Отчет по лабораторной работе №2

Цель работы

Изучить иерархию памяти в CUDA, работу с разделяемой памятью и основы оптимизации программы в CUDA.

Задание 1

Даны два вектора A и B из N натуральных (ненулевых) элементов (задаются случайно). Вектора расположены в глобальной памяти. Написать программу на Си с использованием CUDA runtime API, выполняющую поэлементное умножение двух векторов на GPU так, чтобы продемонстрировать паттерны доступа к глобальной памяти, приводящие и не приводящие к объединению запросов в одну транзакцию. Измерить время работы программ. Написать программу для верификации результатов. Результаты занести в отчёт.

Код

```
#include <cuda_runtime.h>

#include <cstdlib>
#include <ctime>
#include <iostream>
#include <vector>

#include "Common.h"

#define BLOCK_SIZE 256
#define RUN_COUNT 10

__global__ void vectorMultiplyCoalesced(const int* A, const int* B, int* C,
int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) C[idx] = A[idx] * B[idx];
}

__global__ void vectorMultiplyNonCoalesced(const int* A, const int* B, int*
C, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < N - 1)
    {
        if(idx == 4)                    C[idx+1] = A[idx+1] * B[idx+1];
        else if(idx == 5)               C[idx-1] = A[idx-1] * B[idx-1];
        else if(idx > 1 << 19)          C[idx+1] = A[idx+1] * B[idx+1];
        else                            C[idx]   = A[idx]   * B[idx];
    }
```

```
    if(idx == N - 1) { C[524289] = A[524289] * B[524289];  }
}

void verifyResults(const std::vector<int>& A, const std::vector<int>& B,
const std::vector<int>& C, int N)
{
    for (int i = 0; i < N; i++)
    {
        if (C[i] != A[i] * B[i])
        {
            printf("Ошибка: элемент %d не совпадает!\n", i);
            return;
        }
    }
    printf("Результаты корректны!\n");
}

int main()
{
    const int N = 1 << 20;

    size_t bytes = N * sizeof(int);
    std::vector<int> h_A(N), h_B(N), h_C(N);

    int *d_A, *d_B, *d_C;

    dim3 threads(BLOCK_SIZE);
    dim3 blocks((N + BLOCK_SIZE - 1) / BLOCK_SIZE);

    for(int i = 0; i < RUN_COUNT; ++i)
    {
        srand(time(0));
        for (int i = 0; i < N; i++)
        {
            h_A[i] = rand() % 1000 + 1;
            h_B[i] = rand() % 1000 + 1;
        }

        cudaMalloc(&d_A, bytes);
        cudaMalloc(&d_B, bytes);
        cudaMalloc(&d_C, bytes);

        cudaMemcpy(d_A, h_A.data(), bytes, cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, h_B.data(), bytes, cudaMemcpyHostToDevice);

        {
            printf("\nРаботает Coalesced\n");
            CudaTimer t;
            vectorMultiplyCoalesced<<<blocks, threads>>>(d_A, d_B, d_C, N);
        }

        cudaMemcpy(h_C.data(), d_C, bytes, cudaMemcpyDeviceToHost);
        verifyResults(h_A, h_B, h_C, N);
```

```
    }

    printf("\nСреднее время выполнения Coalesced: %.5f мс\n",
CudaTimer::avgElapsedTime);

    CudaTimer::avgElapsedTime = 0;

    for(int i = 0; i < RUN_COUNT; ++i)
    {
        srand(time(0));
        for (int i = 0; i < N; i++)
        {
            h_A[i] = rand() % 1000 + 1;
            h_B[i] = rand() % 1000 + 1;
        }

        cudaMalloc(&d_A, bytes);
        cudaMalloc(&d_B, bytes);
        cudaMalloc(&d_C, bytes);

        cudaMemcpy(d_A, h_A.data(), bytes, cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, h_B.data(), bytes, cudaMemcpyHostToDevice);

        {
            printf("\nРаботает Non-Coalesced\n");
            CudaTimer t;
            vectorMultiplyNonCoalesced<<<blocks, threads>>>(d_A, d_B, d_C,
N);
        }

        cudaMemcpy(h_C.data(), d_C, bytes, cudaMemcpyDeviceToHost);
        verifyResults(h_A, h_B, h_C, N);
    }

    printf("\nСреднее время выполнения Non-Coalesced: %.5f мс\n",
CudaTimer::avgElapsedTime);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
```

Результаты

```
Работает Coalesced
Среднее время выполнения Coalesced: 0.06149 мс

Работает Non-Coalesced
Среднее время выполнения Non-Coalesced: 0.06688 мс
```

Задание 2 (0 вариант)

Написать программу на Си с использованием CUDA runtime API в соответствии с вариантом задания. Измерить время работы программы для различных значений параметров на CPU, GPU с использованием глобально памяти, GPU с использованием разделяемой памяти. Написать программу для верификации результатов. Результаты занести в отчёт.

Дана матрица A из NxN натуральных (ненулевых) элементов (задаются случайно). Матрица расположена в глобальной памяти. Написать программу, выполняющую транспонирование матрицы в двумерной сетке.

