Лабораторная работа №1
Начальные сведения о среде программирования NVIDIA CUDA

За счет того, что программы в CUDA пишутся фактически на обычном языке C (на самом деле для частей, выполняющихся на CPU, можно использовать язык C++), в который добавлено небольшое число новых конструкций (спецификаторы типа, встроенные переменные и типы, директива запуска ядра), написание программ с использованием технологии CUDA оказывается заметно проще, чем при использовании традиционного GPGPU (то есть использующего графические API для доступа GPU). Кроме того, в распоряжении программиста оказывается гораздо больше возможностей по работе с GPU.

Расширения языка C

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

Вводимые в CUDA расширения языка C состоят из:

- спецификаторов функций, показывающих, где будет выполняться функция и откуда она может быть вызвана;
- спецификаторов переменных, задающих тип памяти, используемый для данных переменных;
- директивы, служащей для запуска ядра, задающей как данные, так и иерархию нитей;
- встроенных переменных, содержащих информацию о текущей нити;
- runtime, включающей в себя дополнительные типы данных.

Спецификаторы функций и переменных

В CUDA используется следующие спецификаторы функций (табл. 1.1).

Таблица 1. 1. Спецификаторы функций в CUDA

Спецификатор	Функция выполняется на:	Функция может вызываться из:
__device__	device (GPU)	device (GPU)
__global__	device (GPU)	host (CPU)
__host__	host (CPU)	host (CPU)

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

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

На функции, выполняемые на GPU (device и global), накладываются следующие ограничения:

- нельзя брать их адрес (за исключением global функций);

- не поддерживается рекурсия;
- не поддерживаются static-переменные внутри функции;
- не поддерживается переменное число входных аргументов.

Для задания размещения в памяти GPU переменных используются следующие спецификаторы

__device__, __constant__ и __shared__. На их использование также накладывается ряд ограничений:

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

Добавленные типы

В язык добавлены 1/2/3/4-мерные векторы из базовых типов (char, unsigned, char, short, unsigned short, int, unsigned int, long, unsigned long, longlong, float и double) – char1, char2, char3,char4, uchar1, uchar2, uchar3, uchar4, short1, short2, short3, short4, ushort1, ushort2, ushort3, ushort4, int1, int2, int3, int4, uint1, uint2, uint3, uint4, long1, long2, long3, long4, ulong1, ulong2, ulong3, ulong4, float1, float2, float3, float4, longlong1, longlong2, double1 и double2.

Обращение к компонентам вектора идет по именам – x, y, z и w. Для создания значений векторов заданного типа служит конструкция вида make_<typeName>.

int2 a = make_int2 (1,7); // Создает вектор (1,7).
float3 u = make_float3 (1, 2, 3.4f); // Создает вектор (1.0f, 2.0f, 3.4f).

Для этих типов не поддерживаются векторные покомпонентные операции, то есть нельзя просто сложить два вектора при помощи оператора "+" – это необходимо явно делать для каждой компоненты.

Также добавлен тип dim3, используемый для задания размерности. Этот тип основан на типе uint3, но обладает при этом нормальным конструктором, инициализирующим все незаданные компоненты единицами.

dim3 blocks (16, 16); / / Эквивалентно blocks (16, 16, 1).
dim3 grid (256); / / Эквивалентно grid (256, 1, 1).

Добавленные переменные

В язык добавлены следующие специальные переменные:

- gridDim – размер сетки (имеет тип dim3);
- blockDim – размер блока (имеет тип dim3);
- blockldx – индекс текущего блока в сетке (имеет тип uint3);
- threadldx – индекс текущей нити в блоке (имеет тип uint3);
- warpSize – размер warp'a (имеет тип int).

Директива вызова ядра

Для запуска ядра на GPU используется следующая конструкция:

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

Здесь *kernelName* это имя (адрес) соответствующей __global__ функции.

Через *Dg* обозначена переменная (или значение) типа dim3, задающая размерность и размер сетки (в блоках).

