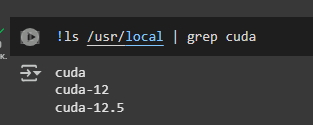
**Лабораторная работа**

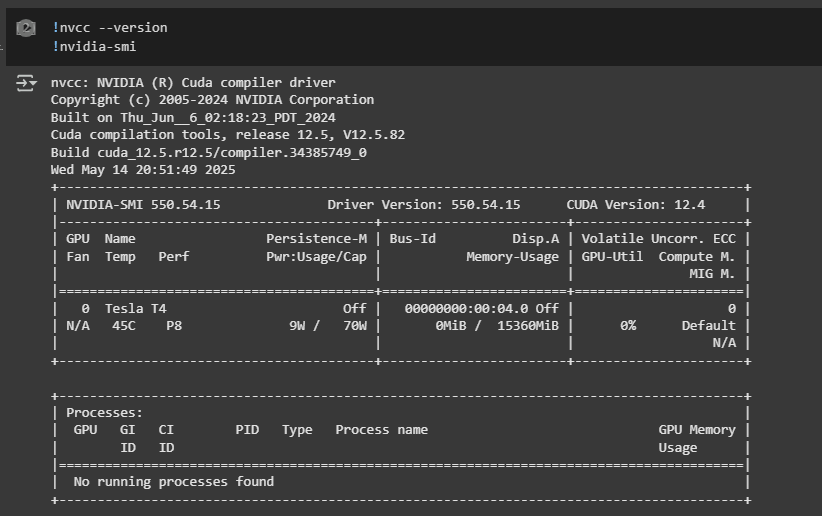
**NVIDIA CUDA**

**Работа с CUDA в Google Colab**

Проверка версий CUDA.



Информация о компиляторе и драйвере GPU.



Для настройки его требуется несколько дополнительных команд

# 1. Удаление старых репозиториев (если они есть)

!rm -rf /var/cuda-repo-\*

# 2. Обновление списка пакетов

!apt-get update --allow-releaseinfo-change

# 3. Скачивание и установка ключей репозитория CUDA

!wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2204/x86\_64/cuda-keyring\_1.0-1\_all.deb

!dpkg -i cuda-keyring\_1.0-1\_all.deb

# 4. Обновление пакетов и установка CUDA Toolkit 11.8

!apt-get update

!apt-get install -y cuda-toolkit-11-8

# 5. Настройка переменных окружения

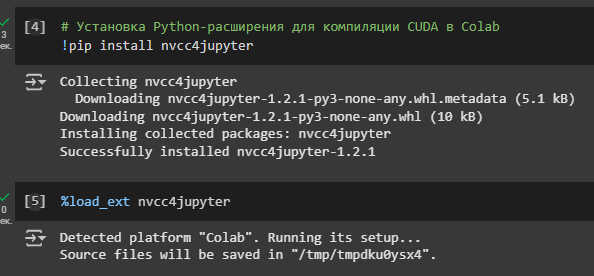
import os

os.environ['PATH'] = '/usr/local/cuda-11.8/bin:' + os.environ['PATH']

os.environ['LD\_LIBRARY\_PATH'] = '/usr/local/cuda-11.8/lib64:' + os.environ.get('LD\_LIBRARY\_PATH', '')

!pip install nvcc4jupyter —расширение для запуска CUDA в Jupyter.

Команда %load\_ext nvcc4jupyter загружает расширение (Jupyter magic) для поддержки CUDA-кода (nvcc) в Jupyter Notebook.



# Проверка

%%cuda

#include <stdio.h>

\_\_global\_\_ void addKernel(int a, int b, int \*c) {

    \*c = a + b;

}

int main() {

    int a = 5, b = 3, c;

    int \*d\_c;

    cudaMalloc(&d\_c, sizeof(int));

    addKernel<<<1, 1>>>(a, b, d\_c);

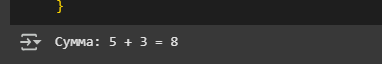
    cudaMemcpy(&c, d\_c, sizeof(int), cudaMemcpyDeviceToHost);

    printf("Сумма: %d + %d = %d\n", a, b, c);

    cudaFree(d\_c);

    return 0;

}



# Пример лекция 1

%%cuda

#include <stdio.h>

#define N (1024\*1024)

\_\_global\_\_ void kernel ( float \* dA )

{ int idx = blockIdx.x \* blockDim.x + threadIdx.x;

 float x = 2.0f \* 3.1415926f \* (float) idx / (float) N;

 dA [idx] = sinf (sqrtf ( x ) );

}

int main ( int argc, char \* argv [] )

{float \*hA, \*dA;

hA = ( float\* ) malloc (N \* sizeof ( float ) );

cudaMalloc ( (void\*\*)&dA, N \* sizeof ( float ) );

kernel <<< N/512, 512 >>> ( dA );

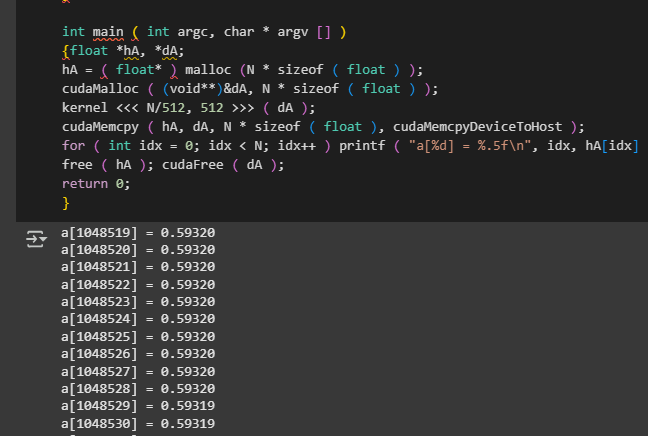
cudaMemcpy ( hA, dA, N \* sizeof ( float ), cudaMemcpyDeviceToHost );

for ( int idx = 0; idx < N; idx++ ) printf ( "a[%d] = %.5f\n", idx, hA[idx] );

free ( hA ); cudaFree ( dA );

return 0;

