Документация по работе с математической библиотекой CUDA

Руководитель проекта: Бокарев Станислав

Разработчики: Чуваков Никита, Назарова Анастасия

2016 г.

Оглавление

<u>Функции приложения</u>

Математическая библиотека предоставляет собой параллельные алгоритмы, которые выполняются на видеокарте Nvidia, что позволяет при больших входных данных ускорить процесс расчета. В ней реализованы следующие математические операции.

1. ***Нахождение приближенного решения системы обыкновенных дифференциальных уравнений:***
   - *метод Эйлера;*
   - *метода Рунге-Кутты 2;*
   - *метод Рунге-Кутты 4;*
2. ***Численное интегрирование определенного интеграла:***
   - *метод Симпсона;*
   - *метод Симпсона 3/8;*
   - *метод Гаусса;*
3. ***Работа с матрицами:***
   - *транспонирование матрицы;*
   - *умножение матрицы на матрицу;*
   - *умножение матрицы на вектор.*

Расчетные формулы:

<u>Нахождение приближенного решения системы обыкновенных дифференциальных уравнений:</u>

Пусть поведение некоторого объекта описывается следующими уравнениями:

$$\frac{dx_i}{dt} = F_i(t, x_1, x_2, \dots, x_n),\ i=1,\ 2,\ \dots,\ n,$$

где t - время; x_1, x_2, …, x_n - параметры, определяющие состояние объекта; F_i- заданные функции своих аргументов (правые части).

К уравнению присоединяются начальные условия:

$$x_i\big|_{t=t^0} = x_i^0,\ i = 1,\ 2,\ \dots,\ n,$$

где t^0 - начальное время; x_i^0, - начальные значения параметров. Задача Коши состоит в отыскании значений параметров x_i для любого будущего момента времени $t > t^0$.

Метод Эйлера

- простейший численный метод решения систем обыкновенных дифференциальных уравнений. Вводится шаг по времени τ. Время становится дискретным:

$$t^{k+1} = t^k + \tau.$$

Индекс вверху означает номер момента времени. Соответственно, x^k_i - это значение x_i; в момент времени t^k. Согласно методу Эйлера, интегрирование исходных дифференциальных уравнений сводится к следующему вычислительному процессу:

$$x_i^{k+1} = x_i^k + \tau \cdot F_i\left(t^k, x_1^k, x_2^k, \ldots, x_n^k\right),\ i=1,\ 2,\ \ldots,\ n.$$

Метод Рунге-Кутты 2-го порядка (Усовершенствованный метод Эйлера).

Каждый шаг по времени т состоит из двух этапов. На первом этапе выполняется половина шага по методу Эйлера:

$$x_i^{k+1/2} = x_i^k + \frac{\tau}{2} F_i\left(t^k, x_1^k, x_2^k, \ldots, x_n^k\right);$$

$$t^{k+1/2} = t^k + \frac{\tau}{2},\quad i=1,\ 2,\ \ldots,\ n.$$

После этого вычисляются значения производных для среднего значения отрезка τ:

$$F_i^{k+1/2} = F_i\left(t^{k+1/2}, x_1^{k+1/2}, \ldots, x_n^{k+1/2}\right).$$

На втором этапе из исходной точки выполняется целый шаг при этом берется правая часть для средней точки:

$$x_i^{k+1} = x_i^k + \tau \cdot F_i^{k+1/2};$$

$$t^{k+1} = t^k + \tau,\quad i=1,\ 2,\ \ldots,\ n.$$

Точность вычислений возрастает, т.к. значение производной в средней точке шага по времени лучше передает скорость изменения искомых параметров, чем значения производных, взятых на краю отрезка.

Метод Рунге-Кутты 4-го порядка

Вычисление нового значения проходит в четыре стадии:

$$1.\ \boldsymbol{R}_i^1 = \boldsymbol{\tau} * \boldsymbol{F}_i * \left(\boldsymbol{t}^k, \boldsymbol{x}_1{}^k,\ \ldots, \boldsymbol{x}_n{}^k\right);$$

$$2.\ \boldsymbol{R}_i^2 = \boldsymbol{\tau} * \boldsymbol{F}_i * \left(\boldsymbol{t}^k + \frac{\boldsymbol{\tau}}{2}, \boldsymbol{x}_1{}^k + \frac{1}{2}\boldsymbol{R}_1^1,\ \ldots, \boldsymbol{x}_n{}^k + \frac{1}{2}\boldsymbol{R}_n{}^1\right);$$

$$3.\ \boldsymbol{R}_i^3 = \boldsymbol{\tau} * \boldsymbol{F}_i * \left(\boldsymbol{t}^k + \frac{\boldsymbol{\tau}}{2}, \boldsymbol{x}_1{}^k + \frac{1}{2}\boldsymbol{R}_1^2,\ \ldots, \boldsymbol{x}_n{}^k + \frac{1}{2}\boldsymbol{R}_n{}^2\right);$$

