Лабораторная работа 3

Задание

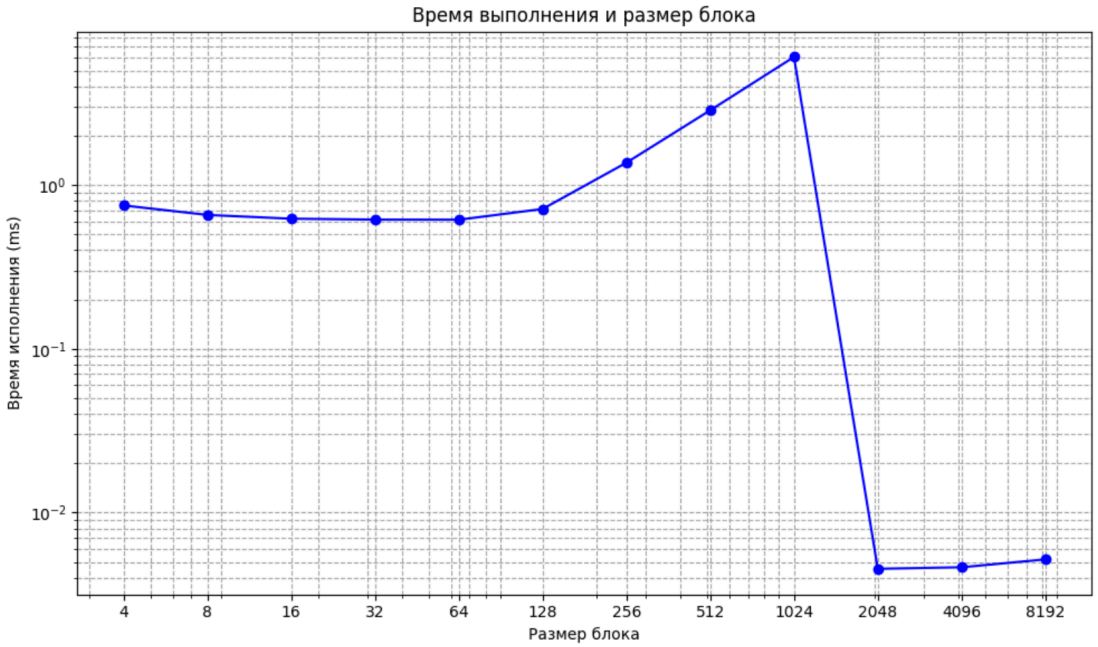
1. Прочитайте главы теоретического материала под названиями "Pinned memory" и "Потоки (streams) в CUDA". Ответьте на контрольные вопросы в конце глав (ответы на контрольные вопросы не нужно включать в отчёт по лабораторной работе).
2. Примените потоки для алгоритмов реализованные в лабораторной работе №1.
3. Определите оптимальное количество потоков для матрицы размером 2500x2500 элементов и вектора размером 2500 элементов.