}



# Пример 1 Лекция 2

# Результат 1.1

%%cuda

#include <stdlib.h>

#include <stdio.h>

\_\_global\_\_ void function (float \*dA, float \*dB, float \*dC, int size )

{

int i = blockIdx.x \* blockDim.x + threadIdx.x;

if ( i < size ) dC [i] = dA[i] + dB[i];

}

int main( int argc, char\* argv[] )

{// инициализация переменных-событий для таймера

float timerValueGPU, timerValueCPU;

cudaEvent\_t start, stop;

cudaEventCreate ( &start );

cudaEventCreate ( &stop );

float \*hA,\*hB,\*hC,\*dA,\*dB,\*dC;

int size = 512 \* 50000; // размер каждого массива

int N\_thread = 512; // число нитей в блоке

int N\_blocks,i;

// задание массивов hA,hB,hC для host

unsigned int mem\_size = sizeof(float)\*size;

hA = (float\*) malloc (mem\_size);

hB = (float\*) malloc (mem\_size);

hC = (float\*) malloc (mem\_size);

// выделение памяти на device под массивы hA, hB, hC

cudaMalloc ((void\*\*) &dA, mem\_size);

cudaMalloc ((void\*\*) &dB, mem\_size);

cudaMalloc ((void\*\*) &dC, mem\_size);

// заполнение массивов hA,hB и обнуление hC

for ( i=0; i < size; i++ )

{hA[i] = 1.0f / ( ( i + 1.0f ) \* ( i + 1.0f ) );

 hB[i] = expf ( 1.0f / ( i + 1.0f ) );

 hC[i]= 0.0f;

}

 // определение числа блоков

if ((size % N\_thread)==0)

{ N\_blocks = size / N\_thread;

} else

{ N\_blocks = ( int )( size / N\_thread )+1;

}

dim3 blocks( N\_blocks );

 // ----------------------GPU вариант -------------------

// Старт таймера

cudaEventRecord ( start, 0);

// Копирование массивов с host на device

cudaMemcpy ( dA, hA, mem\_size, cudaMemcpyHostToDevice);

cudaMemcpy ( dB, hB, mem\_size, cudaMemcpyHostToDevice);

// Запуск функции-ядра

function <<< N\_blocks, N\_thread >>> (dA, dB, dC, size);

// Копирование результат с device на host

cudaMemcpy ( hC, dC, mem\_size, cudaMemcpyDeviceToHost );

// Остановка таймера и вывод времени

// вычисления GPU варианта

cudaEventRecord ( stop, 0 );

cudaEventSynchronize ( stop );

cudaEventElapsedTime ( &timerValueGPU, start, stop );

printf ("\n GPU calculation time: %f ms\n", timerValueGPU );

 // --------------------- CPU вариант --------------------

// Старт таймера

cudaEventRecord ( start, 0 );

// вычисления

for ( i = 0; i < size; i++ ) hC[i] = hA[i] + hB[i];

// Остановка таймера и вывод времени

// вычисления СPU варианта

cudaEventRecord ( stop, 0 );

cudaEventSynchronize ( stop );

cudaEventElapsedTime ( &timerValueCPU, start, stop );

printf ("\n CPU calculation time: %f ms\n", timerValueCPU );

// Вывод коэффициента ускорения

printf ("\n Rate: %f x\n",timerValueCPU/timerValueGPU);

 // Освобождение памяти на host и device

free ( hA );

free ( hB );

free ( hC );

cudaFree ( dA );

cudaFree ( dB );

cudaFree ( dC );

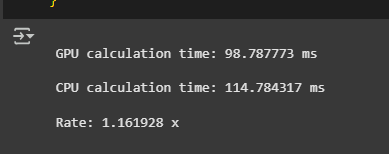
// уничтожение переменных-событий

cudaEventDestroy ( start );

cudaEventDestroy ( stop );

return 0;

}



# Пример 1 Лекция 2

# Результат 1.2

%%cuda

#include <stdlib.h>

#include <stdio.h>

\_\_global\_\_ void function (float \*dA, float \*dB, float \*dC, int size )

{

int i = blockIdx.x \* blockDim.x + threadIdx.x;

if ( i < size ) dC [i] = dA[i] + dB[i];

}

int main( int argc, char\* argv[] )

{// инициализация переменных-событий для таймера

float timerValueGPU, timerValueCPU;

cudaEvent\_t start, stop;

cudaEventCreate ( &start );

cudaEventCreate ( &stop );

float \*hA,\*hB,\*hC,\*dA,\*dB,\*dC;

int size = 512 \* 50000; // размер каждого массива

int N\_thread = 512; // число нитей в блоке

int N\_blocks,i;

// задание массивов hA,hB,hC для host

unsigned int mem\_size = sizeof(float)\*size;

hA = (float\*) malloc (mem\_size);

hB = (float\*) malloc (mem\_size);

hC = (float\*) malloc (mem\_size);

// выделение памяти на device под массивы hA, hB, hC

cudaMalloc ((void\*\*) &dA, mem\_size);

cudaMalloc ((void\*\*) &dB, mem\_size);

cudaMalloc ((void\*\*) &dC, mem\_size);

// заполнение массивов hA,hB и обнуление hC

for ( i=0; i < size; i++ )

{hA[i] = 1.0f / ( ( i + 1.0f ) \* ( i + 1.0f ) );

 hB[i] = expf ( 1.0f / ( i + 1.0f ) );

 hC[i]= 0.0f;

}

 // определение числа блоков

if ((size % N\_thread)==0)

{ N\_blocks = size / N\_thread;

} else

{ N\_blocks = ( int )( size / N\_thread )+1;

}

dim3 blocks( N\_blocks );