$$4.\ \boldsymbol{R}_i^4 = \boldsymbol{\tau} * \boldsymbol{F}_i * \left(\boldsymbol{t}^k + \boldsymbol{\tau}, \boldsymbol{x}_1{}^k + \frac{1}{2}\boldsymbol{R}_1^3,\ \ldots, \boldsymbol{x}_n{}^k + \boldsymbol{R}_n{}^3\right);$$

$$\boldsymbol{x}_i^{k+1} = \boldsymbol{x}_i^k + \frac{1}{6}(\boldsymbol{R}_i^1 + 2\boldsymbol{R}_i^2 + 2\boldsymbol{R}_i^3 + \boldsymbol{R}_i^4)$$

$$\boldsymbol{i} = 1, 2, \ldots, \boldsymbol{n}$$

$$\boldsymbol{t}^{k+1} = \boldsymbol{t}^k + \boldsymbol{\tau}$$

<u>Численное интегрирование определенного интеграла:</u>

Дана функция f(x), необходимо найти $\int_a^b \boldsymbol{f(x)dx}$.

Метод Симпсона

Расчетная формула для данного метода выглядит следующим образом:

$$\int_a^b f(x)dx = \frac{h}{3}\left(f(a) + f(b) + 4 * \sum_{i=1}^{N/2} f(a + (2i-1)h) + 2 * \sum_{i=1}^{\frac{N}{2}-1} f(a + 2 * i * h)\right)$$

Условие применимости метода: N должно быть кратно двум.

Метод Симпсона имеет 4-й порядок точности. Этого достаточно для большинства инженерных расчетов (один из наиболее популярных методов).

Метод Симпсона 3/8

Существует более удобный вариант без ограничения на четность N – метод Симпсона 3/8.

Расчетная формула для данного метода выглядит следующим образом:

$$\int_a^b f(x)dx = h\left(\frac{3}{8}\left(f(a) + f(b)\right) + \frac{7}{6}\left(f(a+h) + f(b-h)\right) + \frac{23}{24}\left(f(a+2h) + f(b-2h)\right) + \sum_{i=4}^{N-2} f(a + (i-1) * h)\right)$$

Условие применимости метода: N≥6.

Данный метод также имеет 4-й порядок точности и проще реализуется, так как не надо по отдельности вычислять суммы четных и нечетных членов. Однако его погрешность несколько выше, чем у классического метода Симпсона.

Метод Гаусса

Расчетная формула для данного метода выглядит следующим образом:

$$\int_a^b f(x)dx = \frac{h}{2}\sum_{j=0}^{N-1}\sum_{i=1}^{p} c_i * f(x_i \frac{h}{2} + a + jh + \frac{h}{2})$$

Для реализации метода также понадобятся значения абсцисс (x_i) и коэффициентов (c_i) (приведены ниже в таблицах 1 и 2).

Таблица 1. Абсциссы метода Гаусса.

p	x_1	x_2	x_3	x_4
1	0			
2	-0.57735	0.57735		
3	-0.774597	0	0.774597	
4	-0.861135	-0.339981	0.339981	0.861136

Таблица 2. Коэффициенты метода Гаусса.

p	c_1	c_2	c_3	c_4
1	2			
2	1	1		
3	0.55556	0.888889	0.55556	
4	0.34785	0.65215	0.65215	0.34785

Стоит отметить, что данный метод имеет очевидный недостаток: если на интервале [a; b] функция f(x) имеет достаточно сложный вид, то вычисление интеграла не даст качественного результата. Поэтому вышеуказанный метод эффективен для несложных функций.

Работа с матрицами:

Матрица — ***математический объект***, записываемый в виде прямоугольной таблицы элементов ***кольца*** или ***поля*** (например, ***целых***, ***действительных*** или***комплексных*** чисел), которая представляет собой совокупность ***строк*** и ***столбцов***, на пересечении которых находятся её элементы. Количество строк и столбцов матрицы задает размер матрицы.

Транспонирование

Для каждой матрицы

$$A = (a_{i,j})_{i=1,\dots,m_{j=1,\dots,n}} = \begin{pmatrix} a_{1,1} \dots & a_{1,n} \\ & \vdots & \\ a_{m,1} \dots & a_{m,n} \end{pmatrix},$$

размера $n \times m$ можно построить матрицу

$$B = (b_{j,i})_{j=1,\dots,n_{i=1,\dots,m}} = \begin{pmatrix} b_{1,1} \dots & b_{1,m} \\ & \vdots & \\ b_{n,1} \dots & b_{n,m} \end{pmatrix},$$

размера $n \times m$, у которой $b_{j,i} = a_{i,j}$ для всех $i = 1, \dots, m$ и $j = 1, \dots, n$.

Такая матрица называется ***транспонированной матрицей*** для A и обозначается A^T.

При транспонировании строки (столбцы) матрицы A становятся столбцами (соответственно - строками) матрицы A^T.

Очевидно, $(A^T)^T = A$.

