МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»

Институт информатики, математики и электроники
Факультет информатики
Кафедра технической кибернетики

Параллельное программирование

Методические указания
к лабораторной работе №4

САМАРА 2017

Составитель: доц., Козлова Е.С.

УДК 519.681

Методические указания к лабораторным работам

Самарский государственный аэрокосмический университет имени академика С.П.КОРОЛЁВА (национальный исследовательский университет)

Составитель: Е.С. Козлова

Самара, 2018. 10 с.

Методические указания предназначены для бакалавров направления 010400.62 «Прикладная математика и информатика»

Печатается по решению редакционно-издательского совета Самарского университета

Рецензент:

1 Технология CUDA

1.1 Графические ускорители

Термин Graphics Processing Unit (GPU) был впервые использован корпорацией NVIDIA для обозначения того факта, что графический ускоритель, первоначально используемый только для ускорения трехмерной графики, стал мощным программируемым устройством (процессором), пригодным для решения значительно более широкого класса вычислительных задач (General Purpose computations on GPU – GPGPU). Современные GPU представляют собой массивно-паралельные вычислительные устройства с производительностью порядка Терафлопса и большим объемом собственной памяти (DRAM).

Основные отличия CPU и GPU заключаются в следующем: во-первых, CPU имеет лишь небольшое число ядер, работающих на высокой тактовой частоте независимо друг от друга. GPU же напротив работает на низкой тактовой частоте и имеет сотни сильно упрощенных вычислительных элементов (например, отсутствует предсказатель ветвлений и суперскалярное исполнение команд); во-вторых, значительная доля площади кристалла CPU занята кэшем, в то время как практически весь GPU состоит из арифметико-логических блоков. В архитектуре GPU кэш имеет меньшее значение, поскольку используется принципиально иная стратегия покрытия латентности памяти. За счет этих отличий производительность каждого нового поколения GPU быстро растет как в пиковом значении, так и на реальных приложениях, например, Linpack

1.2 Модель программирования CUDA

Compute Unified Device Architecture (CUDA) – это программная модель, включающая описание вычислительного параллелизма и иерархичной структуры памяти непосредственно в язык программирования. С точки зрения программного обеспечения, реализация CUDA представляет собой кроссплатформенную систему компиляции и исполнения программ, части которых работают на CPU и GPU. CUDA предназначена для разработки GPGPU-приложений без привязки к графическим API и поддерживается всеми GPU NVIDIA, начиная с серии GeForce 8.

В литературе о CUDA основная система, к которой подключен GPU, для краткости называется термином хост (host), аналогично сам GPU по отношению к хосту часто называется просто устройством (device). CUDA-программа задействует как CPU, так и GPU, на CPU выполняется последовательная часть кода и подготовительные стадии для GPU-вычислений. Параллельные участки кода могут быть перенесены на GPU, где будут одновременно выполняться большим множеством нитей (threads). Важно отметить ряд принципиальных различий между обычными потоками CPU и нитями GPU:

• Нить GPU чрезвычайно легковесна, ее контекст минимален, регистры распределены заранее;

• Для эффективного использования ресурсов GPU программе необходимо задействовать тысячи отдельных нитей, в то время как на многоядерном CPU максимальная эффективность обычно достигается при числе потоков, равном или в несколько раз большем количества ядер.

В целом работа нитей на GPU соответствует принципу SIMD, однако есть существенное различие. Только нити в пределах одной группы (для GPU архитектуры Fermi – 32 нити), называемой варпом (warp) выполняются физически одновременно. Нити различных варпов могут находиться на разных стадиях выполнения программы. Такой метод обработки данных обозначается термином SIMT (Single Instruction – Multiple Threads). Управление работой варпов производится на аппаратном уровне.

Нить GPU имеет координаты во вложенных трехмерных декартовых равномерных сетках «индексы блоков» и «индексы потоков внутри каждого блока», двумерный случай показан на рис. 1. В контексте каждой нити значения координат и размерностей доступны через встроенные переменные threadIdx, blockIdx и blockDim, gridDim соответственно. Если проводить аналогию с Message Passing Interface (MPI), то значения этих переменных по смыслу аналогичны результатам функций MPI Comm rank и MPI Comm size.