// ----------------------GPU вариант -------------------

// Копирование массивов с host на device

cudaMemcpy ( dA, hA, mem\_size, cudaMemcpyHostToDevice);

cudaMemcpy ( dB, hB, mem\_size, cudaMemcpyHostToDevice);

// Старт таймера

cudaEventRecord ( start, 0);

// Запуск функции-ядра

function <<< N\_blocks, N\_thread >>> (dA, dB, dC, size);

// Ожидание завершения выполнения функции-ядра

cudaDeviceSynchronize ();

// Остановка таймера и вывод времени

// вычисления GPU варианта

cudaEventRecord ( stop, 0 );

cudaEventSynchronize ( stop );

cudaEventElapsedTime ( &timerValueGPU, start, stop );

printf ("\n GPU calculation time: %f ms\n", timerValueGPU );

 // --------------------- CPU вариант --------------------

// Старт таймера

cudaEventRecord ( start, 0 );

// вычисления

for ( i = 0; i < size; i++ ) hC[i] = hA[i] + hB[i];

// Остановка таймера и вывод времени

// вычисления СPU варианта

cudaEventRecord ( stop, 0 );

cudaEventSynchronize ( stop );

cudaEventElapsedTime ( &timerValueCPU, start, stop );

printf ("\n CPU calculation time: %f ms\n", timerValueCPU );

// Вывод коэффициента ускорения

printf ("\n Rate: %f x\n",timerValueCPU/timerValueGPU);

 // Освобождение памяти на host и device

free ( hA );

free ( hB );

free ( hC );

cudaFree ( dA );

cudaFree ( dB );

cudaFree ( dC );

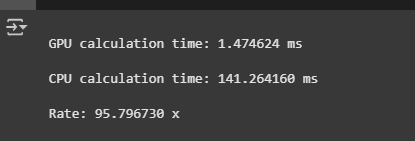
// уничтожение переменных-событий

cudaEventDestroy ( start );

cudaEventDestroy ( stop );

return 0;

}



# Пример 2 Лекция 2

# Результат 2.1

%%cuda

#include <stdlib.h>

#include <stdio.h>

\_\_global\_\_ void function (float \*dA, float \*dB, float \*dC, int size )

{

int i = blockIdx.x \* blockDim.x + threadIdx.x;

if ( i < size ) dC [i] = sinf ( sinf ( dA[i] \* dB[i] ) );

}

int main( int argc, char\* argv[] )

{// инициализация переменных-событий для таймера

float timerValueGPU, timerValueCPU;

cudaEvent\_t start, stop;

cudaEventCreate ( &start );

cudaEventCreate ( &stop );

float \*hA,\*hB,\*hC,\*dA,\*dB,\*dC;

int size = 512 \* 50000; // размер каждого массива

int N\_thread = 512; // число нитей в блоке

int N\_blocks,i;

// задание массивов hA,hB,hC для host

unsigned int mem\_size = sizeof(float)\*size;

hA = (float\*) malloc (mem\_size);

hB = (float\*) malloc (mem\_size);

hC = (float\*) malloc (mem\_size);

// выделение памяти на device под массивы hA, hB, hC

cudaMalloc ((void\*\*) &dA, mem\_size);

cudaMalloc ((void\*\*) &dB, mem\_size);

cudaMalloc ((void\*\*) &dC, mem\_size);

// заполнение массивов hA,hB и обнуление hC

for ( i=0; i < size; i++ )

{hA[i] = 1.0f / ( ( i + 1.0f ) \* ( i + 1.0f ) );

 hB[i] = expf ( 1.0f / ( i + 1.0f ) );

 hC[i]= 0.0f;

}

 // определение числа блоков

if ((size % N\_thread)==0)

{ N\_blocks = size / N\_thread;

} else

{ N\_blocks = ( int )( size / N\_thread )+1;

}

dim3 blocks( N\_blocks );

 // ----------------------GPU вариант -------------------

// Старт таймера

cudaEventRecord ( start, 0);

// Копирование массивов с host на device

cudaMemcpy ( dA, hA, mem\_size, cudaMemcpyHostToDevice);

cudaMemcpy ( dB, hB, mem\_size, cudaMemcpyHostToDevice);

// Запуск функции-ядра

function <<< N\_blocks, N\_thread >>> (dA, dB, dC, size);

// Копирование результат с device на host

cudaMemcpy ( hC, dC, mem\_size, cudaMemcpyDeviceToHost );

// Остановка таймера и вывод времени

// вычисления GPU варианта

cudaEventRecord ( stop, 0 );

cudaEventSynchronize ( stop );

cudaEventElapsedTime ( &timerValueGPU, start, stop );

printf ("\n GPU calculation time: %f ms\n", timerValueGPU );

 // --------------------- CPU вариант --------------------

// Старт таймера

cudaEventRecord ( start, 0 );

// вычисления

for ( i = 0; i < size; i++ ) hC [i] = sinf ( sinf ( hA[i] \* hB[i] ) );

// Остановка таймера и вывод времени

// вычисления СPU варианта

cudaEventRecord ( stop, 0 );

cudaEventSynchronize ( stop );

cudaEventElapsedTime ( &timerValueCPU, start, stop );

printf ("\n CPU calculation time: %f ms\n", timerValueCPU );

// Вывод коэффициента ускорения

printf ("\n Rate: %f x\n",timerValueCPU/timerValueGPU);

 // Освобождение памяти на host и device

free ( hA );

free ( hB );

free ( hC );

cudaFree ( dA );

cudaFree ( dB );

cudaFree ( dC );

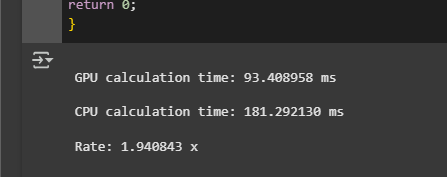
// уничтожение переменных-событий

cudaEventDestroy ( start );

cudaEventDestroy ( stop );

return 0;