Умножение матрицы на матрицу

Умножение матриц (обозначение: AB, реже со знаком умножения $A \times B$) — есть операция вычисления матрицы C, каждый элемент которой равен сумме произведений элементов в соответствующей строке первого множителя и столбце второго.

$$c_{ij} = \sum_{k=1}^{n} a_{ik}\, b_{kj}.$$

Количество столбцов в матрице A должно совпадать с количеством строк в матрице B, иными словами, матрица A обязана быть **согласованной** с матрицей B. Если матрица A имеет размерность $m \times n$, B — $n \times k$, то размерность их произведения $AB = C$ есть $m \times k$.

Умножение матрицы на вектор

По обычным правилам матричного умножения осуществляется умножение на матрицу слева вектора-столбца, а также умножение вектора-строки на матрицу справа. Поскольку элементы вектора-столбца или вектора-строки можно записать (что обычно и делается), используя один, а не два индекса, это умножение можно записать так:

для вектора-столбца v (получая новый вектор-столбец Av):

$$(Av)_i = \sum_{k=1}^{n} a_{ik}\, v_k,$$

для вектора-строки s (получая новый вектор-строку sA):

$$(sA)_i = \sum_{k=1}^{n} s_k a_{ki}.$$

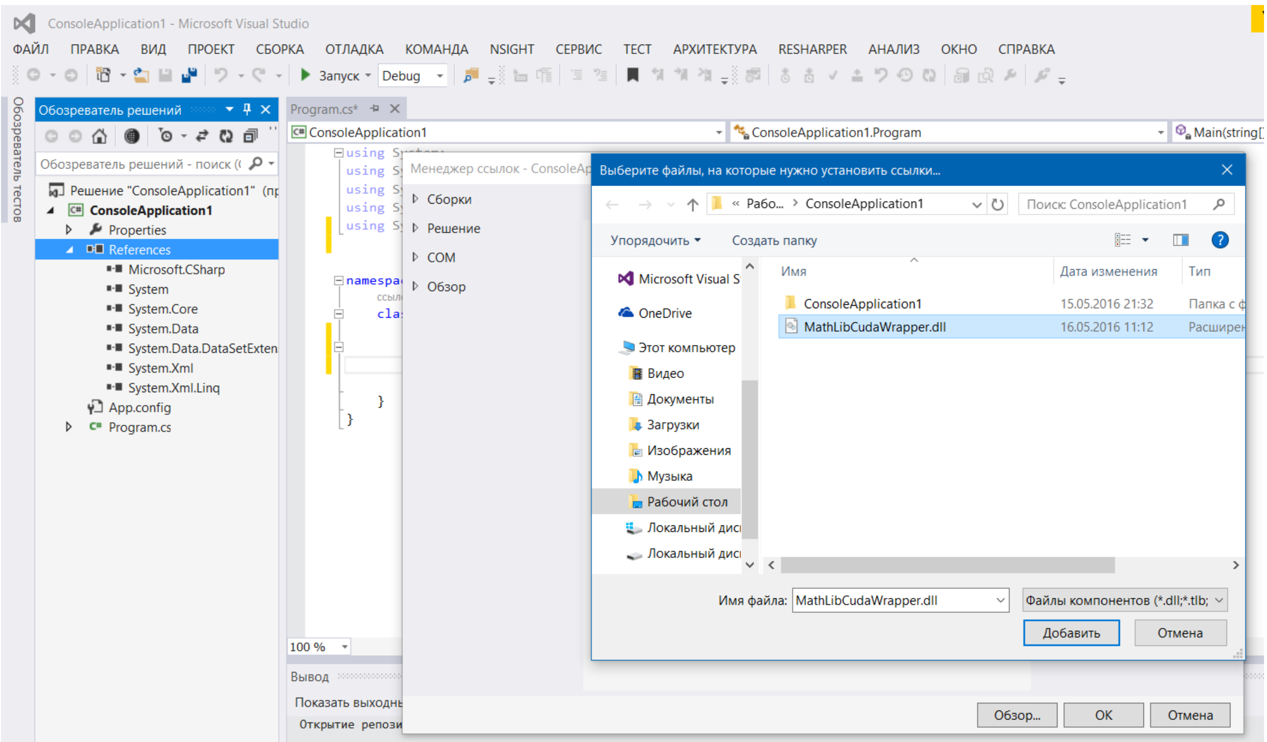
Вектор-строка, матрица и вектор-столбец могут быть умножены друг на друга, давая число (скаляр):

$$sAv = \sum_{k,i} s_k\, a_{ki} v_i.$$

Инструкция для пользователя

Для приложений на C#

1. Чтобы воспользоваться данной математической библиотекой помимо видеокарты Nvidia с поддержкой CUDA (все современные видеокарты поддерживают данную технологию) необходимо следующее программное обеспечение: MS Visual Studio 2013 и выше (на версиях младше стабильная работа приложения не гарантируется), также потребуется скачать и установить Nvidia CUDA Toolkit 7.5 для MS Visual Studio 2013.
2. Далее необходимо в папку со своим проектом скопировать два файла MathLibCudaWrapper.dll (математическая библиотека) и MathLibCudaWrapper.xml (встроенная документация к библиотеке для Visual Studio).
3. В своем открытом проекте в MS Visual Studio в пункте References обозревателя решений добавьте ссылку на библиотеку «MathLibCudaWrapper.dll»



