

## **Лабораторная работа 2**

### **Задание**

1. Прочитайте главу из теоретического материала "Разделяемая память" и ответьте на контрольные вопросы (ответы на контрольные вопросы не нужно включать в отчёт по лабораторной работе).

2. Оптимизируйте алгоритмы, реализованные в лабораторной работе №1 при помощи разделяемой памяти.

3. Постройте графики зависимости времени выполнения алгоритма от размера матрицы и вектора (Размеры матрицы 1000x500, 1000x1000, 1500x1000, 2000x1000, 2000x1500, 2500x1500, 2500x2000).

4. Проанализируйте, реализованные алгоритмы при помощи утилиты nvprof на эффективность доступа к глобальной памяти.

### **Оборудование**

GPU NVIDIA T4\*2

## Анализ

Time for AxV with size (1000,500): 0.266208 ms  
Time for VxA with size (1000,500): 0.085472 ms  
Time for AxV with size (1000,1000): 0.49104 ms  
Time for VxA with size (1000,1000): 0.094432 ms  
Time for AxV with size (1500,1000): 0.49344 ms  
Time for VxA with size (1500,1000): 0.147072 ms  
Time for AxV with size (2000,1000): 0.490016 ms  
Time for VxA with size (2000,1000): 0.187616 ms  
Time for AxV with size (2000,1500): 0.769984 ms  
Time for VxA with size (2000,1500): 0.191808 ms  
Time for AxV with size (2500,1500): 0.769408 ms  
Time for VxA with size (2500,1500): 0.23536 ms  
Time for AxV with size (2500,2000): 0.96064 ms  
Time for VxA with size (2500,2000): 0.225568 ms  
==136== Profiling application: ./output\_program2  
==136==

Profiling result:

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	62.44%	24.472ms	14	1.7480ms	1.4400us	8.9861ms	[CUDA memcpy HtoD]
	13.90%	4.1259ms	7	589.41us	240.12us	946.32us	matrixVectorMultKernel(float*, float*, float*, int, int, unsigned long)
	3.66%	1.0853ms	7	155.05us	71.647us	224.22us	vectorMatrixMultKernel(float*, float*, float*, int, int, unsigned long)
API calls:	59.79%	155.38ms	7	22.197ms	147.53us	154.43ms	cudaMallocPitch
	13.62%	26.529ms	7	3.7898ms	1.0195ms	9.4064ms	cudaMemcpy2D
	2.70%	5.2521ms	14	375.15us	72.530us	950.00us	cudaEventSynchronize
	2.22%	4.3311ms	21	206.24us	4.8600us	894.55us	cudaFree
	0.44%	854.22us	2	427.11us	342.40us	511.83us	cuDeviceTotalMem
	0.44%	850.90us	14	60.778us	3.6870us	158.53us	cudaMalloc
	0.31%	607.11us	202	3.0050us	187ns	143.90us	cuDeviceGetAttribute
	0.26%	503.59us	7	71.941us	41.745us	93.707us	cudaMemcpy
	0.09%	184.47us	14	13.176us	10.028us	23.647us	cudaLaunchKernel
	0.05%	91.680us	28	3.2740us	2.3370us	7.8010us	cudaEventRecord
	0.04%	68.951us	2	34.475us	25.076us	43.875us	cuDeviceGetName
	0.01%	24.623us	14	1.7580us	1.4570us	2.2590us	cudaEventElapsedTime
	0.01%	24.348us	14	1.7390us	659ns	8.3800us	cudaEventCreate
	0.01%	17.928us	14	1.2800us	629ns	2.7720us	cudaEventDestroy
	0.01%	10.064us	2	5.0320us	2.0760us	7.9880us	cuDeviceGetPCIBusId
	0.00%	2.9710us	4	742ns	332ns	1.6960us	cuDeviceGet
	0.00%	1.8630us	3	621ns	418ns	1.0260us	cuDeviceGetCount
	0.00%	547ns	2	273ns	250ns	297ns	cuDeviceGetUuid



## Вывод

Проанализировав результаты выполнения нашего приложения, можно заметить впечатляющую эффективность GPU в различных операциях умножения матрицы и вектора. Особенно примечательно, что при увеличении размера матрицы и вектора, время выполнения операций  $A \times V$  и  $V \times A$  растёт нелинейно, что указывает на оптимизированную работу с памятью и вычислениями.

Также стоит отметить, что большая часть GPU-активности связана с операциями [CUDA memcpy HtoD], что говорит о быстром перемещении данных между хостом и устройством. Профилирование показало, что ключевые компоненты, такие как `matrixVectorMultKernel` и `vectorMatrixMultKernel`, работают очень эффективно, обеспечивая быстрое выполнение операций умножения.