Переменная (или значение) *Db* типа dim3, задает размерность и размер блока (в нитях). Необязательная переменная (или значение) *Ns* типа size_t задает дополнительный объем разделяемой памяти в байтах, которая должна быть динамически выделена каждому блоку (к уже статически выделенной разделяемой памяти), если не задано, то используется значение 0. Переменная (или значение) *S* типа cudaStream_t задает поток (CUDA stream), в котором должен произойти вызов, по умолчанию используется поток 0.

Через args обозначены аргументы вызова функции *kernelName* (их может быть несколько).

Следующий пример запускает ядро с именем *myKernel* параллельно на *n* нитях, используя одномерный массив из двумерных (16x16) блоков нитей, и передает на вход ядру два параметра *a* и *n*. При этом каждому блоку дополнительно выделяется 512 байт разделяемой памяти, и запуск, производится на потоке *myStream*.

myKernel <<<dim3(*n*/256), dim3(l6,l6), 512, *myStream*>>> (*a*, *n*);

Добавленные функции

CUDA поддерживает все математические функции из стандартной библиотеки языка C. При этом большинство стандартных математических функций используют числа с двойной точностью (double). Однако поскольку для современных GPU операции с double числами выполняются медленнее, чем операции с f1oat-числами, то предпочтительнее там, где это возможно, использовать f1oat-аналоги стандартных функций. Так, f1oat-аналогом функции sin() является функция sinf().

Кроме того, CUDA предоставляет также специальный набор функций пониженной точности, но обеспечивающих еще большее быстродействие. Таким аналогом для функции вычисления синуса является функция sinf().

В табл.1.2 приведены основные f1oat-функции и их оптимизированные версии пониженной точности.

Для ряда функций можно задать требуемый способ округления. Используемый способ задается при помощи одного из следующих суффиксов:

- rn – округление к ближайшему;
- rz – округление к нулю;
- ru – округление вверх;
- rd – округление вниз.

Таблица 1.2. Математические float-функции в CUDA

Функция	Значение
__fadd_[rn, rz, ru,rd] (x, y)	Сложение, никогда не переводимое в команду FMAD
__fmul_[rn, rz, ru,rd] (x, y)	Умножение, никогда не переводимое в команду FMAD
__fmaf_[rn, rz, rU,rd] (x, y, z)	(x x y)+z
__frcp_[rn, rz, ru,rd] (x)	1/x

__fsqrt_[rn, rz, ru,rd] (x)	$\sqrt{x}$
__fdiv_[rn, rz, ru,rd] (x, y)	x/y
__fdividef (x, y)	x/y, но если $2^{126} < y < 2^{128}$, то 0
__expf (x)	e^x
__exp10f (x)	10^x
__logf (x)	log(x)
__log2f (x)	$\log_2(x)$
__log10f (x)	$\log_{10}(x)$
__sinf (x)	sin(x)
__cosf (x)	cos(x)
__sincosf (x, sptr, cptr)	*sptr ==sin(x); *cptr ==cos(x)
__tanf (x)	tan(x)
__fmul[rn, rz, ru,rd] (x, y)	Умножение, никогда не переводимое в команду FMAD
__powf (x, y)	x^y
__int_as_float (x)	32 бита, образующие целочисленное значение, интерпретируются как floatзначение. Так, значение 0xC000000 будет переведено в 2.0f
__float_as_int (x)	32 бита, образующие float-значение, интерпретируются как целочисленное значение. Так, значение 1.0f будет переведено в 0x3F80000
__saturate (x)	min(0,max(1, x))
__float_to_int_[rn, rz, rU,rd] (x)	Приведение float-значения к целочисленному значению с заданным округлением
__float_to_uint_[rn, rz, ru,rd] (x)	Приведение float-значения к беззнаковому целочисленному значению с заданным округлением
__int_to_float_[rn, rz, ru,rd] (x)	Приведение целочисленного значения к float-значению с заданным округлением
__uint_to_float_[rn, rz, ru,rd] (x)	Приведение беззнакового целочисленного значения к float-значению с заданным округлением
__float_to_ll_[rn, rz, ru,rd] (x)	Приведение float-значения к 64битовому целочисленному значению с заданным округлением
__float_to_ull_[rn, rz, ru,rd] (x)	При ведение float-значения к 64битовому беззнаковому целочисленному значению с заданным округлением