}



# Пример 2 Лекция 2

# Результат 2.2

%%cuda

#include <stdlib.h>

#include <stdio.h>

\_\_global\_\_ void function (float \*dA, float \*dB, float \*dC, int size )

{

int i = blockIdx.x \* blockDim.x + threadIdx.x;

if ( i < size ) dC [i] = sinf ( sinf ( dA[i] \* dB[i] ) );

}

int main( int argc, char\* argv[] )

{// инициализация переменных-событий для таймера

float timerValueGPU, timerValueCPU;

cudaEvent\_t start, stop;

cudaEventCreate ( &start );

cudaEventCreate ( &stop );

float \*hA,\*hB,\*hC,\*dA,\*dB,\*dC;

int size = 512 \* 50000; // размер каждого массива

int N\_thread = 512; // число нитей в блоке

int N\_blocks,i;

// задание массивов hA,hB,hC для host

unsigned int mem\_size = sizeof(float)\*size;

hA = (float\*) malloc (mem\_size);

hB = (float\*) malloc (mem\_size);

hC = (float\*) malloc (mem\_size);

// выделение памяти на device под массивы hA, hB, hC

cudaMalloc ((void\*\*) &dA, mem\_size);

cudaMalloc ((void\*\*) &dB, mem\_size);

cudaMalloc ((void\*\*) &dC, mem\_size);

// заполнение массивов hA,hB и обнуление hC

for ( i=0; i < size; i++ )

{hA[i] = 1.0f / ( ( i + 1.0f ) \* ( i + 1.0f ) );

 hB[i] = expf ( 1.0f / ( i + 1.0f ) );

 hC[i]= 0.0f;

}

 // определение числа блоков

if ((size % N\_thread)==0)

{ N\_blocks = size / N\_thread;

} else

{ N\_blocks = ( int )( size / N\_thread )+1;

}

dim3 blocks( N\_blocks );

// ----------------------GPU вариант -------------------

// Копирование массивов с host на device

cudaMemcpy ( dA, hA, mem\_size, cudaMemcpyHostToDevice);

cudaMemcpy ( dB, hB, mem\_size, cudaMemcpyHostToDevice);

// Старт таймера

cudaEventRecord ( start, 0);

// Запуск функции-ядра

function <<< N\_blocks, N\_thread >>> (dA, dB, dC, size);

// Ожидание завершения выполнения функции-ядра

cudaDeviceSynchronize ();

// Остановка таймера и вывод времени

// вычисления GPU варианта

cudaEventRecord ( stop, 0 );

cudaEventSynchronize ( stop );

cudaEventElapsedTime ( &timerValueGPU, start, stop );

printf ("\n GPU calculation time: %f ms\n", timerValueGPU );

 // --------------------- CPU вариант --------------------

// Старт таймера

cudaEventRecord ( start, 0 );

// вычисления

for ( i = 0; i < size; i++ ) hC [i] = sinf ( sinf ( hA[i] \* hB[i] ) );

// Остановка таймера и вывод времени

// вычисления СPU варианта

cudaEventRecord ( stop, 0 );

cudaEventSynchronize ( stop );

cudaEventElapsedTime ( &timerValueCPU, start, stop );

printf ("\n CPU calculation time: %f ms\n", timerValueCPU );

// Вывод коэффициента ускорения

printf ("\n Rate: %f x\n",timerValueCPU/timerValueGPU);

 // Освобождение памяти на host и device

free ( hA );

free ( hB );

free ( hC );

cudaFree ( dA );

cudaFree ( dB );

cudaFree ( dC );

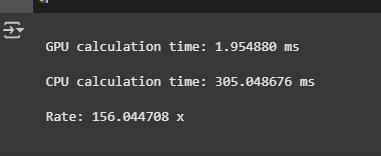
// уничтожение переменных-событий

cudaEventDestroy ( start );

cudaEventDestroy ( stop );

return 0;

}



# PINNED-Память

%%cuda

#include <stdlib.h>

#include <stdio.h>

\_\_global\_\_ void function (float \*dA, float \*dB, float \*dC, int size )

{

int i = blockIdx.x \* blockDim.x + threadIdx.x;

if ( i < size ) dC [i] = sinf ( sinf ( dA[i] \* dB[i] ) );

}

int main( int argc, char\* argv[] )

{// инициализация переменных-событий для таймера

float timerValueGPU, timerValueCPU;

cudaEvent\_t start, stop;

cudaEventCreate ( &start );

cudaEventCreate ( &stop );

float \*hA,\*hB,\*hC,\*dA,\*dB,\*dC;

int size = 512 \* 50000; // размер каждого массива

int N\_thread = 512; // число нитей в блоке

int N\_blocks,i;

// задание массивов hA,hB,hC для host

unsigned int mem\_size = sizeof(float)\*size;

cudaHostAlloc ((void\*\*) &hA, mem\_size, cudaHostAllocDefault);

cudaHostAlloc ((void\*\*) &hB, mem\_size, cudaHostAllocDefault);

cudaHostAlloc ((void\*\*) &hC, mem\_size, cudaHostAllocDefault);

// выделение памяти на device под массивы hA, hB, hC

cudaMalloc ((void\*\*) &dA, mem\_size);

cudaMalloc ((void\*\*) &dB, mem\_size);

cudaMalloc ((void\*\*) &dC, mem\_size);

// заполнение массивов hA,hB и обнуление hC

for ( i=0; i < size; i++ )

{hA[i] = 1.0f / ( ( i + 1.0f ) \* ( i + 1.0f ) );

 hB[i] = expf ( 1.0f / ( i + 1.0f ) );

 hC[i]= 0.0f;

}

 // определение числа блоков

if ((size % N\_thread)==0)

{ N\_blocks = size / N\_thread;

} else

{ N\_blocks = ( int )( size / N\_thread )+1;

}

dim3 blocks( N\_blocks );

// ----------------------GPU вариант -------------------

// Копирование массивов с host на device

cudaMemcpy ( dA, hA, mem\_size, cudaMemcpyHostToDevice);

cudaMemcpy ( dB, hB, mem\_size, cudaMemcpyHostToDevice);