## Программный код

```
%%writefile your_code2.cu
#include <iostream>
#include <cuda_runtime.h>
#include <cassert>
#include <vector>

const int THREADS_PER_BLOCK = 128; // Количество потоков в одном блоке

// Обработка ошибок CUDA
#define CHECK_CUDA_ERROR(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, const char *file, int line, bool abort=true) {
    if (code != cudaSuccess) {
        fprintf(stderr, "GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
        if (abort) exit(code);
    }
}

// Ядро для умножения матрицы на вектор с использованием shared memory
__global__ void matrixVectorMultKernel(float* A, float* V, float* C, int M, int N, size_t pitch) {
    __shared__ float shared_vector[THREADS_PER_BLOCK];
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0;

    // Разделение вектора на части и загрузка каждой части в shared memory
    for (int k = 0; k < (M + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK; ++k) {
        if (threadIdx.x + k * THREADS_PER_BLOCK < M) {
            shared_vector[threadIdx.x] = V[threadIdx.x + k * THREADS_PER_BLOCK];
        }
        __syncthreads();

        // Умножение строки матрицы на часть вектора
        if (i < N) {
            float* row = (float*)((char*)A + i * pitch);
            for (int j = 0; j < THREADS_PER_BLOCK && j + k * THREADS_PER_BLOCK < M; ++j) {
                sum += row[j + k * THREADS_PER_BLOCK] * shared_vector[j];
            }
            __syncthreads();
        }

        if (i < N) {
            C[i] = sum;
        }
    }
}

// Ядро для умножения вектора на матрицу с использованием shared memory
__global__ void vectorMatrixMultKernel(float* V, float* A, float* C, int M, int N, size_t pitch) {
    __shared__ float shared_vector[THREADS_PER_BLOCK];
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0;

    // Разделение вектора на части и загрузка каждой части в shared memory
    for (int k = 0; k < (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK; ++k) {
        if (threadIdx.x + k * THREADS_PER_BLOCK < N) {
            shared_vector[threadIdx.x] = V[threadIdx.x + k * THREADS_PER_BLOCK];
        }
        __syncthreads();

        // Умножение части вектора на столбец матрицы
        if (i < M) {
            for (int j = 0; j < THREADS_PER_BLOCK && j + k * THREADS_PER_BLOCK < N; ++j) {
                float* row = (float*)((char*)A + (j + k * THREADS_PER_BLOCK) * pitch);
                sum += shared_vector[j] * row[i];
            }
        }
    }
}
```

```

    }
    }
    __syncthreads();
}

if (i < M) {
    C[i] = sum;
}
}

int main() {
    // Список размеров матриц
    std::vector<std::pair<int, int>> sizes = {
        {1000, 500}, {1000, 1000}, {1500, 1000},
        {2000, 1000}, {2000, 1500}, {2500, 1500},
        {2500, 2000}
    };

    for (auto& size : sizes) {
        int N = size.first; // Количество строк матрицы
        int M = size.second; // Количество столбцов матрицы

        float* h_A = new float[N * M];
        float* h_V = new float[M];
        float* h_C = new float[N];
        float *d_A, *d_V, *d_C;
        size_t pitch;
        // Выделение памяти на GPU с учетом выравнивания (pitch)
        CHECK_CUDA_ERROR(cudaMallocPitch(&d_A, &pitch, M * sizeof(float), N));
        CHECK_CUDA_ERROR(cudaMalloc(&d_V, M * sizeof(float)));
        CHECK_CUDA_ERROR(cudaMalloc(&d_C, N * sizeof(float)));

        // Копирование данных с хоста на устройство
        CHECK_CUDA_ERROR(cudaMemcpy2D(d_A, pitch, h_A, M * sizeof(float), M * sizeof(float), N,
        cudaMemcpyHostToDevice));
        CHECK_CUDA_ERROR(cudaMemcpy(d_V, h_V, M * sizeof(float), cudaMemcpyHostToDevice));

        // Измерение времени выполнения для AxV
        cudaEvent_t start, stop;
        CHECK_CUDA_ERROR(cudaEventCreate(&start));
        CHECK_CUDA_ERROR(cudaEventCreate(&stop));

        CHECK_CUDA_ERROR(cudaEventRecord(start));

        int blocks = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
        matrixVectorMultKernel<<<blocks, THREADS_PER_BLOCK>>>>(d_A, d_V, d_C, M, N, pitch);

        CHECK_CUDA_ERROR(cudaEventRecord(stop));
        CHECK_CUDA_ERROR(cudaEventSynchronize(stop));

        float milliseconds = 0;
        CHECK_CUDA_ERROR(cudaEventElapsedTime(&milliseconds, start, stop));
        std::cout << "Time for AxV with size (" << N << ", " << M << "): " << milliseconds << " ms" << std::endl;

        // Измерение времени выполнения для VxA
        CHECK_CUDA_ERROR(cudaEventRecord(start));

        vectorMatrixMultKernel<<<blocks, THREADS_PER_BLOCK>>>>(d_V, d_A, d_C, M, N, pitch);

        CHECK_CUDA_ERROR(cudaEventRecord(stop));
        CHECK_CUDA_ERROR(cudaEventSynchronize(stop));

        milliseconds = 0;
        CHECK_CUDA_ERROR(cudaEventElapsedTime(&milliseconds, start, stop));
        std::cout << "Time for VxA with size (" << N << ", " << M << "): " << milliseconds << " ms" << std::endl;
    }
}

```

```
// Освобождение памяти
delete[] h_A;
delete[] h_V;
delete[] h_C;
CHECK_CUDA_ERROR(cudaFree(d_A));
CHECK_CUDA_ERROR(cudaFree(d_V));
CHECK_CUDA_ERROR(cudaFree(d_C));

// Уничтожение событий CUDA
CHECK_CUDA_ERROR(cudaEventDestroy(start));
CHECK_CUDA_ERROR(cudaEventDestroy(stop));
}
return 0;
}
```