Кроме ряда оптимизированных функций для работы с числами с плавающей точкой, также есть ряд "быстрых" функций для работы с целыми числами, приводимых в табл. 1.3.

Таблица 1.3. Целочисленные функции в CUDA

Функция	Значение
__[u]mul24(x, y)	Вычисляет произведение младших 24 бит целочисленных параметров x и y, возвращает младшие 32 бита результата. Старшие 8 бит аргументов игнорируются.
__[u]mulhi (x, y)	Возвращает старшие 32 бита произведения целочисленных операндов x и y
__[u]muI64hi (x, y)	Вычисляет произведение 64-битовых целых чисел и возвращает младшие 64 бита этого произведения
__[u]sad (x, y, z)	Возвращает z + |x – y|

__clz (x)	Возвращает целое число от 0 до 32 включительно последовательных нулевых битов для целочисленного параметра x, начиная со старших бит
__clzll (x)	Возвращает целое число от 0 до 64 включительно последовательных нулевых битов для целочисленного 64-битового параметра x, начиная со старших бит
__ffs (x)	Возвращает позицию первого (наименее значимого) единичного бита для аргумента x. Если x равен нулю, то возвращается нуль
__ffsll (x)	Возвращает позицию первого (наименее значимого) единичного бита для целочисленного 64битового аргумента x. Если x равен нулю, то возвращается нуль
__popc (x)	Возвращает число бит, которые равны единице в двоичном представлении 32битового целочисленного аргумента x
__popcll (x)	Возвращает число бит, которые равны единице в двоичном представлении 64битового целочисленного аргумента x
__brev (x)	Возвращает число, полученное перестановкой (то есть биты в позициях k и 31-k меняются местами для всех k от 0 до 31) битов исходного 32-битового целочисленного аргумента x
__brevll (x)	Возвращает число, полученное перестановкой (то есть биты в позициях k и 63-k меняются местами для всех k от 0 до 63) битов исходного 64-битового целочисленного аргумента x

Использование этих функций может заметно поднять быстродействие программы, однако при этом следует иметь в виду, что выигрыш от использования некоторых функций (__mul24) может исчезнуть для GPU следующих поколений.

Установка CUDA на компьютер

Для установки CUDA на компьютер необходимо по адресу

http://www.nvidia.com/object/cuda_get.html

скачать и установить на свой компьютер следующие файлы: CUDA driver, CUDA Toolkit и CUDA SDK.

Перейти в веб-браузере на эту страницу, выбрать в комбо-боксе тип и разрядность своей операционной системы, после чего на экранной форме появятся ссылки для скачивания нескольких последних версий CUDA.

Версия CUDA для платформы Linux также содержит CUDA-отладчик (cuda-gdb), устанавливаемый как отдельная компонента.

После установки всех этих компонент можно будет посмотреть готовые примеры, изучить прилагаемую документацию и начать писать свои программы с использованием CUDA.

Для компиляции программ на CUDA также потребуется установленный компилятор с C/C++. В качестве такого компилятора может выступать компилятор, входящий в состав Microsoft Visual Studio, а также компиляторы rningw и cygwin. Это связано с тем, что используемый для компиляции программ на CUDA компилятор nvcc использует внешний компилятор для компиляции частей кода, выполняемых на CPU.

Компиляция программ на CUDA

Традиционно программы на CUDA имеют расширение .cu. Для их компиляции используется утилита nvcc, входящая в состав CUDA Toolkit. Данная утилита разделяет код для CPU и код для GPU, используя для компиляции кода на CPU внешний

компилятор. У этой утилиты есть очень много входных ключей, основные из которых приведены в табл. 1.4.

