Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Институт информатики и вычислительной техники

09.03.01 "Информатика и вычислительная техника"
профиль "Программное обеспечение средств
вычислительной техники и автоматизированных систем"

Кафедра прикладной математики и кибернетики

Расчётно графическое задание по дисциплине программирование графических процессоров

Тема: Сравнительный анализ производительности программ, реализующих алгоритмы линейной алгебры

Выполнил:

студент гр.ИП-ИП-012 _____/ Николаев А.Д. /
ФИО студента

Проверил

Старший преподаватель
кафедры ПМиК _____/ Нужнов А. В. /
ФИО преподавателя

Новосибирск 2023 г.

Задание

Сравнительный анализ производительности программ, реализующих алгоритмы линейной алгебры с использованием библиотек Thrust, cuBLAS и «сырого» и не «сырого» CUDA C кода с дополнительным сравнением производительности программ на основе интерфейса CUDA и на основе OpenGL - вычислительных шейдеров.

Введение

В программировании часто приходится встречаться с тем, что необходимо использовать операции линейной алгебры: начиная от теории графов для нахождения минимального пути, заканчивая матрицами преобразования для отображения графических объектов. Данные операции имеют высокую сложность алгоритмов (для по элементных операций над матрицами и транспонирования сложность квадратичная, а для перемножения матриц - кубическая), поэтому для их быстрого выполнения эффективнее будет работать не на CPU, а на устройстве, которое может выполнять параллельно миллионы операций в секунду – GPU.

В рамках данного расчётно-графического задания будут использованы библиотека CUDA для прямого программирования на GPU фирмы Nvidia на языке C, библиотека thrust, предназначенная для реализации высоко производительных параллельных алгоритмов, а также библиотека cuBlas, предназначенная для выполнения алгоритмов линейной алгебры над матрицами, включая их перемножение суммирование и транспонирования.

Также будет использоваться библиотека OpenGL, с помощью которой можно выполнять операции на графических интерфейсах с помощью вычислительных шейдеров. В данной работе вычислительные шейдеры OpenGL будут использоваться только для транспонирования матрицы.

После подсчёта времени выполнения транспонирования матриц разных размеров можно будет сделать выводы о том, какие библиотеки будет эффективнее всего использовать для реализации транспонирования матриц.