4. Теперь у вас есть доступ к основным классам данной библиотеки и их методам:

Класс *MathLibCUDA.MathFuncsIntegral* – для нахождения определенного интеграла методами:

.Simpson – Симпсона

.Simpson_3_8 – Симпсона 3/8

.Gauss – Гаусса с разным числом точек

Класс *MathLibCUDA.MathFuncsDiffEquations* – для решения системы обыкновенных дифференциальных уравнений методами:

.Eiler – Эйлера

.RK2 – Рунге-Кутты 2

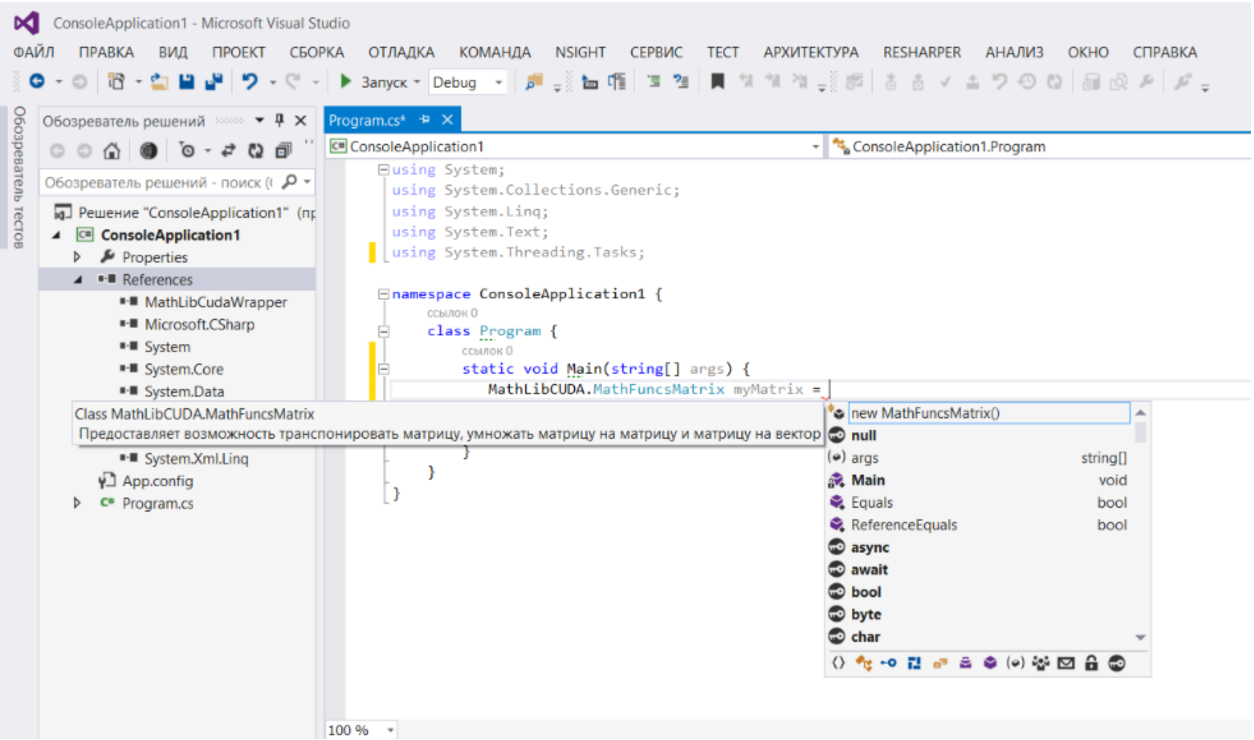
.RK4 – Рунге-Кутты 4

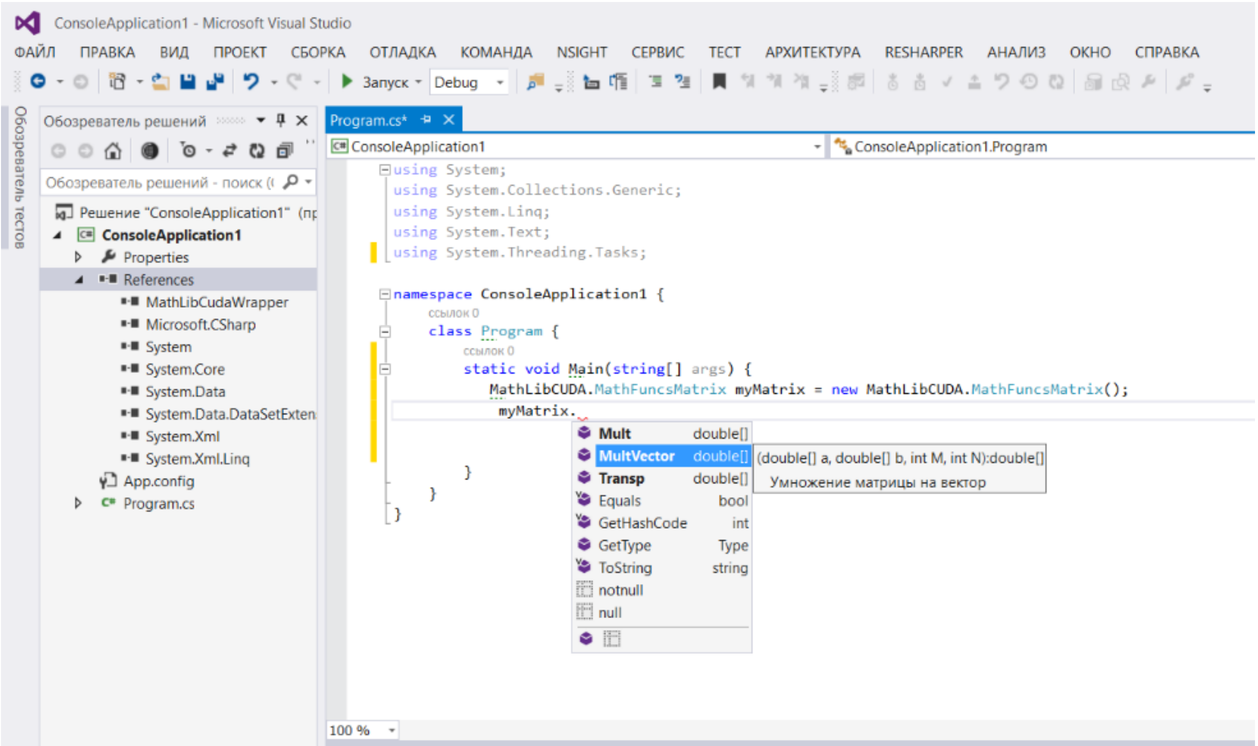
Класс *MathLibCUDA.MathFuncsMatrix* – для операций с матрицами:

.Transp – транспонирование матрицы

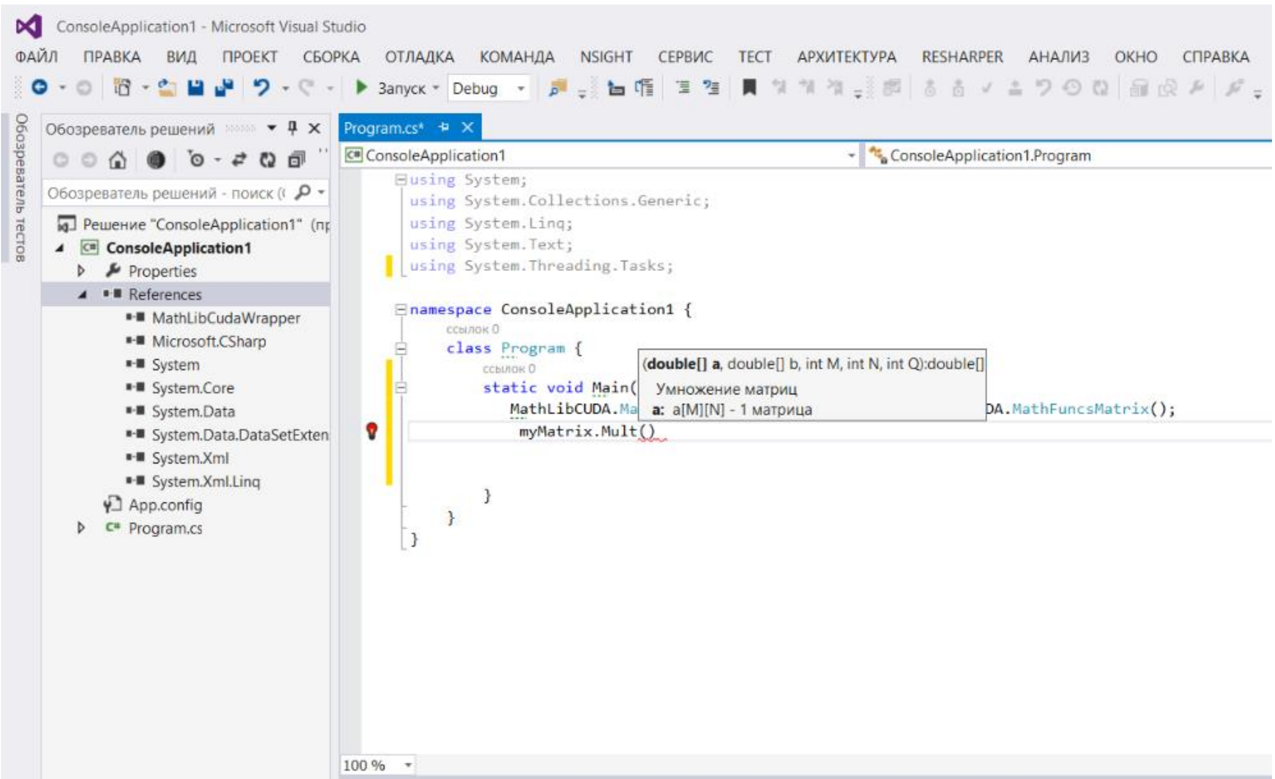
.Mult – умножение матрицы на матрицу

.MultVector – умножение матрицы на вектор





Каждый метод имеет подсказку, в которой описываются параметры и их предназначение