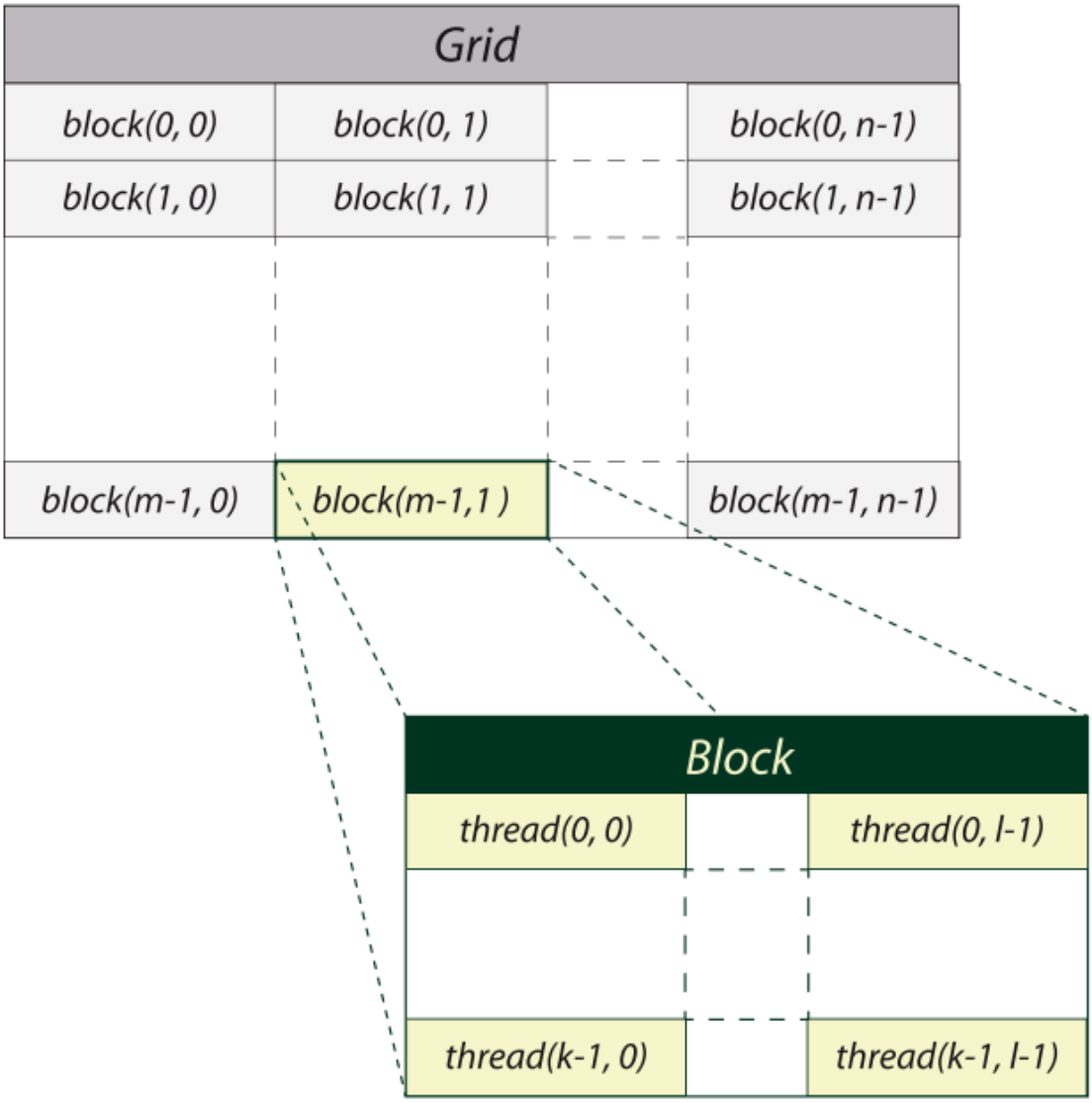

Рис. 1. Иерархия нитей в CUDA

Функция, которая выполняет основной алгоритм на графическом ускорителе называется функцией ядра. Код ядра начинается с определения глобального индекса обрабатываемых данных, зависящего от координат нити. В общем случае соответстве нитей и частей задачи может быть любым, например, одна нить может обрабатывать не один элемент массива или определенный диапазон. В течение времени работы ядра

декартовы сетки нитей и блоков зафиксированы для отображения на аппаратный уровень вычислительных элементов GPU. Каждая нить производит некоторую работу (например, сложение элементов массивов a и b) и помещает результат в память (элемент массива c). После этого нить завершает работу.

1.3 Программы с использованием CUDA

В самой программе необходимо выполнить следующие этапы:

1. Получить данные для расчетов.
2. Скопировать эти данные в GPU память.
3. Произвести вычисление в GPU через функцию ядра.
4. Скопировать вычисленные данные из GPU памяти в ОЗУ.
5. Посмотреть результаты.
6. Высвободить используемые ресурсы.

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

```
cudaError_t cudaMalloc (void **devPtr, size_t size);
```

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

```
cudaError_t cudaFree (void *devPtr);
```

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

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

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

Отметим, что для замера времени исполнения ядра используется механизм событий (Events) в CUDA. В начале создаются временные метки start, stop с помощью cudaEventCreate(), далее фиксируется время старта с помощью cudaEventRecord(), выполняются вычисления, фиксируется время завершения. Поскольку некоторые операции могут выполняться асинхронно вызывается cudaEventSynchronize(), что бы метка записалась корректно. Далее вычисляется разница во времени с помощью cudaEventElapsedTime() и печатается результат.

2 Задание на лабораторную работу

Лабораторная работа 4

Произвести запуск программы для поиска суммы векторов, которая использует технологию CUDA, на различном количестве нитей. В ходе анализа работы программы оцените время ее выполнения на различном количестве исполняющих нитей.