Листинг

Код программы транспонирование матрицы с использованием Thrust, cuBLAS, «сырого» CUDA C кода и не «сырого»:

```
#include<cuda.h>
#include<stdio.h>
#include<stdlib.h>
#include<iostream>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sequence.h>
#include <thrust/inner_product.h>
#include <thrust/gather.h>
#include "cublas_v2.h"

#define START_SIZE (1<<5)
#define FINAL_SIZE (1<<15)
#define ITERATIONS (15-5+1)
#define COUNT 10
#define DATA_COUNT (ITERATIONS)
#define THREADS_PER_BLOCK (1<<5)
#define SH_DIM (THREADS_PER_BLOCK)

using namespace std;

struct init_functor
{
     float h;
     init_functor(float _h) :h(_h) {}
     __device__ float operator()(float x)
     {
          return h * x;
     }
};

__global__ void gTranspose(float* da, float* db, int N, int M)
{
     int m = threadIdx.x + blockIdx.x * blockDim.x;
     int n = threadIdx.y + blockIdx.y * blockDim.y;

     db[n + m * N] = da[m + n * M];
}

__global__ void gTransposeOpt(float* matr, float* matr_t, long
long N, long long M)
{
     __shared__ float buffer[SH_DIM][SH_DIM + 1];
     int m = threadIdx.x + blockIdx.x * blockDim.x;
     int n = threadIdx.y + blockIdx.y * blockDim.y;

     buffer[threadIdx.y][threadIdx.x] = matr[m + n * M];
     __syncthreads();
```

```
    m = threadIdx.x + blockIdx.y * blockDim.x;
    n = threadIdx.y + blockIdx.x * blockDim.y;
    matr_t[m + n * N] = buffer[threadIdx.x][threadIdx.y];
}


__global__ void gInit(float* d, int s)
{
    int j = threadIdx.x + blockIdx.x * blockDim.x;
    int i = threadIdx.y + blockIdx.y * blockDim.y;
    int J = blockDim.x * gridDim.x;

    d[j + i * J] = s * (float)((j + i * J) * 1.0E-3) + (1 - s) *
1.0f;
}

void printMatr(float* d, long long N, long long M)
{
    for (long long i = 0; i < N; i++)
    {
        for (long long j = 0; j < M; j++) printf("%4.2f ", d[j
+ i * M]);
        printf("\n");
    }
    printf("\n");
}

void printMatr(thrust::host_vector<float> d, long long N, long
long M)
{
    for (long long i = 0; i < N; i++)
    {
        for (long long j = 0; j < M; j++) printf("%4.2f ", d[j
+ i * M]);
        printf("\n");
    }
    printf("\n");
}

void printResults(float time_cuda[], float time_cuda_opt[],
float time_thrust[], float time_cublas[])
{
    FILE* fp = fopen("Out2.csv", "w");
    long long N = START_SIZE;
    long long M = START_SIZE;
    long long index;
    fprintf(fp, "N;M;Threads per block;CUDA C;CUDA C
opt;Thrust;CuBlas;\n");
    for (long long i = 0; i < ITERATIONS; i++)
    {
        index = i;
        fprintf(fp, "%lli;%lli;", N, M);
```

```
            fprintf(fp, "%i;", THREADS_PER_BLOCK);
            fprintf(fp, "%f;", time_cuda[index] / COUNT);
            fprintf(fp, "%f;", time_cuda_opt[index] / COUNT);
            fprintf(fp, "%f;", time_thrust[index] / COUNT);
            fprintf(fp, "%f;", time_cublas[index] / COUNT);
            fprintf(fp, "\n");
            N <<= 1;
        }
        fclose(fp);
}

int main()
{
        float time_cublas[DATA_COUNT];
        float time_cuda_opt[DATA_COUNT];
        float time_thrust[DATA_COUNT];
        float time_cuda[DATA_COUNT];
        long long N = START_SIZE;
        long long M = START_SIZE;
        float* ha, * hb, * da, * db;

        init_functor I(1.0E-3);
        thrust::device_vector<long long> dmap;
        thrust::device_vector<float> dVa;
        thrust::device_vector<float> dVb;
        thrust::host_vector<float> hVb;
        long long* map;

        const float alpha = 1.0;
        const float beta = 0.0;

        cudaEvent_t time_start, time_stop;
        cudaEventCreate(&time_start);
        cudaEventCreate(&time_stop);

        for (long long i = 0; i < ITERATIONS; i++)
        {
            cudaMalloc((void**)&da, N * M * sizeof(float));
            cudaMalloc((void**)&db, M * N * sizeof(float));
            gInit << <dim3(M / THREADS_PER_BLOCK, N /
THREADS_PER_BLOCK), dim3(THREADS_PER_BLOCK, THREADS_PER_BLOCK)
>> > (da, 1);
            cudaDeviceSynchronize();
            cudaEventRecord(time_start, 0);
            for (long long k = 0; k < COUNT; k++)
            {
                gTranspose << <dim3(M / THREADS_PER_BLOCK, N /
THREADS_PER_BLOCK), dim3(THREADS_PER_BLOCK, THREADS_PER_BLOCK)
>> > (da, db, N, M);
                cudaDeviceSynchronize();
            }
            cudaEventRecord(time_stop, 0);
            cudaEventSynchronize(time_stop);
```

```
          cudaEventElapsedTime(&(time_cuda[i]), time_start,
time_stop);
          if (i == 2)
          {
               ha = (float*)malloc(N * M * sizeof(float));
               hb = (float*)malloc(M * N * sizeof(float));
               cudaMemcpy(ha, da, N * M * sizeof(float), cudaMem-
cpyDeviceToHost);
               cudaMemcpy(hb, db, M * N * sizeof(float), cudaMem-
cpyDeviceToHost);
               printMatr(ha, N, M);
               printMatr(hb, M, N);
               free(ha);
               free(hb);
          }
          cudaFree(db);

          cudaMalloc((void**)&db, M * N * sizeof(float));
          cudaDeviceSynchronize();
          cudaEventRecord(time_start, 0);
          for (long long k = 0; k < COUNT; k++)
          {
               gTransposeOpt << <dim3(M / THREADS_PER_BLOCK, N /
THREADS_PER_BLOCK), dim3(THREADS_PER_BLOCK, THREADS_PER_BLOCK)
>> > (da, db, N, M);
               cudaDeviceSynchronize();
          }
          cudaEventRecord(time_stop, 0);
          cudaEventSynchronize(time_stop);
          cudaEventElapsedTime(&(time_cuda_opt[i]), time_start,
time_stop);
          if (i == 2)
          {
               hb = (float*)malloc(M * N * sizeof(float));
               cudaMemcpy(hb, db, M * N * sizeof(float), cudaMem-
cpyDeviceToHost);
               printMatr(hb, M, N);
               free(hb);
          }
          cudaFree(db);

          dVa = thrust::device_vector<float>(N * M);
          dVb = thrust::device_vector<float>(M * N);
          hVb = thrust::host_vector<float>(N * M);
          thrust::sequence(thrust::device, dVa.begin(),
dVa.end());
          thrust::transform(dVa.begin(), dVa.end(), dVa.begin(),
I);
          map = (long long*)malloc(M * N * sizeof(long long));
          for (long long i0 = 0; i0 < M * N; i0++) map[i0] = (i0
% N) * M + (i0 / N);
          dmap = thrust::device_vector<long long>(map, map + M *
N);
```

```
          cudaEventRecord(time_start, 0);
          for (long long k = 0; k < COUNT; k++)
thrust::gather(dmap.begin(), dmap.end(), dVa.begin(),
dVb.begin());
          cudaEventRecord(time_stop, 0);
          cudaEventSynchronize(time_stop);
          cudaEventElapsedTime(&(time_thrust[i]), time_start,
time_stop);
          hVb = dVb;
          if (i == 2)
          {
               printMatr(hVb, M, N);
          }
          free(map);


          cudaMalloc((void**)&db, M * N * sizeof(float));
          cublasHandle_t cublas_handle;
          cublasCreate(&cublas_handle);
          cudaEventRecord(time_start, 0);
          for (long long k = 0; k < COUNT; k++)
          {
               cublasSgeam(cublas_handle, CUBLAS_OP_T,
CUBLAS_OP_T, N, M, &alpha, da, M, &beta, da, M, db, N);
          }
          cudaEventRecord(time_stop, 0);
          cudaEventSynchronize(time_stop);
          cudaEventElapsedTime(&(time_cublas[i]), time_start,
time_stop);
          cublasDestroy(cublas_handle);
          if (i == 2)
          {
               hb = (float*)malloc(N * M * sizeof(float));
               cublasGetMatrix(M, N, sizeof(float), db, M, hb,
M);
               printMatr(hb, M, N);
               free(hb);
          }
          cudaFree(da);
          cudaFree(db);
          N <<= 1;
     }
     printResults(time_cuda, time_cuda_opt, time_thrust,
time_cublas);
     cudaEventDestroy(time_start);
     cudaEventDestroy(time_stop);
}
```