Таблица 1.4. Опции команды nvcc

Опция	Значение
--help -h	Получение справки по всем опциям команды
--cuda -cuda	Откомпилировать все входные .cu-файлы в файлы .cu.c
--cubin -cubin	Откомпилировать все входные .cuj/.ptxj/.gpu-файлы в .cubin-файлы (для выполнения при помощи driver API). В этом случае весь код для CPU отбрасывается
--ptx -ptx	Откомпилировать все входные .cu/.gpu-файлы в .ptx-файлы (специальный ассемблер). В этом случае весь код для CPU отбрасывается
--gpu -gpu	Откомпилировать все входные .cu-файлы в .ptx-файлы (специальный ассемблер). В этом случае весь код для CPU отбрасывается
--compile -c	Откомпилировать все входные .c/.cc/.cpp/.cxx/.cu в .obj-файлы
--link -link	Эта опция задает режим компиляции и линковки всех входных файлов
--lib -lib	Откомпилировать все входные файлы и объектные файлы и добавить к заданной выходной библиотеке
--output-file<file> -o<file>	Задает имя и местоположение выходного файла. При этом допускается только один входной файл, кроме случая, когда происходит линковка
--pre-include < include-file > -include <include-file>	Задать заголовочные файлы, которые должны быть подключены во время препроцессинга
--library < library> -1 <library>	Задать библиотеки, которые необходимо использовать при сборке программы, указанные библиотеки ищутся в каталогах, заданных при помощи опции -L
--define-macro <macrodef> -D <macrodef>	Задать макросы для препроцессирования и компиляции
--undefine-macro <macrodef> -U <macrodef>	Задать макросы, определения которых необходимо удалить для препроцессирования и компиляции
--include-path <include-path> -I <include-path>	Задать пути для подключения файлов по директиве #include
--library-path <library-path> -L <library-path >	Задать пути для поиска библиотек
--output-directory <directory> -odir <directory>	Задать каталог для размещения выходного файла
--compiler-bindir <directory> -ccbin <directory>	Задать каталог, где расположен компилятор с C/C++. По умолчанию компилятор ищется стандартным способом (по путям, заданным переменной окружения path)
--profile -pg	Включить инструментирование выходного кода для использования утилиты gprof (только для Linux)
--debug -g	Включить создание отладочной информации для кода, выполняемого на CPU
--device-debug <level>	Включить создание отладочной информации для кода,

-G <level>	выполняемого на CPU, и также задать уровень оптимизации. Допустимые значения для уровня оптимизации: 0,1,2,3
--optimize <level> -O <level>	Задать уровень оптимизации для кода, выполняемого на CPU
--machine <bits> -m <bits>	Задать использование 32 или 64битовой архитектуры. Допустимые значения: 32,64. Значение по умолчанию: 32
--compiler-options<options>,... -Xcompiler <options>	Задать опции для препроцессора/компилятора (с C/C++)
--linker-options <options>,... -Xlinker <options>	Задать опции для сборки программы
--ptxas-options <options>,... -Xptxas <options>,...	Задать опции для оптимизирующего ассембера ptx
--gpu-architecture <gpu-architecture-name> -arch <gpu-architecture-name>	Задать класс Nvidia GPU, для которого необходимо осуществить компиляцию. Как правило, заданная архитектура должна быть виртуальной (например, compute_10). Эта опция не вызывает генерации кода, ее цель – управление стадией nvopencc, задавая архитектуру промежуточного ptx-кода (см. опция –gpu-code). Для удобства принято следующее соглашение: если не указано значения для опции --gpu-code, то в качестве этого значения берется значение параметра --gpu--architecture. В этом случае можно задавать невиртуальную архитектуру (например, sm_13), тогда nvcc выберет наиболее близкую виртуальную архитектуру. Например, 'nvcc-arch=sm_13' эквивалентно 'nvcc-arch=compute_13-code=sm_13'. Допустимыми значениями для этой опции являются: 'compute_10', 'compute_11', 'compute_12', 'compute_13', 'sm_10', 'sm_11', 'sm_12' 'sm_13'
--gpu-code <gpu-architecture-name> , -code <gpu-architecture-name>	Задать имя архитектуры GPU, для которой генерируется код. При этом nvcc встраивает в выходной выполнимый файл код для каждой из указанных архитектур. Это будет настоящий бинарный код для данной архитектуры (например, sm_13) или ptx-код для каждой «виртуальной» архитектуры (например, compute_10). На шаге выполнения, если подходящий бинарный образ не найден, но есть подходящий ptx-код, то он будет "на ходу" откомпилирован в код для конкретного GPU. Указанные архитектуры могут быть как «настоящими», так и виртуальными, но каждая из этих архитектур должна быть совместимой с архитектурой заданной командой '--gpu-architecture'. Например, 'arch'=compute_ 13 не совместима с 'code'=sm_10, поскольку сгенерированный ptx-код будет предполагать поддержку возможностей архитектуры compute_3, многие из которых не поддерживаются на