// Старт таймера

cudaEventRecord ( start, 0);

// Запуск функции-ядра

function <<< N\_blocks, N\_thread >>> (dA, dB, dC, size);

// Ожидание завершения выполнения функции-ядра

cudaDeviceSynchronize ();

// Остановка таймера и вывод времени

// вычисления GPU варианта

cudaEventRecord ( stop, 0 );

cudaEventSynchronize ( stop );

cudaEventElapsedTime ( &timerValueGPU, start, stop );

printf ("\n GPU calculation time: %f ms\n", timerValueGPU );

 // --------------------- CPU вариант --------------------

// Старт таймера

cudaEventRecord ( start, 0 );

// вычисления

for ( i = 0; i < size; i++ ) hC [i] = sinf ( sinf ( hA[i] \* hB[i] ) );

// Остановка таймера и вывод времени

// вычисления СPU варианта

cudaEventRecord ( stop, 0 );

cudaEventSynchronize ( stop );

cudaEventElapsedTime ( &timerValueCPU, start, stop );

printf ("\n CPU calculation time: %f ms\n", timerValueCPU );

// Вывод коэффициента ускорения

printf ("\n Rate: %f x\n",timerValueCPU/timerValueGPU);

 // Освобождение памяти на host и device

cudaFreeHost ( hA );

cudaFreeHost ( hB );

cudaFreeHost ( hC );

cudaFree ( dA );

cudaFree ( dB );

cudaFree ( dC );

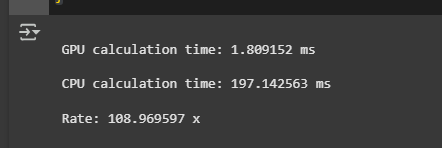
// уничтожение переменных-событий

cudaEventDestroy ( start );

cudaEventDestroy ( stop );

return 0;

}



# Пример 1 Лекция 3

%%cuda

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <cuda\_runtime.h>

#define EPS 1.e-15  // Точность приближенного решения

#define N 10240     // Число уравнений в системе

#define N\_thread 512 // Число нитей в блоке

\_\_global\_\_ void Solve(double \*dA, double \*dF, double \*dX0, double \*dX1, int equations) {

    double aa, sum = 0.0;

    int t = blockIdx.x \* blockDim.x + threadIdx.x;

    if (t < N) {

        for (int j = 0; j < N; j++) {

            sum += dA[j + t \* N] \* dX0[j];

            if (j == t) aa = dA[j + t \* N];

        }

        dX1[t] = dX0[t] + (dF[t] - sum) / aa;  // Исправлен символ минуса

    }

}

\_\_global\_\_ void Eps(double \*dX0, double \*dX1, double \*delta, int equations) {

    int i = blockIdx.x \* blockDim.x + threadIdx.x;

    if (i < N) {

        delta[i] = fabs(dX0[i] - dX1[i]);

        dX0[i] = dX1[i];

    }

}

int main() {

    int equations = N;

    // Инициализация таймеров

    float timerValueCPU, timerValueGPU\_A, timerValueGPU\_AT;

    float timerValueGPU\_kernel\_A, timerValueGPU\_kernel\_AT;

    cudaEvent\_t start, stop, start\_kernel, stop\_kernel;

    cudaEventCreate(&start);

    cudaEventCreate(&stop);

    cudaEventCreate(&start\_kernel);

    cudaEventCreate(&stop\_kernel);

    // Размеры данных

    int size = N \* N;

    unsigned int mem\_sizeA = sizeof(double) \* size;

    unsigned int mem\_sizeX = sizeof(double) \* N;

    // Выделение памяти на хосте

    double \*hA = (double\*)malloc(mem\_sizeA);

    double \*hF = (double\*)malloc(mem\_sizeX);

    double \*hX = (double\*)malloc(mem\_sizeX);

    double \*hX0 = (double\*)malloc(mem\_sizeX);

    double \*hX1 = (double\*)malloc(mem\_sizeX);

    double \*hDelta = (double\*)malloc(mem\_sizeX);

    // Инициализация данных

    for (int i = 0; i < N; i++) {

        hF[i] = 1.0;

        hX[i] = 1.0;

        hX0[i] = 0.0;

        for (int j = 0; j < N; j++) {

            hA[j + i \* N] = (i == j) ? 2.0 : 0.1;

        }

    }

    // Выделение памяти на устройстве

    double \*dA, \*dF, \*dX0, \*dX1, \*delta;

    cudaMalloc((void\*\*)&dA, mem\_sizeA);

    cudaMalloc((void\*\*)&dF, mem\_sizeX);

    cudaMalloc((void\*\*)&dX0, mem\_sizeX);

    cudaMalloc((void\*\*)&dX1, mem\_sizeX);

    cudaMalloc((void\*\*)&delta, mem\_sizeX);

    // Конфигурация запуска ядра

    dim3 blocks((N + N\_thread - 1) / N\_thread);

    dim3 threads(N\_thread);

    // ---------------------- GPU вариант (синхронный) -------------------

    cudaEventRecord(start, 0);

    // Копирование данных

    cudaMemcpy(dA, hA, mem\_sizeA, cudaMemcpyHostToDevice);

    cudaMemcpy(dF, hF, mem\_sizeX, cudaMemcpyHostToDevice);

    cudaMemcpy(dX0, hX0, mem\_sizeX, cudaMemcpyHostToDevice);

    double eps = 1.0;

    int k = 0;

    cudaEventRecord(start\_kernel, 0);

    while (eps > EPS) {

        k++;

        Solve<<<blocks, threads>>>(dA, dF, dX0, dX1, N);

        Eps<<<blocks, threads>>>(dX0, dX1, delta, N);

        cudaMemcpy(hDelta, delta, mem\_sizeX, cudaMemcpyDeviceToHost);

        eps = 0.0;

        for (int j = 0; j < N; j++) eps += hDelta[j];

        eps /= N;