Код программы транспонирование матрицы с использованием OpenGL:

1) Файл main.cpp:

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <stdio.h>
#include <string>
#include <stdlib.h>
#include <fstream>

#define START_SIZE (1<<5)
#define FINAL_SIZE (1<<15)
#define ITERATIONS (15-5+1)
#define DATA_COUNT (ITERATIONS)
#define COUNT 10
void initGL();
GLuint* bufferID;
void initBuffers(GLuint*& bufferID);
void transformBuffers(GLuint* bufferID, long long ind);
void outputBuffers(GLuint* bufferID);

const unsigned int window_width = 512;
const unsigned int window_height = 512;
float* time_openGL;

unsigned int N = START_SIZE;
unsigned int M = START_SIZE;


void initGL() {
     GLFWwindow* window;
     if (!glfwInit()) {
          fprintf(stderr, "Failed to initialize GLFW\n");
          getchar();
          return;
     }
     glfwWindowHint(GLFW_VISIBLE, 0);
     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
     glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
     glfwWindowHint(GLFW_OPENGL_PROFILE,
          GLFW_OPENGL_COMPAT_PROFILE);
     window = glfwCreateWindow(window_width, window_height,
          "Random Life", NULL, NULL);
     if (window == NULL) {
          fprintf(stderr, "Failed to open GLFW window. \n");
          getchar();
          glfwTerminate();
          return;
     }
     glfwMakeContextCurrent(window);
     glewInit();
     return;
```