Код

```
#include <cuda_runtime.h>

#include <cstdlib>
#include <ctime>
#include <iostream>
#include <vector>

#include "Common.h"

#define BLOCK_SIZE 16
#define RUN_COUNT 1000

__global__ void transpose(const int* inData, int* outData, int n)
{
    unsigned int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int yIndex = blockIdx.y * blockDim.y + threadIdx.y;

    if (xIndex < n && yIndex < n)
    {
        unsigned int inIndex = yIndex * n + xIndex;
        unsigned int outIndex = xIndex * n + yIndex;

        outData[outIndex] = inData[inIndex];
    }
}

__global__ void transpose_shared(const int* inData, int* outData, int N)
{
    __shared__ float tile[BLOCK_SIZE][BLOCK_SIZE];

    int x = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    int y = blockIdx.y * BLOCK_SIZE + threadIdx.y;

    int width = gridDim.x * BLOCK_SIZE;
    if (x < N && y < N) tile[threadIdx.y][threadIdx.x] = inData[y*width +
x];
```

```
    __syncthreads();

    x = blockIdx.y * BLOCK_SIZE + threadIdx.x;
    y = blockIdx.x * BLOCK_SIZE + threadIdx.y;

    if (x < N && y < N) outData[y*width + x] = tile[threadIdx.x]
[threadIdx.y];
}

void verifyTranspose(const std::vector<int>& input, const std::vector<int>&
output, int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (input[i * n + j] != output[j * n + i])
            {
                printf("Ошибка: транспонирование неверно на элементе [%d]
[%d]\n", i, j);
                return;
            }
        }
    }
}

int main()
{
    const int   N       = 256;
    size_t      bytes   = N * N * sizeof(int);

    std::vector<int> h_inMatrix(N * N);
    std::vector<int> h_outMatrix(N * N, 0);

    int* d_inMatrix;
    int* d_outMatrix;

    cudaMalloc(&d_inMatrix, bytes);
    cudaMalloc(&d_outMatrix, bytes);

    dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
    dim3 blocks((N + BLOCK_SIZE - 1) / BLOCK_SIZE, (N + BLOCK_SIZE - 1) /
BLOCK_SIZE);

    for(int i = 0; i < RUN_COUNT; ++i)
    {
        srand(time(0));
        for (int i = 0; i < N * N; i++) h_inMatrix[i] = rand() % 1000 + 1;
        cudaMemcpy(d_inMatrix, h_inMatrix.data(), bytes,
cudaMemcpyHostToDevice);

        {
            CudaTimer t(true);
```

```
            transpose<<<blocks, threads>>>(d_inMatrix, d_outMatrix, N);
            cudaDeviceSynchronize();
        }

        cudaMemcpy(h_outMatrix.data(), d_outMatrix, bytes,
cudaMemcpyDeviceToHost);
        verifyTranspose(h_inMatrix, h_outMatrix, N);
    }

    printf("\nСреднее время выполнения non-shared: %.5f мс\n",
CudaTimer::avgElapsedTime);

    CudaTimer::avgElapsedTime = 0;

    for(int i = 0; i < RUN_COUNT; ++i)
    {
        srand(time(0));
        for (int i = 0; i < N * N; i++) h_inMatrix[i] = rand() % 1000 + 1;
        cudaMemcpy(d_inMatrix, h_inMatrix.data(), bytes,
cudaMemcpyHostToDevice);

        {
            CudaTimer t(true);

            transpose_shared<<<blocks, threads>>>(d_inMatrix, d_outMatrix,
N);
            cudaDeviceSynchronize();
        }

        cudaMemcpy(h_outMatrix.data(), d_outMatrix, bytes,
cudaMemcpyDeviceToHost);
        verifyTranspose(h_inMatrix, h_outMatrix, N);
    }

    printf("\nСреднее время выполнения shared: %.5f мс\n",
CudaTimer::avgElapsedTime);

    cudaFree(d_inMatrix);
    cudaFree(d_outMatrix);

    return 0;
}
```

Результаты

```
Среднее время выполнения non-shared: 0.02941 мс
Среднее время выполнения shared: 0.01047 мс
```

Ответы на контрольные вопросы

1. Какие типы памяти поддерживаются в CUDA?
Регистровая, локальная, разделяемая, глобальная, константная и текстурная.

2. В чем заключается эффективность работы с глобальной и разделяемой памятью в CUDA?

- Глобальная память: эффективна при выравненном (coalesced) доступе.
- Разделяемая память: имеет низкую латентность, доступна всем нитям блока.

3. Какие паттерны доступа к глобальной памяти существуют?

- Coalesced (упорядоченный): обеспечивает объединение запросов в одну транзакцию.
- Non-coalesced (неупорядоченный): снижает пропускную способность.

4. Какие паттерны доступа к разделяемой памяти существуют?

- Без конфликтов: равномерный доступ к разным банкам.
- С конфликтами: несколько нитей обращаются к одному банку.

5. В чем особенность схемы расположения банков в разделяемой памяти?
Разделяемая память разбита на банки. Конфликты возникают, если нити обращаются к одному и тому же банку.

6. Каким образом можно определить объем разделяемой памяти?
Через свойства устройства `cudaDeviceProp.sharedMemPerBlock`.

7. Как определить номер банка памяти?
Используя формулу:
(\text{Номер банка} = (\text{адрес в байтах} / 4) % \text{количество банков}).

8. Основные принципы оптимизации работы с памятью в CUDA:

- Упорядоченный доступ к глобальной памяти.
- Максимальное использование разделяемой памяти.
- Минимизация конфликтов банков.

9. Общий шаблон решения задач в CUDA:

1. Разделение задачи на блоки и нити.
2. Загрузка данных в разделяемую память.
3. Выполнение вычислений.
4. Сохранение результата в глобальную память.
5. Учет ограничений по памяти и параллелизму.