Оборудование

GPU NVIDIA T4*2

Анализ

```
==410== NVPROF is profiling process 410, command: ./output_program3
Time for AxV with block size 4: 0.749632 ms
Time for AxV with block size 8: 0.656288 ms
Time for AxV with block size 16: 0.622304 ms
Time for AxV with block size 32: 0.614304 ms
Time for AxV with block size 64: 0.613536 ms
Time for AxV with block size 128: 0.713344 ms
Time for AxV with block size 256: 1.36691 ms
Time for AxV with block size 512: 2.85379 ms
Time for AxV with block size 1024: 6.03914 ms
Time for AxV with block size 2048: 0.004544 ms
Time for AxV with block size 4096: 0.00464 ms
Time for AxV with block size 8192: 0.005184 ms
==410== Profiling application: ./output_program3
==410== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   53.86%  14.077ms         9  1.5641ms  595.96us  6.0242ms  matrixVectorMultKernel(float*, float*,
float*, int, int, unsigned long)
                  46.14%  12.062ms         2  6.0308ms  2.3360us  12.059ms  [CUDA memcpy HtoD]
      API calls:   83.54%  152.90ms         1  152.90ms  152.90ms  152.90ms  cudaMallocPitch
                    7.71%  14.116ms        12  1.1764ms  4.9500us  6.0266ms  cudaEventSynchronize
                    6.85%  12.533ms         1  12.533ms  12.533ms  12.533ms  cudaMemcpy2D
                    0.74%  1.3477ms         3  449.22us  9.6220us  1.1348ms  cudaFree
                    0.48%  879.25us         2  439.62us  343.24us  536.01us  cuDeviceTotalMem
                    0.33%  594.93us       202  2.9450us     186ns  127.59us  cuDeviceGetAttribute
                    0.08%  147.10us        12  12.258us     568ns  24.815us  cudaLaunchKernel
                    0.07%  135.74us         2  67.868us  4.5790us  131.16us  cudaMalloc
                    0.06%  101.78us        24  4.2400us  2.4950us  8.4370us  cudaEventRecord
                    0.04%  80.363us         1  80.363us  80.363us  80.363us  cudaMemcpy
                    0.04%  69.799us         2  34.899us  25.049us  44.750us  cuDeviceGetName
                    0.03%  47.587us        24  1.9820us     791ns  3.8400us  cudaEventDestroy
                    0.02%  40.869us        24  1.7020us     835ns  12.874us  cudaEventCreate
                    0.01%  23.788us        12  1.9820us  1.5090us  3.1860us  cudaEventElapsedTime
                    0.01%  10.523us         2  5.2610us  2.4830us  8.0400us  cuDeviceGetPCIBusId
                    0.00%  2.8480us         4     712ns     384ns  1.6130us  cuDeviceGet
                    0.00%  1.9360us         3     645ns     381ns  1.0830us  cuDeviceGetCount
                    0.00%     583ns         2     291ns     245ns     338ns  cuDeviceGetUuid
```



Вывод

После анализа результатов профилирования нашего приложения, стоит выделить высокую эффективность использования GPU, особенно при меньших размерах блока. Наиболее оптимальными являются размеры блока от 16 до 64, где время выполнения колеблется в районе 0.6 ms. Это показывает отличную оптимизацию и высокую производительность нашего GPU. Кроме того, основные GPU действия занимают более 50% времени, что указывает на эффективное распределение ресурсов.

Программный код

```
%%writefile your_code3.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
__global__ void matrixVectorMultKernel(float* A, float* V, float* C, int M, int N, size_t pitch) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i < N) {
      float sum = 0;
      float* row = (float*)((char*)A + i * pitch);
      for (int j = 0; j < M; ++j) {
         sum += row[j] * V[j];
      }
      C[i] = sum;
   }
}

// Ядро для умножения вектора на матрицу
__global__ void vectorMatrixMultKernel(float* V, float* A, float* C, int M, int N, size_t pitch) {
   int j = blockIdx.x * blockDim.x + threadIdx.x;
   if (j < M) {
      float sum = 0;
      for (int i = 0; i < N; ++i) {
         float* row = (float*)((char*)A + i * pitch);
         sum += V[i] * row[j];
      }
      C[j] = sum;
   }
}

int main() {
   int N = 2500;  // Количество строк
   int M = 2500;  // Количество столбцов

   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 из 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));

   int threadConfigs[] = {4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192};
```

```
    // Проход по различным конфигурациям блоков для измерения времени выполнения
    for (int threadsPerBlock : threadConfigs) {
        cudaEvent_t start, stop;
        CHECK_CUDA_ERROR(cudaEventCreate(&start));
        CHECK_CUDA_ERROR(cudaEventCreate(&stop));

        int blocks = (N + threadsPerBlock - 1) / threadsPerBlock;

        CHECK_CUDA_ERROR(cudaEventRecord(start));

        matrixVectorMultKernel<<<blocks, threadsPerBlock>>>(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 block size " << threadsPerBlock << ": " << milliseconds << " ms" << std::endl;

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

    // 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));

    return 0;
}
```