```
}



int main()
{
	initGL();
	time_openGL=new float[DATA_COUNT];
	for (long long i=0;i<ITERATIONS;i++)
	{
		time_openGL[i] = 0.0f;
		bufferID = (GLuint*)calloc(2, sizeof(GLuint));
		initBuffers(bufferID);
		for (int k = 0; k < COUNT; k++) transformBuffers(buff-
erID, i);
		if (i == 2) outputBuffers(bufferID);
		glDeleteBuffers(3, bufferID);
		free(bufferID);
		N *= 2;
	}
	glfwTerminate();
	std::ofstream fcout("OutOpenGL.csv");
	for (long long i = 0; i < ITERATIONS; i++) fcout <<
(time_openGL[i] / COUNT) << ";\n";
	return 0;
}
```

2) Файл csh_common.cpp:

```
#include<GL/glew.h>
#include <stdio.h>
#include <string>
#include <stdlib.h>
#include <chrono>

extern float *time_openGL;
extern unsigned int N;
extern unsigned int M;
GLuint genInitProg();
GLuint genTransformProg();

void initBuffers(GLuint*& bufferID) {
	glGenBuffers(2, bufferID);
	glBindBuffer(GL_SHADER_STORAGE_BUFFER, bufferID[0]);
	glBufferData(GL_SHADER_STORAGE_BUFFER, N*M * sizeof(float),
0,
		GL_DYNAMIC_DRAW);
	glBindBuffer(GL_SHADER_STORAGE_BUFFER, bufferID[1]);
	glBufferData(GL_SHADER_STORAGE_BUFFER, M*N * sizeof(float),
0,
		GL_DYNAMIC_DRAW);

	glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, bufferID[0]);
```

```
	glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, bufferID[1]);

	GLuint csInitID = genInitProg();
	glUseProgram(csInitID);
	glDispatchCompute(M/32, N/32, 1);
	glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT |
		GL_BUFFER_UPDATE_BARRIER_BIT);
	glDeleteProgram(csInitID);
}

GLuint genInitProg() {
	GLuint progHandle = glCreateProgram();
	GLuint cs = glCreateShader(GL_COMPUTE_SHADER);
	const char* cpSrc[] = {
	"#version 430\n",
	"layout (local_size_x = 32, local_size_y = 32, local_size_z
= 1) in; \
layout(std430, binding = 0) buffer BufferA{float A[];};\
void main() {\
uint m = gl_GlobalInvocationID.x;\
uint n = gl_GlobalInvocationID.y;\
uint M = gl_NumWorkGroups.x*gl_WorkGroupSize.x;\
A[m+n*M]=0.001f*(m+n*M);\
}"
	};
	glShaderSource(cs, 2, cpSrc, NULL);
	glCompileShader(cs);
	int rvalue;
	glGetShaderiv(cs, GL_COMPILE_STATUS, &rvalue);
	if (!rvalue) {
		fprintf(stderr, "Error in compiling vp\n");
		exit(30);
	}
	glAttachShader(progHandle, cs);
	glLinkProgram(progHandle);
	glGetProgramiv(progHandle, GL_LINK_STATUS, &rvalue);
	if (!rvalue) {
		fprintf(stderr, "Error in linking sp\n");
		exit(32);
	}
	return progHandle;
}


void transformBuffers(GLuint* bufferID, long long ind) {
	uint64_t time_start;
	uint64_t time_stop;
	glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, bufferID[0]);
	glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, bufferID[1]);
	GLuint csTransformID = genTransformProg();
	glUseProgram(csTransformID);

	GLuint nID = glGetUniformLocation(csTransformID, "N");
```

```
     GLuint mID = glGetUniformLocation(csTransformID, "M");
     glUniform1ui(nID, N);
     glUniform1ui(mID, M);

     time_start = std::chrono::duration_cast<std::chrono::nano-
seconds>(std::chrono::sys-
tem_clock::now().time_since_epoch()).count();
     glDispatchCompute(M / 32, N / 32, 1);
     time_stop = std::chrono::duration_cast<std::chrono::nanosec-
onds>(std::chrono::sys-
tem_clock::now().time_since_epoch()).count();
     time_openGL[ind] += (float)(time_stop - time_start) /
1000000;
     glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT |
          GL_BUFFER_UPDATE_BARRIER_BIT);
     glDeleteProgram(csTransformID);
}