	sm_10. Допустимыми значениями для данного параметра являются: 'compute_10', 'compute_11', 'compute_12', 'compute_13', 'sm_10', 'sm_11', 'sm_12' 'sm_13'
--maxrregcount <N>, -maxrregcount <N>	Задать максимальное число регистров, которое может использовать функция на GPU. До определенного предела увеличение этого значения будет приводить к увеличению быстродействия. Однако поскольку общее число регистров в мультипроцессоре ограничено, то увеличение этого параметра также снижает максимальный размер блока, что ведет к уменьшению параллелизма. Таким образом, хорошее значение параметра maxrregcount является результатом баланса. Если эта опция не задана, то никакого ограничения на число регистров не предполагается. В противном случае заданное значение будет округлено до следующего гратного 4 значения
--device-emulation, -deviceemu	Осуществить компиляцию для режима эмуляции GPU
--use_fast_math, -use_fast_math	Заменить все вызовы float-функций на их быстрые аналоги

Для того, чтобы просто откомпилировать программу, состоящую из одного или нескольких файлов, сразу в выполняемый файл, можно воспользоваться следующей командой (для Microsoft Windows):

nvcc myfile1.cu myfile2.cu myfile3.cpp -o myprogram.exe

Для Linux и Mac OS X команда выглядит точно так же, только не указывается расширение .exe для выполнимого файла:

nvcc myfile1.cu myfile2.cu myfile3.cpp -o myprogram

Для сборки проектов, состоящих из многих файлов, можно воспользоваться утилитой make (или ее аналогом в Microsoft Windows – утилитой nmake) или же использовать Microsoft Visual Studio.

Для сборки при помощи утилиты nmake (Microsoft Windows) используется файл Makefile.nmake, соответствующая команда выглядит следующим образом:

nmake f Makefile.nmake

Обратите внимание, что для нескольких последних версий CUDA запуск пусс из команды nmake приводит к ошибке, для борьбы с этим достаточно "завернуть" вызов пусс в .bat-файл (в примерах используется файл пусс), приводимый ниже, после чего вызывать уже этот .bat-файл:

nvcc %nvcc_args%

```
# Пример Makefile для Microsoft Windows

!include <win32.mak>
EXES = info.exe
all: $(EXES)
info.exe: info.cu
  set nvcc_args= info.cu -o $@
  _nvcc.bat

clean:
  @del $(EXES) *.ilk *.pdb *.linkinfo 2> nul


# Пример Makefile для Linux
all: info

info: info.cu
  nvcc info.cu -o $@

clean:
rm -r -f incr info *.o $(OBJS) 2> /dev/null
```

При разработке программы под Microsoft Windows можно использовать Microsoft Visual Studio для компиляции и отладки проектов. Для этого необходимо создать новый проект и добавить к нему все используемые .cu-файлы. Однако поскольку сама среда не знает, как их нужно компилировать, следует использовать специальный файл, задающий правила компиляции этих файлов (Custom Build Step). CUDA SDK содержит такой файл – Cuda.Rules.

