МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное
образовательное учреждение высшего образования
«Самарский национальный исследовательский университет
имени академика С.П. Королева»
(Самарский университет)

Институт информатики, математики и электроники
Факультет информатики
Кафедра технической кибернетики

ЛАБОРАТОРНАЯ РАБОТА №4

Поиск суммы элементов с использованием технологии CUDA

по курсу
Параллельное программирование

Студент гр. 6407 _____ В.С. Гринина
Преподаватель,
к.ф.-м.н. _____ Е.С. Козлова

САМАРА 2018

ЗАДАНИЕ

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

Для запуска задачи использовать код скрипта, представленный методических указаниях. [1]

ВВЕДЕНИЕ

Раннее рассмотренные технологии OpenMP и MPI использовали вычислительную мощь CPU. Однако существует альтернативный подход к параллельным вычислениям. А именно, использование GPU. Изначально GPU был ориентирован на обработку графических данных. Поэтому использование его в вычислениях приводит к выигрышу по времени, когда исходные данные представляют собой матрицы, в самой программе отсутствуют ветвления, каждый из результатов вычисляется из подмножества данных, а операции над данными производятся с плавающей запятой. [3]

Задач вышеописанного типа в реальной жизни очень много (например, задачи управления или математической физики). Так что применение GPU в параллельных алгоритмах является актуальной темой.

В данной лабораторной работе ознакомимся с запуском программы с использованием технологии CUDA, а также с ее особенностями работы с памятью.

1 Ход выполнения работы

Прежде всего перед запуском программы нужно написать функцию ядра, которая бы производила подсчет суммы на отдельной нити. Было написано два варианта такого ядра:

1. Каждая нить обрабатывает непрерывный блок исходного массива;
2. Каждая обертка нитей за тик запрашивает непрерывный кусок исходного массива.

Так как каждая обертка обрабатывается за тик целиком, то предполагается, что второй способ написания ядра будет давать выигрыш во времени.

Модель логического устройства CUDA представлена на рисунке 1. [3]

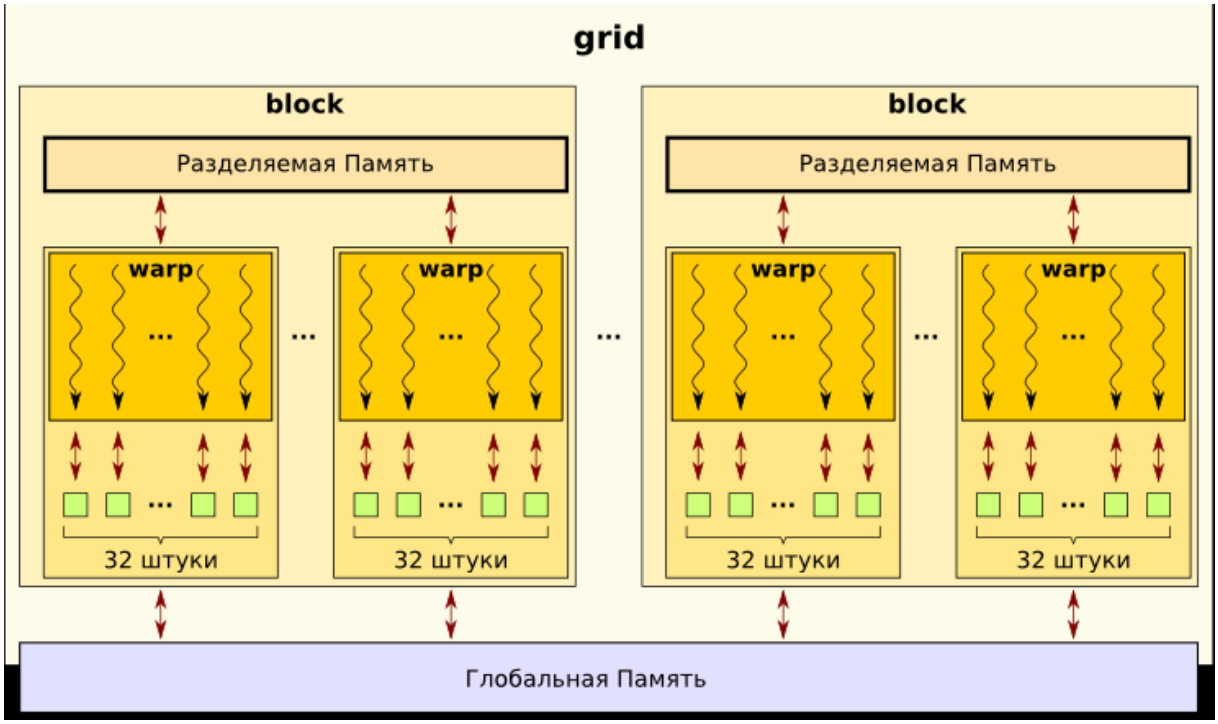

Рисунок 1 – логическое устройство CUDA

Запуск программы осуществлялся с помощью скриптов, которые представлены в приложении. Код программы также есть в приложении.

Стоит заметить, что если честно и добросовестно обрабатывать возможные ошибки, то код с использованием технологии CUDA становится

более чем громоздким. Однако, без обработок различных ошибок код смотрится довольно лаконично.