GLuint genTransformProg() {
     GLuint progHandle = glCreateProgram();
     GLuint cs = glCreateShader(GL_COMPUTE_SHADER);
     const char* cpSrc[] = {
     "#version 430\n",
     "layout (local_size_x = 32, local_size_y = 32, local_size_z
= 1) in; \
uniform uint N;\
uniform uint M;\
layout(std430, binding = 0) buffer BufferA{float A[];};\
layout(std430, binding = 1) buffer BufferB{float B[];};\
\
void main() {\
uint m = gl_GlobalInvocationID.x;\
uint n = gl_GlobalInvocationID.y;\
B[n+m*N]=A[m+n*M];\
}"
     };
     glShaderSource(cs, 2, cpSrc, NULL);
     glCompileShader(cs);
     int rvalue;
     glGetShaderiv(cs, GL_COMPILE_STATUS, &rvalue);
     if (!rvalue) {
          fprintf(stderr, "Error in compiling 3vp\n");
          exit(30);
     }
     glAttachShader(progHandle, cs);
     glLinkProgram(progHandle);
     glGetProgramiv(progHandle, GL_LINK_STATUS, &rvalue);
     if (!rvalue) {
          fprintf(stderr, "Error in linking sp\n");
          exit(32);
     }
     return progHandle;
}
```