    }

    cudaEventRecord(stop\_kernel, 0);

    cudaEventSynchronize(stop\_kernel);

    cudaMemcpy(hX1, dX0, mem\_sizeX, cudaMemcpyDeviceToHost);

    cudaEventRecord(stop, 0);

    cudaEventSynchronize(stop);

    // Измерение времени

    cudaEventElapsedTime(&timerValueGPU\_A, start, stop);

    cudaEventElapsedTime(&timerValueGPU\_kernel\_A, start\_kernel, stop\_kernel);

    // ---------------------- GPU вариант (асинхронный) -------------------

    cudaStream\_t stream;

    cudaStreamCreate(&stream);

    cudaEventRecord(start, 0);

    cudaMemcpyAsync(dA, hA, mem\_sizeA, cudaMemcpyHostToDevice, stream);

    cudaMemcpyAsync(dF, hF, mem\_sizeX, cudaMemcpyHostToDevice, stream);

    cudaMemcpyAsync(dX0, hX0, mem\_sizeX, cudaMemcpyHostToDevice, stream);

    cudaEventRecord(start\_kernel, 0);

    eps = 1.0; k = 0;

    while (eps > EPS) {

        k++;

        Solve<<<blocks, threads, 0, stream>>>(dA, dF, dX0, dX1, N);

        Eps<<<blocks, threads, 0, stream>>>(dX0, dX1, delta, N);

        cudaMemcpyAsync(hDelta, delta, mem\_sizeX, cudaMemcpyDeviceToHost, stream);

        cudaStreamSynchronize(stream);

        eps = 0.0;

        for (int j = 0; j < N; j++) eps += hDelta[j];

        eps /= N;

    }

    cudaEventRecord(stop\_kernel, 0);

    cudaEventSynchronize(stop\_kernel);

    cudaMemcpyAsync(hX1, dX0, mem\_sizeX, cudaMemcpyDeviceToHost, stream);

    cudaEventRecord(stop, 0);

    cudaEventSynchronize(stop);

    // Измерение времени

    cudaEventElapsedTime(&timerValueGPU\_AT, start, stop);

    cudaEventElapsedTime(&timerValueGPU\_kernel\_AT, start\_kernel, stop\_kernel);

    cudaStreamDestroy(stream);

    // ---------------------- CPU вариант --------------------

    cudaEventRecord(start, 0);

    eps = 1.0; k = 0;

    while (eps > EPS) {

        k++;

        for (int i = 0; i < N; i++) {

            double sum = 0.0;

            for (int j = 0; j < N; j++) {

                sum += hA[j + i \* N] \* hX0[j];

            }

            hX1[i] = hX0[i] + (hF[i] - sum) / hA[i + i \* N];  // Исправлен символ минуса

        }

        eps = 0.0;

        for (int j = 0; j < N; j++) {

            eps += fabs(hX0[j] - hX1[j]);

            hX0[j] = hX1[j];

        }

        eps /= N;

    }

    cudaEventRecord(stop, 0);

    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&timerValueCPU, start, stop);

    // Вывод результатов

    printf("\nCPU calculation time: %.0f ms", timerValueCPU);

    printf("\nGPU calculation time\* A: %.0f ms", timerValueGPU\_A);

    printf("\nGPU calculation time\* AT: %.0f ms", timerValueGPU\_AT);

    printf("\nGPU calculation time\*\* A: %.0f ms", timerValueGPU\_kernel\_A);

    printf("\nGPU calculation time\*\* AT: %.0f ms", timerValueGPU\_kernel\_AT);

    printf("\nRate\* A: %.1f x, Rate\* AT: %.1f x",

           timerValueCPU / timerValueGPU\_A,

           timerValueCPU / timerValueGPU\_AT);

    printf("\nRate\*\* A: %.1f x, Rate\*\* AT: %.1f x\n",

           timerValueCPU / timerValueGPU\_kernel\_A,

           timerValueCPU / timerValueGPU\_kernel\_AT);

    // Освобождение ресурсов

    free(hA); free(hF); free(hX); free(hX0); free(hX1); free(hDelta);

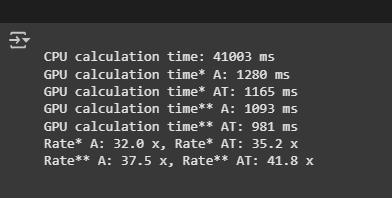
    cudaFree(dA); cudaFree(dF); cudaFree(dX0); cudaFree(dX1); cudaFree(delta);

    cudaEventDestroy(start); cudaEventDestroy(stop);

    cudaEventDestroy(start\_kernel); cudaEventDestroy(stop\_kernel);

    return 0;

}



# Пример 2 Лекция 3

%%cuda

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <cuda\_runtime.h>

#define EPS\_G 1.e-8    // Точность для внешнего цикла

#define EPS\_L 1.e-8    // Точность для внутреннего цикла

#define N 128         // Размер системы

#define TAU 0.01       // Параметр метода

#define BAND\_WIDTH 0.15 // Ширина ленты матрицы

int eqv\_length = N;

// Макрос для диагональных элементов

#define D(i,j) ((i) == (j) ? 1.0 : 0.0)

// Ядра GPU

\_\_global\_\_ void Matrix\_A(double \*dA, double \*dX, int eqv\_length) {

    int j = blockIdx.x \* blockDim.x + threadIdx.x;

    int i = blockIdx.y \* blockDim.y + threadIdx.y;

    int Nd = (int)(BAND\_WIDTH \* eqv\_length);

    if (i < eqv\_length && j < eqv\_length) {

        if (i <= j + Nd && i >= j - Nd) {

            dA[i + j \* eqv\_length] = pow(sin(dX[j]) \* cos(dX[i]), 2.0) + (double)eqv\_length \* D(i, j);