Время выполнения каждого способа при разных размерностях векторов было засечено и записано в таблицу 1.

Таблица 1 – Время выполнения программ при различных параметрах, с

Размерность вектора	1500000	100
1 способ	0.000636	0.000061
2 способ	0.000314	0.000060

Из таблицы 1 можно сделать вывод, что при малых размерностях вектора непосредственно способ обращения к памяти не имеет значения. Однако, при достаточно большой размерности вектора наблюдается незначительное преимущество второго способа написания функции ядра. Можно сделать предположение, что если увеличить размер вектора еще больше, то выигрыш по времени будет наблюдаться более отчетливо.

ЗАКЛЮЧЕНИЕ

В ходе проведения данной лабораторной работы были получены базовые представления о технологии CUDA и работе с GPU. Также получен практический опыт написания CUDA программ.

Были написаны две функции ядра, которые по-разному обращаются с памятью. Были проанализированы оба варианта написания ядра теоретически и практически. Оказалось, что на практике выигрыш варианта, который обращается с памятью более аккуратно, по времени заметен лишь на больших размерностях массива.

Было замечено, что технология CUDA рассчитана на больший объем вычислений, нежели OpenMP или MPI. Однако, в данной технологии выполняются более примитивные алгоритмы. Также стоит заметить, что с использованием технологии CUDA на программиста лежит ответственность за рациональное размещение алгоритма на исполняющих элементах, так как доступ к памяти реализован более хитро. [3]

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Козлова Е.С. Методические указания к лабораторной работе №4 по курсу «Параллельное программирование» Методические указания к лабораторным работам [Текст]/ Сост. Козлова Е.С. – Самара, 2017. 12 с.

[2] Параллельные вычисления [Электронный ресурс] // Википедия: свободная энцикл. – Электрон. дан. – [Б. м.], 2017. – URL: https://en.wikipedia.org/wiki/Concurrent_computing (дата обращения: 10.09.2018).

[3] Руководство по работе Cuda [Электронный ресурс] // Cuda Toolkit Documentation : – Электрон. дан. – [Б. м.], 2018. – URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (дата обращения: 21.11.2018).

[4] CUDA [Электронный ресурс] // Википедия: свободная энцикл. – Электрон. дан. – [Б. м.], 2018. – URL: https://en.wikipedia.org/wiki/CUDA (дата обращения: 15.11.2018).

[5] Страуструп, Б. Язык программирования C++ [Текст]/ / Б. Страуструп. – М:Бином, 2011 –1136 с

ПРИЛОЖЕНИЕ А
КОД ПРОГРАММЫ CUDA

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

__global__ void addKernel1(int *c, int *a, int *b, unsigned
int size) {
  int threadSize = (size + gridDim.x * blockDim.x - 1) /
gridDim.x / blockDim.x;
  int index = blockIdx.x * blockDim.x + threadIdx.x *
threadSize;
  for (int i = 0; i < threadSize && index + i < size; i++) {
    c[index + i] = a[index + i] + b[index + i];
  }
}
__global__ void addKernel2(int *c, int *a, int *b, unsigned
int size) {
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  while (index < size) {
    c[index] = a[index] + b[index];
    index += blockDim.x * gridDim.x;
  }
}
#define kernel addKernel1

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

  int n = 100;
  int n2b = n * sizeof(int);
  int n2=n;

  srand(time(NULL));
  // Выделение памяти на хосте
  int * a=(int*)calloc(n, sizeof(int));
  int * b=(int*)calloc(n, sizeof(int));
  int * c=(int*)calloc(n, sizeof(int));

  // Инициализация массивов
  int i;
  for (i = 0; i < n; i++) {
    a[i] = (int)rand()/100000;
    b[i] = (int)rand()/100000;
  }

  // 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;
  }


  int BLOCK_SIZE = 256;
  int GRID_SIZE = (n + BLOCK_SIZE - 1)/BLOCK_SIZE;
  //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);
  // printf("\na ");
  // for (i = 0; i < 3; i++) {
  //   printf("%.0d ", a[i]);
  // }
  // printf("...\n");
  // printf("b ");
  // for (i = 0; i < 3; i++) {
  //   printf("%.0d ", b[i]);
  // }
  // printf("...\n");
  // printf("c ");
  // for (i = 0; i < 3; i++) {
  //   printf("%.0d ", c[i]);
  // }
  // printf("...\n");

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

  return 0;
}
```

ПРИЛОЖЕНИЕ Б
КОД СКРИПТОВ ЗАПУСКА

Основной скрипт

```
#!/bin/bash

set -euxo pipefail

nvcc -g -G -O0 -lcublas -I/home/COMMON/cuda-
6.5/samples/common/inc/ main.cu -o main

qsub \
  -V \
  -l "nodes=1:ppn=1:gpu" \
  -l walltime=00:01:00 \
  -N "svxf_mpi_lab4_gpu" \
  -j oe \
  -A tk \
  -o ./output \
    qsub_script.sh
```

Скрипт непосредственно запуска

```
#!/bin/bash

cd $PBS_O_WORKDIR

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/COMMON/cuda-
6.5/lib64

./main
```