```
void outputBuffers(GLuint* bufferID)
{
     glBindBuffer(GL_SHADER_STORAGE_BUFFER, bufferID[1]);
     float* data = (float*)glMapBuffer(GL_SHADER_STORAGE_BUFFER,
          GL_READ_ONLY);
     float* hdata = (float*)calloc(N*M, sizeof(float));
     memcpy(&hdata[0], data, sizeof(float) * N*M);
     glUnmapBuffer(GL_SHADER_STORAGE_BUFFER);
     for (int i = 0; i < M; i++)
     {
          for (int j = 0; j < N; j++) fprintf(stdout, "%4.2f ",
hdata[j + i * N]);
          printf("\n");
     }
}
```

Работа программы

Рис. 1. Результат работы программы на с использованием Thrust, cuBLAS, «сырого» CUDA C кода и не «сырого»

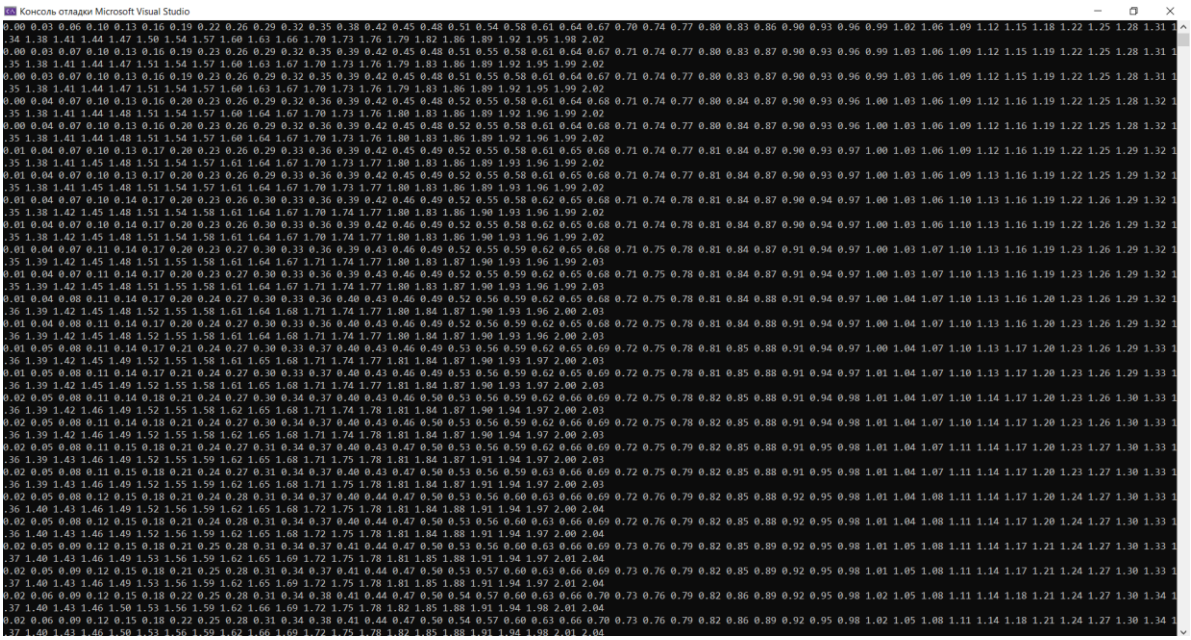

Рис 2. Результат работы программы с использованием OpenGL

N	M	Threads p	CUDA C	CUDA C op	Thrust	CuBlas	OpenGL
32	32	32	0.007264	0.006042	0.003344	0.004397	0.01136
64	32	32	0.007094	0.00599	0.003338	0.003798	0.00538
128	32	32	0.007213	0.006496	0.003382	0.003872	0.00478
256	32	32	0.008672	0.00632	0.003482	0.003827	0.00601
512	32	32	0.011741	0.007494	0.004333	0.004019	0.0041
1024	32	32	0.016682	0.009411	0.006912	0.004832	0.00602
2048	32	32	0.02615	0.01281	0.011562	0.007386	0.0041
4096	32	32	0.046371	0.020317	0.026662	0.011395	0.0051
8192	32	32	0.087021	0.038413	0.079795	0.026742	0.0074
16384	32	32	0.16697	0.071616	0.23096	0.049114	0.01117
32768	32	32	0.326861	0.136522	0.471053	0.09393	0.00728

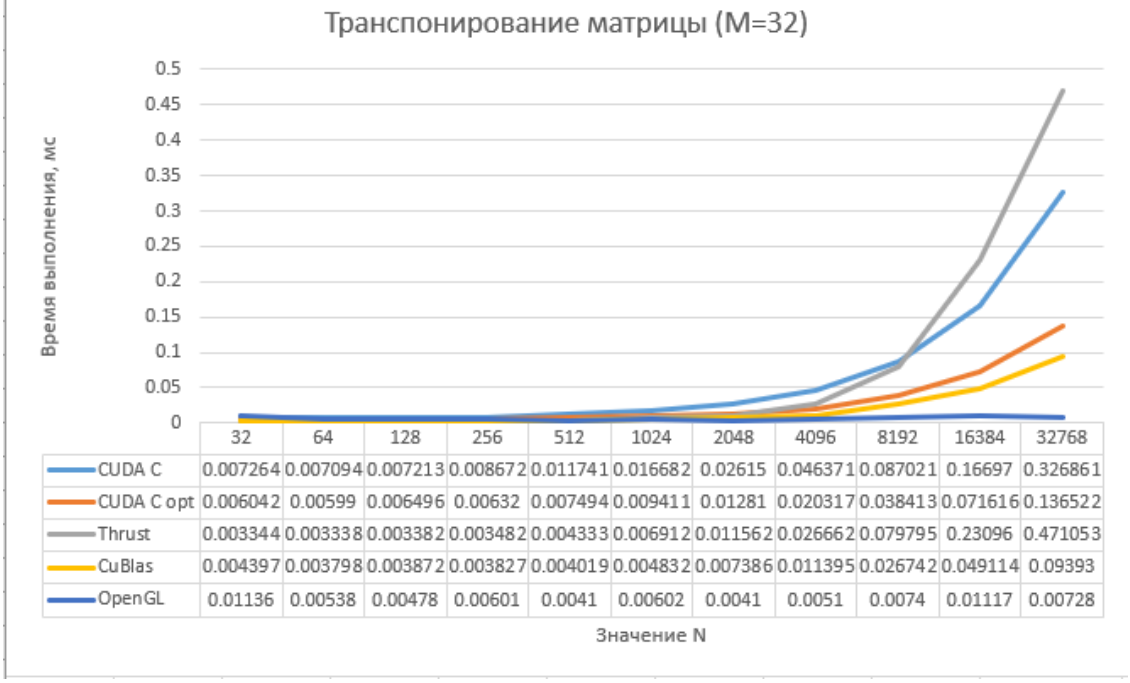

Рис 3. График результата работы программы

Результат работы

В результате выполнения программ, а также из графика, представленного на рисунке 3, можно сделать вывод, что:

1. Библиотека thrust выполняет транспонирование для маленьких матриц быстрее других библиотек, но чем больше матрица, тем не эффективнее её использование.
2. Неоптимизированный CUDA C код для небольших матриц немного уступает библиотеки thrust, но для больших – становиться куда эффективнее (как минимум раза в 2).
3. Оптимизированный CUDA C код и библиотека cuBlas имеют примерно одинаковую трудоёмкость и раза в 2-3 эффективнее неоптимизированного CUDA C кода для больших матриц.
4. OpenGL работает с матрицами размером до 32768x32 практически с линейной трудоёмкости и работает куда эффективнее всех остальных библиотек. (Но и запрограммировать программу сложнее всего)

Подводя итог можно сделать вывод, что эффективнее всего пользоваться библиотекой OpenGL с точки зрения трудоёмкости, а также библиотекой cuBlas с точки зрения отношения трудоёмкости к сложности программирования.