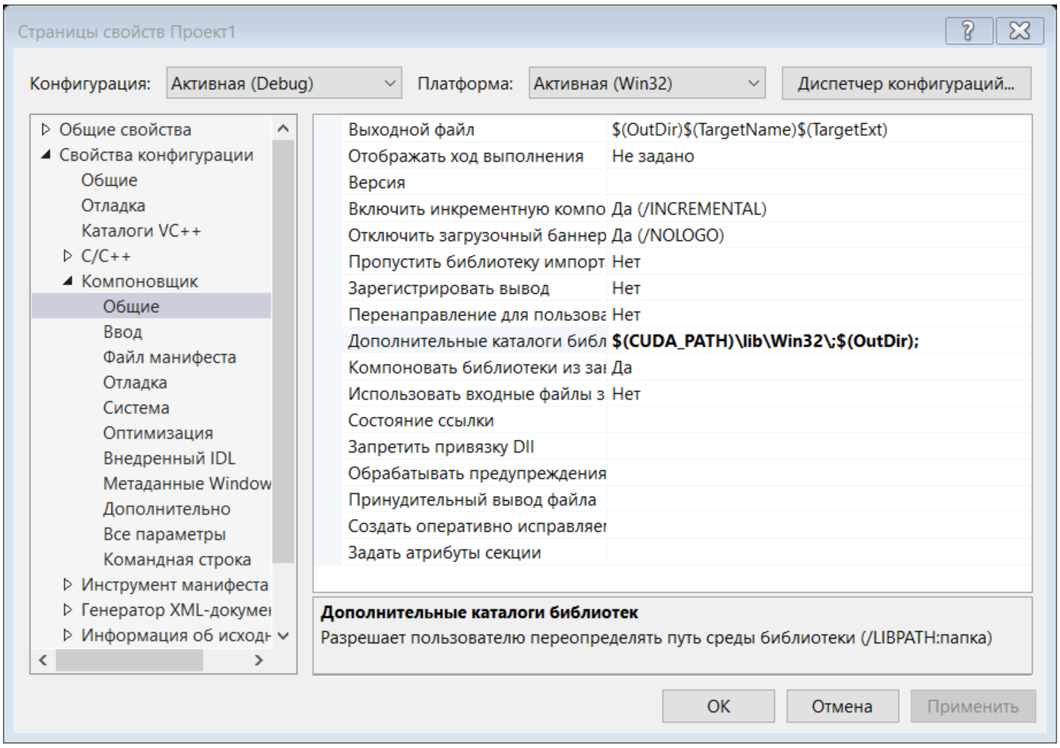


Пример умножения матриц с использованием данной математической библиотеки:

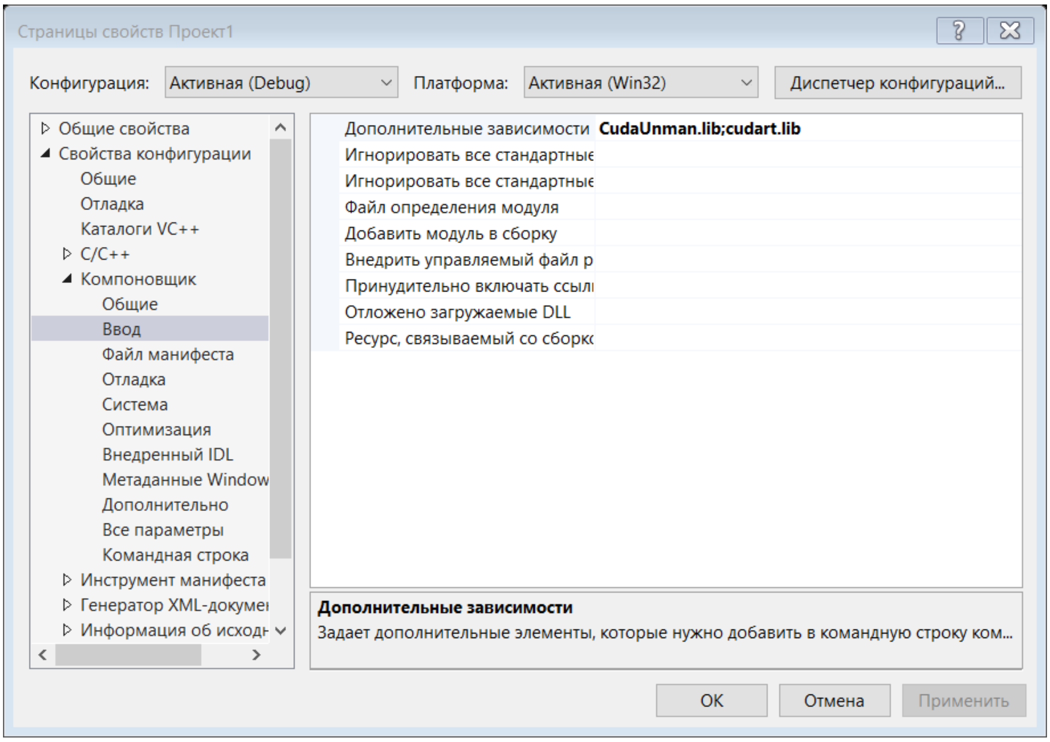
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using MathLibCUDA;//содержит классы и методы данной библиотеки

namespace ConsoleApplication1 {
    class Program {
        static void Main(string[] args) {