При создании нового проекта при первом же добавлении .cu файла выдается сообщение о том, что нет информации о том, что нужно делать с подобными файлами, и будет предложено создать файл с такими правилами.

Выберите создание нового файла с правилами, создайте его (в следующем диалоге задайте в качестве имени Cuda.Rules и сохраните его). После этого сохраните проект и замените созданный файл Cuda.Rules на файл Cuda.Rules из CUDA SDK.

После этого снова открывайте проект, теперь Visual Studio уже знает, что надо делать с .cu-файлами, осталось только подключить библиотеки. Для этого в свойствах проекта откройте пункт **Linker** и в поле **Additional Library Directories** введите $(CUDA_LIB_PATH). В пункте **Linker|Input** в поле **Additional Dependencies** введите используемые библиотеки CUDA (для работы с CUDA runtime API это будет cudart.lib).

Также можно воспользоваться проектом CUDA VS Wizard, бесплатно скачиваемым по адресу

http://sourceforge.net/projects/cudavswizard/ .

Данный проект добавляет к Microsoft Visual Studio новый тип проекта.

Первая программа

Поскольку мы изучаем язык CUDA C, то и начнем с традиционного примера – написания программы "Здравствуй, CUDA!"

```
#include "../common/book.h"
#include <stdio.h>

int main(void)
{  printf ("Hello, CUDA!\n");
   return 0;
}
```

Своей "похожестью" на обычный C пример обязан тому факту, что работает целиком на центральном процессоре – *CPU (host)* и полностью игнорирует любые вычислительные устройства, кроме него. Т.е. инструментальные средства NVIDIA просто передают код компилятору для *CPU*, и все работает так, будто CUDA вообще не существует.

Рассмотрим функцию, исполняемую на *GPU (device, устройство)* и называемую *ядром (kernel).*

```
#include <stdio.h>
#include <conio.h>

__global__ void kernel(void)
{
}

int main(void)
{  kernel<<<1, 1>>>();
   printf ("Hello, CUDA!\n");
   return 0;
}
```

В этой программе есть два добавления к первоначальному примеру: пустая функция `kernel()` с квалификатором `__global__` и вызов этой функции, сопровождаемый синтаксисом `<<<1, 1>>>`. Т.е. CUDA C дополняет стандартный язык C квалификатором `__global__`. Тем самым компилятору сообщается, что эта функция должна компилироваться для исполнения *устройством*, а не *CPU*. Отсюда следует, что нужен признак, позволяющий отправлять код для *CPU* одному компилятору, а для *GPU* – другому. Самое главное здесь – вызвать код для *GPU* из кода, исполняемого *CPU*. Угловыми скобками обозначаются аргументы, которые должны передаваться исполняющей среде. Их не следует путать с аргументами функции, исполняемой *устройством*, т.к. они представляют собой параметры, влияющие на то, как исполняющая среда будет запускать код для *устройства*. Аргументы же самой функции передаются как обычно – в круглых скобках.

Передача параметров

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include "conio.h"

static void HandleError( cudaError_t err, const char *file, int line )
{
    if (err != cudaSuccess)
     {
      printf("%s in %s at line %d\n", cudaGetErrorString(err), file, line);
      exit( EXIT_FAILURE );
     }
}

#define HANDLE_ERROR( err ) (HandleError( err, __FILE__, __LINE__ ))