        } else {

            dA[i + j \* eqv\_length] = 0.0; // Ленточная структура матрицы

        }

    }

}

\_\_global\_\_ void AX(double \*dAX, double \*dA, double \*dX, int eqv\_length) {

    int i = blockIdx.x \* blockDim.x + threadIdx.x;

    if (i < eqv\_length) {

        double sum = 0.0;

        for (int j = 0; j < eqv\_length; j++)

            sum += dA[i + j \* eqv\_length] \* dX[j];

        dAX[i] = sum;

    }

}

\_\_global\_\_ void PHI(double \*dPhi, double \*dAX, double \*dF, int eqv\_length) {

    int i = blockIdx.x \* blockDim.x + threadIdx.x;

    if (i < eqv\_length) {

        dPhi[i] = dAX[i] - dF[i];

    }

}

\_\_global\_\_ void D\_PHI(double \*dL, double \*dX0, int eqv\_length) {

    int j = blockIdx.x \* blockDim.x + threadIdx.x;

    int i = blockIdx.y \* blockDim.y + threadIdx.y;

    int Nd = (int)(BAND\_WIDTH \* eqv\_length);

    if (i < eqv\_length && j < eqv\_length) {

        if (i <= j + Nd && i >= j - Nd) {

            int k1, k2;

            double sum1 = 0.0, sum2 = 0.0;

            // Определение области для суммирования

            if (i <= Nd) { k1 = 0; k2 = i + Nd + 1; }

            else if (i < eqv\_length - Nd) { k1 = i - Nd; k2 = i + Nd + 1; }

            else { k1 = i - Nd; k2 = eqv\_length; }

            for (int k = k1; k < k2; k++) {

                sum1 += D(k, j) \* (pow(sin(dX0[i]) \* cos(dX0[k]), 2.0) + D(i, k) \* (double)eqv\_length);

                sum2 += dX0[k] \* (sin(2.0 \* dX0[i]) \* pow(cos(dX0[k]), 2.0) \* D(i, j) -

                                  sin(2.0 \* dX0[k]) \* pow(sin(dX0[i]), 2.0) \* D(k, j));

            }

            dL[i + j \* eqv\_length] = sum1 + sum2;

        } else {

            dL[i + j \* eqv\_length] = 0.0;

        }

    }

}

\_\_global\_\_ void Solve\_L(double \*dL, double \*dPhi, double \*dV0, double \*dV1, int eqv\_length) {

    int i = blockIdx.x \* blockDim.x + threadIdx.x;

    if (i < eqv\_length) {

        double sum = 0.0;

        for (int j = 0; j < eqv\_length; j++)

            sum += dL[i + j \* eqv\_length] \* dV0[j];

        dV1[i] = dV0[i] - (sum - dPhi[i]) / ((double)eqv\_length \* 1.0);

    }

}

\_\_global\_\_ void Eps\_L(double \*dV0, double \*dV1, double \*d\_dV, int eqv\_length) {

    int i = blockIdx.x \* blockDim.x + threadIdx.x;

    if (i < eqv\_length) {

        d\_dV[i] = fabs(dV0[i] - dV1[i]);

        dV0[i] = dV1[i];

    }

}

\_\_global\_\_ void Solve\_G(double \*dX0, double \*dX1, double \*dV0, double tau, int eqv\_length) {

    int i = blockIdx.x \* blockDim.x + threadIdx.x;

    if (i < eqv\_length) {

        dX1[i] = dX0[i] + tau \* dV0[i];

    }

}

\_\_global\_\_ void Eps\_G(double \*dX0, double \*dX1, double \*d\_dX, int eqv\_length) {

    int i = blockIdx.x \* blockDim.x + threadIdx.x;

    if (i < eqv\_length) {

        d\_dX[i] = fabs(dX0[i] - dX1[i]);

        dX0[i] = dX1[i];

    }

}

// CPU реализации

void cpu\_Matrix\_A(double \*A, double \*X, int eqv\_length) {

    int Nd = (int)(BAND\_WIDTH \* eqv\_length);

    for (int j = 0; j < eqv\_length; j++) {

        for (int i = 0; i < eqv\_length; i++) {

            if (i <= j + Nd && i >= j - Nd) {

                A[i + j \* eqv\_length] = pow(sin(X[j]) \* cos(X[i]), 2.0) + (double)eqv\_length \* D(i, j);

            } else {

                A[i + j \* eqv\_length] = 0.0;

            }

        }

    }

}

void cpu\_AX(double \*AX, double \*A, double \*X, int eqv\_length) {

    for (int i = 0; i < eqv\_length; i++) {

        AX[i] = 0.0;

        for (int j = 0; j < eqv\_length; j++) {

            AX[i] += A[i + j \* eqv\_length] \* X[j];

        }

    }

}

void cpu\_PHI(double \*Phi, double \*AX, double \*F, int eqv\_length) {

    for (int i = 0; i < eqv\_length; i++) {

        Phi[i] = AX[i] - F[i];

    }

}

void cpu\_D\_PHI(double \*L, double \*X0, int eqv\_length) {

    int Nd = (int)(BAND\_WIDTH \* eqv\_length);

    for (int j = 0; j < eqv\_length; j++) {

        for (int i = 0; i < eqv\_length; i++) {

            if (i <= j + Nd && i >= j - Nd) {

                int k1, k2;

                double sum1 = 0.0, sum2 = 0.0;

                if (i <= Nd) { k1 = 0; k2 = i + Nd + 1; }

                else if (i < eqv\_length - Nd) { k1 = i - Nd; k2 = i + Nd + 1; }

                else { k1 = i - Nd; k2 = eqv\_length; }

                for (int k = k1; k < k2; k++) {

                    sum1 += D(k, j) \* (pow(sin(X0[i]) \* cos(X0[k]), 2.0) + D(i, k) \* (double)eqv\_length);

                    sum2 += X0[k] \* (sin(2.0 \* X0[i]) \* pow(cos(X0[k]), 2.0) \* D(i, j) -

                                     sin(2.0 \* X0[k]) \* pow(sin(X0[i]), 2.0) \* D(k, j));