           //создаем объект класса MathFuncsMatrix
           MathFuncsMatrix myMatrix = new MathLibCUDA.MathFuncsMatrix();

           int M = 3;
           int N = 4;
           int Q = 2;
           var A = new double[M * N];
           var B = new double[N * Q];
           double[] C;

           for (int i = 0; i < M; i++)
               for (int j = 0; j < N; j++)
                   A[i * N + j] = 1;

           for (int i = 0; i < N; i++)
               for (int j = 0; j < Q; j++)
                   B[i * Q + j] = 2;

           //вызываем соответствующий метод у объекта myMatrix
           C = myMatrix.Mult(A, B, M, N, Q);
        }
    }
}
```

Для приложений на C++

1. Чтобы воспользоваться данной математической библиотекой помимо видеокарты Nvidia с поддержкой CUDA (все современные видеокарты поддерживают данную технологию) необходимо следующее программное обеспечение: MS Visual Studio 2013 и выше (на версиях младше стабильная работа приложения не гарантируется), также потребуется скачать и установить Nvidia CUDA Toolkit 7.5 для MS Visual Studio 2013.
2. Далее необходимо в папку со своим проектом скопировать два файла CudaUnman.lib (математическая библиотека) и CudaMathFuncs.h (заголовочный файл с набором всех математических функций).
3. В свойствах проекта зайти на вкладку *Компоновщик –> Общие –> Дополнительные каталоги библиотек* прописать следующий путь: *$(CUDA_PATH)\lib\Win32\;$(OutDir);*



4. В свойствах проекта зайти на вкладку *Компоновщик –> Общие –> Дополнительные зависисмости* указать имена следующих библиотек: *CudaUnman.lib; cudart.lib*



5. В своем открытом проекте в MS Visual Studio добавляем заголовочный файл «CudaMathFuncs.h»
6. Теперь у вас есть доступ к основным классам данной библиотеки и их методам:

Класс *MyCudaMathFuncs::Integrals*– для нахождения определенного интеграла методами:

->_Simpson – Симпсона

->_Simpson_3_8 – Симпсона 3/8

->_Gauss – Гаусса с разным числом точек

Класс *MyCudaMathFuncs::DiffEquations* – для решения системы обыкновенных дифференциальных уравнений методами:

->_Eiler – Эйлера

->_RK2 – Рунге-Кутты 2

->_RK4 – Рунге-Кутты 4

Класс *MyCudaMathFuncs::Matrix* – для операций с матрицами:

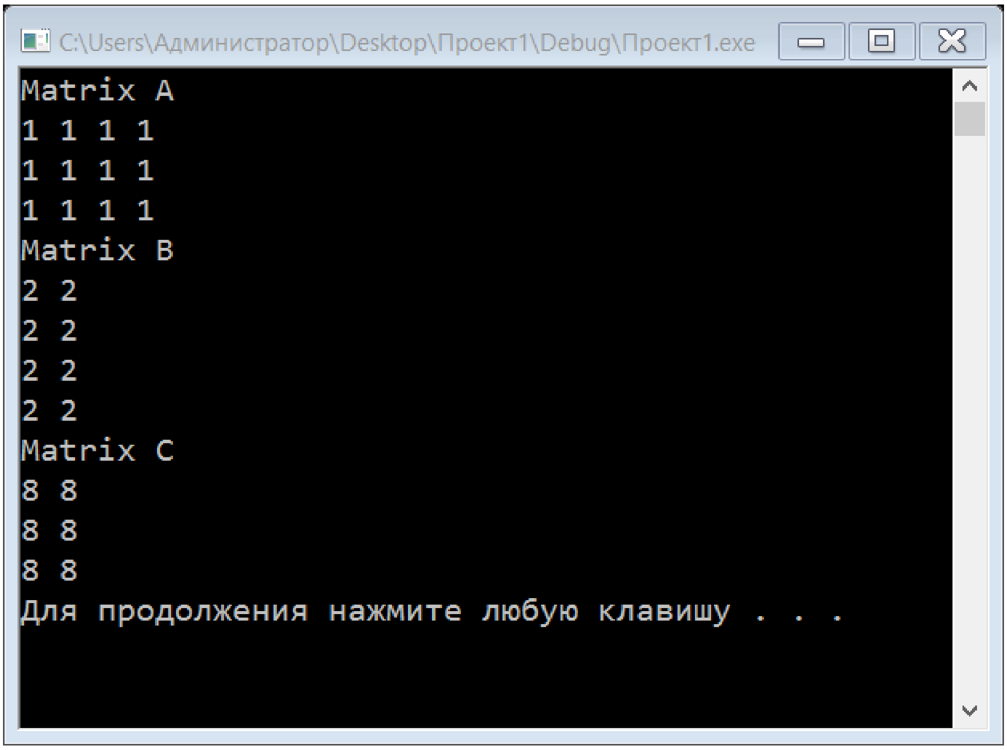
->_Transp – транспонирование матрицы

->_Mult – умножение матрицы на матрицу

->_MultVector – умножение матрицы на вектор

Пример умножения матрицы на матрицу с использованием данной математической библиотеки:

```
#include <stdio.h>
#include <cstdlib>
#include <iostream>
#include "CudaMathFuncs.h"

using namespace std;
void main() {

	//создаем объект класса MathFuncsMatrix
	MyCudaMathFuncs::Matrix *myMatrix = new MyCudaMathFuncs::Matrix();

	int M = 3;
	int N = 4;
	int Q = 2;
	double *A = new double[M * N];
	double *B = new double[N * Q];
	double *C = new double[M * Q];;

	for (int i = 0; i < M; i++)
		for (int j = 0; j < N; j++)
			A[i * N + j] = 1;

	for (int i = 0; i < N; i++)
		for (int j = 0; j < Q; j++)
			B[i * Q + j] = 2;


	cout << "Matrix A" << endl;
	for (int i = 0; i < M; i++){
		for (int j = 0; j < N; j++)
			cout<< A[i*N + j]<< " ";
		cout<<endl;
	}

	cout << "Matrix B" << endl;
	for (int i = 0; i < N; i++){
		for (int j = 0; j < Q; j++)
			cout << B[i*Q + j] << " ";
		cout << endl;
	}

	//вызываем соответствующий метод у объекта myMatrix
	C = myMatrix->_Mult(A, B, M, N, Q);

	cout << "Matrix C" << endl;
	for (int i = 0; i < M; i++){
		for (int j = 0; j < Q; j++)
			cout << C[i*Q + j] << " ";
		cout << endl;
	}

	system("pause");
}
```


C:\Users\Администратор\Desktop\Проект1\Debug\Проект1.exe
Matrix A
1 1 1 1
1 1 1 1
1 1 1 1
Matrix B
2 2
2 2
2 2
2 2
Matrix C
8 8
8 8
8 8
Для продолжения нажмите любую клавишу . . .

Инструкция для программиста

Для открытия проекта https://github.com/boksts/WrapperCUDA, помимо видеокарты Nvidia с поддержкой CUDA (все современные видеокарты поддерживают данную технологию), необходимо следующее программное обеспечение: MS Visual Studio 2013 и выше (на версиях младше стабильная работа приложения не гарантируется), также потребуется скачать и установить Nvidia CUDA Toolkit 7.5 для MS Visual Studio 2013. Решение включает в себя 4 проекта, код в которых содержит подробные комментарии:

1. **CudaUnman** – CUDA проект, содержит файлы с исходным кодом параллельных алгоритмов на CUDA:
   - DiffEquations.cu – для дифференциальных уравнений.
   - Integral.cu – для интегралов
   - Matrix.cu – для матричных операций
   - CudaMathFuncs.h и CudaMathFuncs.cpp – содержат описание классов и методов данной математической библиотеки. Для доступа к функциям из .cu файлов из .cpp файла используется MiniWrapForCuda.h – в котором описаны функции из всех .cu файлов.
2. **MathLibCudaWrapper** – настроенная динамическая библиотека, которая генерирует .dll файл, чтобы впоследствии его можно было подключить в проект C#. Представляет собой враппер (связующее звено между неуправляемым и управляемым кодом). Включает в себя: MathLibCudaWrapper.h – содержит описание классов и методов будущей библиотеки, а также документацию. MathLibCudaWrapper.cpp – содержит реализацию описанных методов. Здесь реализован вызов методов из проекта CudaUnman, передача массивов из управляемого кода в неуправляемый и обратно, а также передача указателей на функцию (делегатов) из управляемого кода и обратно посредством маршалинга.
3. **CSharp** – C# проект. Здесь реализовано использование всех возможностей данной математической библиотеки.
4. **Cpp** – C++ проект. Здесь показан пример использования математической библиотеки.
5. **UnitTestProject1** – содержит Unit-тесты, которые позволяют проводить проверку на достоверность результатов на случай изменения кода.

Текущие проблемы

В математической библиотеке остался не реализован механизм передачи указателя на функцию с хоста в ядро устройства: в проекте **CudaUnman** для интегралов это подынтегральная функция, а для дифференциальных уравнений система дифференциальных уравнений, поэтому они жестко задаются непосредственно в коде с параллельными алгоритмами. Для примера передача данных функций из проекта C# в проект **CudaUnman** через **MathLibCudaWrapper** полностью реализована (в коде есть комментарии во всех проектах в соответствующих местах). Данная проблема связана с тем, что в технологии CUDA нет явного описания функций для такого рода действий. Одним из вариантов решения сложившейся ситуации является передача не указателя на функцию из управляемого кода, а строки в неуправляемый код, построение дерева, используя метод рекурсивного спуска, и передача дерева в ядро устройства.