Прежде всего необходимо подгрузить модули:

```
module load intel/icc
```

```
module load cuda/6.5
```

Для компиляции необходимо использвать Makefile, который можно запустить набрав команду

```
make
```

в консоли, находясь в одной дирректории с данным файлом. Отметим, что программа с использованием технологии CUDA на кластере представлена в виде двух отдельных файлов: main-функции с расширением .c и файла ядра с расширением .cu. Заготовки программ представлены ниже.

Для запуска задачи используйте код скрипта, представленный ниже.

Пример 1. Код main-функции

```
include <cublas_v2.h>
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>


int main(int argc, char* argv[])
{

    int n = atoi(argv[1]);

    printf("n = %d\n", n);

    int n2b = n * sizeof(int);
     int n2=n;

    // Выделение памяти на хосте
     int * a=(int*)calloc(n2,sizeof(int));
     int * b=(int*)calloc(n2,sizeof(int));
     int * c=(int*)calloc(n2,sizeof(int));
     // Инициализация массивов

    // Выделение памяти на устройстве
    int* adev = NULL;
    cudaError_t cuerr = cudaMalloc((void**)&adev, n2b);
    if (cuerr != cudaSuccess)
    {
```

```
        fprintf(stderr, "Cannot allocate device array for a:
%s\n",
            cudaGetErrorString(cuerr));
        return 0;
    }

    int* bdev = NULL;
    cuerr = cudaMalloc((void**)&bdev, n2b);
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot allocate device array for b:
%s\n",
            cudaGetErrorString(cuerr));
        return 0;
    }

    int * cdev = NULL;
    cuerr = cudaMalloc((void**)&cdev, n2b);
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot allocate device array for c:
%s\n",
            cudaGetErrorString(cuerr));
        return 0;
    }

    // Создание обработчиков событий
    cudaEvent_t start, stop;
    float gpuTime = 0.0f;
    cuerr = cudaEventCreate(&start);
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot create CUDA start event: %s\n",
            cudaGetErrorString(cuerr));
        return 0;
    }

    cuerr = cudaEventCreate(&stop);
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot create CUDA end event: %s\n",
            cudaGetErrorString(cuerr));
        return 0;
    }

     // Копирование данных с хоста на девайс
    cuerr = cudaMemcpy(adev, a, n2b, cudaMemcpyHostToDevice);
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot copy a array from host to
device: %s\n",
            cudaGetErrorString(cuerr));
        return 0;
```

```
    }

    cuerr = cudaMemcpy(bdev, b, n2b, cudaMemcpyHostToDevice);
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot copy b array from host to
device: %s\n",
            cudaGetErrorString(cuerr));
        return 0;
    }

    // Установка точки старта
    cuerr = cudaEventRecord(start, 0);
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot record CUDA event: %s\n",
            cudaGetErrorString(cuerr));
        return 0;
    }

    //Запуск ядра
    kernel<<< GRID_SIZE, BLOCK_SIZE >>>(cdev, adev, bdev, n);

    cuerr = cudaGetLastError();
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot launch CUDA kernel: %s\n",
            cudaGetErrorString(cuerr));
        return 0;
    }

     // Синхронизация устройств
    cuerr = cudaDeviceSynchronize();
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot synchronize CUDA kernel: %s\n",
            cudaGetErrorString(cuerr));
        return 0;
    }

     // Установка точки окончания
    cuerr = cudaEventRecord(stop, 0);
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot copy c array from device to
host: %s\n",
            cudaGetErrorString(cuerr));
        return 0;
    }

     // Копирование результата на хост
    cuerr = cudaMemcpy(c, cdev, n2b, cudaMemcpyDeviceToHost);
    if (cuerr != cudaSuccess)
```

```
    {
        fprintf(stderr, "Cannot copy c array from device to
host: %s\n",
            cudaGetErrorString(cuerr));
        return 0;
    }

     // Расчет времени
    cuerr = cudaEventElapsedTime(&gpuTime, start, stop);
    printf("time spent executing %s: %.9f seconds\n", "kernel",
gpuTime/1000);

    // Очищение памяти
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
    cudaFree(adev);
    cudaFree(bdev);
    cudaFree(cdev);
    free(a);
    free(b);
    free(c);


    return 0;
}
```

Пример 2. Код функции ядра

```
__global__ void addKernel(int *c,  int *a,  int *b, unsigned int
size)
{
     // Код функции ядра
}

#define kernel addKernel
#include "mainGPU.h"
```

Пример 3. Код Makefile

```
# The size of shared memory block size
BLOCK_SIZE = 128
NVCC = nvcc
CFLAGS = -g -G -O0 -DBLOCK_SIZE=$(BLOCK_SIZE) -lcublas
Add: addGPU.cu mainGPU.h
     $(NVCC) $(CFLAGS) $< -o $@
```

Пример 4. Код скрипта для запуска

```
#!/bin/bash
#PBS -N matMulGPU
#PBS -l walltime=00:01:10
```

```
#PBS -l nodes=1:ppn=1:gpu
#PBS -j oe
#PBS -A tk
cd $PBS_O_WORKDIR

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/COMMON/cuda-
6.5/lib64
./Add 20000000
```