисков DVD-R

Издательство МЭИ
111250, Москва, Красноказарменная, д. 14, стр.1
izdatmpei@gmail.com