__global__ void add(int a, int b, int *c)
{ *c = a + b;
}

int main(void)
{  int c;
   int *dev_c;
   HANDLE_ERROR (cudaMalloc((void**)&dev_c, sizeof(int)));
   add<<<1, 1>>>(2, 7, dev_c);
   HANDLE_ERROR (cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost));
   printf ("2 + 7 = %d\n", c);
   cudaFree(dev_c);
   getch();
   return 0;
}
```

Здесь две новых концепции – 1) ядру можно передавать параметры, как любой другой функции; 2) необходимо выделить память, если хотим, чтобы *устройство* сделало нечто полезное, например вернуло данные *CPU*.

Передача параметров ядру мало чем отличается от передачи параметров обычной функции в C.

Более интересно выделение памяти с помощью функции `cudaMalloc()`. Она похожа на стандартную функцию `malloc()`, но говорит исполняющей среде CUDA, что память должна быть выделена на *устройстве*. Первый аргумент – это указатель на указатель, в котором будет возвращен адрес выделенной области памяти, а второй – размер этой области.

Сформулируем ОЧЕНЬ ВАЖНЫЕ правила использования указателей на память *устройства*:

1. ***разрешается*** передавать указатели на память, выделенную `cudaMalloc()`, функциям, исполняемым *устройством*,
2. ***разрешается*** использовать указатели на память, выделенную `cudaMalloc()`, для чтения и записи в эту память в коде, исполняемом *устройством*,

3. ***разрешается*** передавать указатели на память, выделенную `cudaMalloc()`, функциям, исполняемым *CPU*,
4. ***запрещается*** использовать указатели на память, выделенную `cudaMalloc()`, для чтения и записи в эту память в коде, исполняемом *CPU*.

Все перечисленное можно сказать и об указателях на память *CPU*.

Для освобождения памяти, выделенной функцией `cudaMalloc()` следует пользоваться функцией `cudaFree()`, но не функцией `free()`.

Мы знаем, как выделять и освобождать память *устройства*, но модифицировать эту память в коде, исполняемом *CPU*, нельзя. В приведенном примере иллюстрируются два способа доступа к памяти: 1) использование возвращенных указателей в коде, исполняемом самим *устройством* и 2) обращение к функции `cudaMemcpy()`.

Остановимся на втором способе. Функция `cudaMemcpy()` похожа на стандартную функцию `memcpy()`, но принимает дополнительный параметр, который говорит о том, какой из двух указателей адресует память устройства:

- ✓ `cudaMemcpyDeviceToHost` – исходные данные находятся в памяти *устройства*, а конечный адрес – в памяти *CPU*,
- ✓ `cudaMemcpyHostToDevice` – исходные данные находятся в памяти *CPU*, а конечный адрес – в памяти *устройства*,
- ✓ `cudaMemcpyDeviceToDevice` – оба указателя относятся к памяти *устройства*.

Если же и начальный, и конечный указатели адресует память *CPU*, то используется стандартная функция `memcpy()`.

Получение информации об устройстве

Прежде чем начать работу с GPU, очень важно получить максимально полную информацию обо всех имеющихся GPU (CUDA может работать сразу с несколькими GPU, если только они не соединены через SLI) и об их возможностях. Для этого можно воспользоваться CUDA runtime API, который предоставляет простой способ получить информацию об имеющихся GPU, которые могут быть использованы CUDA, и обо всех их возможностях. Информация о возможностях GPU возвращается в виде структуры cudaDeviceProp.

```
struct cudaDeviceProp
{
 char name[256] ;            // Название устройства.
 size t totalGlobalMem;      // Полный объем глобальной памяти в байтах.
 sizet sharedMemPerBlock;    // Объем разделяемой памяти в блоке в байтах.
 int regsPerBlock;           // Количество 32-битовых регистров в блоке.
 int warpSize;               // Размер warp'а.
 size t memPitch;            // Максимальный pitch в байтах, допустимый
                        // функциями копирования памяти, выделенной через
                        // cudaMallocPitch
 int maxThreadsPerBlock;     // Максимальное число активных нитей в блоке.
 int maxThreadsDim [3];   // Максимальный размер блока по каждому измерению.
 int maxGridSize [3];     // Максимальный размер сетки по каждому измерению.
 sizet totalConstMem;        // Объем константной памяти в байтах.
 int major;                  // Compute Capability, старший номер.
 int minor;                  // Compute Capability, младший номер.
 int clockRate;              // Частота в килогерцах.
 size t textureAlignment;    // Выравнивание памяти для текстур.
 int deviceOverlap;          // Можно ли осуществлять копирование параллельно
                             // с вычислениями.
```

```
 int multiProcessorCount;          // Количество мультипроцессоров в GPU.
 int kernelExecTimeoutEnables;     // 1, если есть ограничение на время
                                   // выполнения ядра
 int integrated;              // 1, если GPU встроено в материнскую плату
 int canMapHostMemory;        // 1, если можно отображать память CPU в память
                              // CUDA, для использования функциями
                              // cudaHostAlloc,cudaHostGetDevicePointer

 int computeMode;             // Режим, в котором находится GPU.
                              // Возможные значения:
                              // cudaComputeModeDefault,
                              // cudaComputeModeExclusive – только одна нить
                              //  может вызвать cudaSetDevice для данного GPU,
                              // cudaComputeModeProhibited – ни одна нить не
                              // может вызвать cudaSetDevice для данного GPU.
};
```

Для обозначения возможностей GPU CUDA использует понятие Compute Capability, выражаемое парой целых чисел major.minor. Первое число обозначает глобальную архитектурную версию, второе – небольшие изменения. Так, GPU GeForce 8800 Ultra/GTX/GTS имеют Compute Capability, равную 1.0.

Ниже приводится исходный текст простой программы, перечисляющей все доступные GPU и их основные возможности.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include "conio.h"

int main (int argc, char * argv [] )
{
  int deviceCount;
  cudaDeviceProp devProp;
  cudaGetDeviceCount ( &deviceCount );
  printf ( "Found %d devices\n", deviceCount );
  for ( int device = 0; device < deviceCount; device++)
   {cudaGetDeviceProperties ( &devProp, device );

    printf ("Device %d\n", device );
    printf ("Compute capability    : %d.%d\n", devProp.major, devProp.minor);
    printf ("Name                  : %s\n", devProp.name);
                              // Полный объем глобальной памяти в Мбайтах:
    printf ("Total Global Mem: %d\n", (devProp.totalGlobalMem/(1024*1024)));
    printf ("Shared memory per block: %d\n" , devProp.sharedMemPerBlock );
    printf ("Registers per block   : %d\n", devProp.regsPerBlock);
    printf ("Warp size       : %d\n", devProp.warpSize);
    printf ("Max threads per block: %d\n", devProp.maxThreadsPerBlock);
    printf ("Total constant memory: %d\n", devProp.totalConstMem);
    printf ("Clock Rate            : %d\n", devProp.clockRate);
    printf ("Texture Alignment     : %u\n", devProp.textureAlignment);
    printf ("Device Overlap  : %d\n", devProp.deviceOverlap);
```

```
    printf ("Multiprocessor Count: %d\n", devProp.multiProcessorCount);
    printf ("Max Threads Dim : %d %d %d\n", devProp.maxThreadsDim[0],
             devProp.maxThreadsDim[1], devProp.maxThreadsDim[2] );
    printf ("Max Grid Size   : %d %d %d\n", devProp.maxGridSize [0],
             devProp.maxGridSize [1], devProp.maxGridSize [2]);
   }
  return 0;
}
```

Лабораторное задание

Задание 1. Набрать в редакторе среды программирования две версии программы "Hello, CUDA!", скомпилировать, отладить, запустить. Выполнить сравнительный анализ версий.

Задание 2. . Набрать в редакторе среды программирования программу, содержащую функцию с параметрами, скомпилировать, отладить, запустить. Организовать возвращение результата с помощью оператора *return*.

Задание 3. Набрать в редакторе среды программирования программу получения информации об устройстве, скомпилировать, отладить, запустить. Проанализировать полученные данные.