                }

                L[i + j \* eqv\_length] = sum1 + sum2;

            } else {

                L[i + j \* eqv\_length] = 0.0;

            }

        }

    }

}

void cpu\_Solve\_L(double \*L, double \*Phi, double \*V0, double \*V1, int eqv\_length) {

    for (int i = 0; i < eqv\_length; i++) {

        double sum = 0.0;

        for (int j = 0; j < eqv\_length; j++) {

            sum += L[i + j \* eqv\_length] \* V0[j];

        }

        V1[i] = V0[i] - (sum - Phi[i]) / ((double)eqv\_length \* 1.0);

    }

}

void cpu\_Eps\_L(double \*V0, double \*V1, double \*dV, int eqv\_length) {

    for (int i = 0; i < eqv\_length; i++) {

        dV[i] = fabs(V0[i] - V1[i]);

        V0[i] = V1[i];

    }

}

void cpu\_Solve\_G(double \*X0, double \*X1, double \*V0, double tau, int eqv\_length) {

    for (int i = 0; i < eqv\_length; i++) {

        X1[i] = X0[i] + tau \* V0[i];

    }

}

void cpu\_Eps\_G(double \*X0, double \*X1, double \*dX, int eqv\_length) {

    for (int i = 0; i < eqv\_length; i++) {

        dX[i] = fabs(X0[i] - X1[i]);

        X0[i] = X1[i];

    }

}

int main() {

    // Инициализация таймеров

    float cpu\_time, gpu\_time;

    cudaEvent\_t start, stop;

    cudaEventCreate(&start);

    cudaEventCreate(&stop);

    // Размеры данных

    unsigned int mem\_sizeX = sizeof(double) \* N;

    unsigned int mem\_sizeA = sizeof(double) \* N \* N;

    // Выделение памяти на хосте

    double \*hA = (double\*)malloc(mem\_sizeA);

    double \*hF = (double\*)malloc(mem\_sizeX);

    double \*hX0 = (double\*)malloc(mem\_sizeX);

    double \*hX1 = (double\*)malloc(mem\_sizeX);

    double \*hV0 = (double\*)malloc(mem\_sizeX);

    double \*hV1 = (double\*)malloc(mem\_sizeX);

    double \*hPhi = (double\*)malloc(mem\_sizeX);

    double \*hAX = (double\*)malloc(mem\_sizeX);

    double \*hL = (double\*)malloc(mem\_sizeA);

    double \*hdX = (double\*)malloc(mem\_sizeX);

    double \*hdV = (double\*)malloc(mem\_sizeX);

    // Инициализация данных

    for (int i = 0; i < N; i++) {

        hF[i] = 1.0;

        hX0[i] = 0.0;

        hV0[i] = 1.0;

    }

    // ---------------------- CPU вариант -------------------

    clock\_t cpu\_start = clock();

    double eps\_G = 1.0, eps\_L;

    int k, j;

    while (eps\_G > EPS\_G) {

        cpu\_Matrix\_A(hA, hX0, N);

        cpu\_AX(hAX, hA, hX0, N);

        cpu\_PHI(hPhi, hAX, hF, N);

        cpu\_D\_PHI(hL, hX0, N);

        for (int i = 0; i < N; i++) hV0[i] = 1.0;

        eps\_L = 1.0;

        while (eps\_L > EPS\_L) {

            cpu\_Solve\_L(hL, hPhi, hV0, hV1, N);

            cpu\_Eps\_L(hV0, hV1, hdV, N);

            eps\_L = 0.0;

            for (j = 0; j < N; j++) eps\_L += hdV[j];

            eps\_L /= N;

        }

        cpu\_Solve\_G(hX0, hX1, hV0, TAU, N);

        cpu\_Eps\_G(hX0, hX1, hdX, N);

        eps\_G = 0.0;

        for (k = 0; k < N; k++) eps\_G += hdX[k];

        eps\_G /= N;

    }

    cpu\_time = (float)(clock() - cpu\_start) / CLOCKS\_PER\_SEC \* 1000;

    printf("CPU calculation time: %.0f ms\n", cpu\_time);

    // ---------------------- GPU вариант -------------------

    // Выделение памяти на устройстве

    double \*dA, \*dF, \*dX0, \*dX1, \*dV0, \*dV1, \*dPhi, \*dAX, \*dL, \*ddX, \*ddV;

    cudaMalloc((void\*\*)&dA, mem\_sizeA);

    cudaMalloc((void\*\*)&dF, mem\_sizeX);

    cudaMalloc((void\*\*)&dX0, mem\_sizeX);

    cudaMalloc((void\*\*)&dX1, mem\_sizeX);

    cudaMalloc((void\*\*)&dV0, mem\_sizeX);

    cudaMalloc((void\*\*)&dV1, mem\_sizeX);

    cudaMalloc((void\*\*)&dPhi, mem\_sizeX);

    cudaMalloc((void\*\*)&dAX, mem\_sizeX);

    cudaMalloc((void\*\*)&dL, mem\_sizeA);

    cudaMalloc((void\*\*)&ddX, mem\_sizeX);

    cudaMalloc((void\*\*)&ddV, mem\_sizeX);

    // Копирование данных на устройство

    cudaMemcpy(dF, hF, mem\_sizeX, cudaMemcpyHostToDevice);

    cudaMemcpy(dX0, hX0, mem\_sizeX, cudaMemcpyHostToDevice);

    cudaMemcpy(dV0, hV0, mem\_sizeX, cudaMemcpyHostToDevice);

    // Настройка параметров запуска ядер

    dim3 nTid\_L(256);

    dim3 nBlk\_L((N + nTid\_L.x - 1) / nTid\_L.x);

    dim3 nTid\_MtrxA(16, 16);

    dim3 nBlk\_MtrxA((N + nTid\_MtrxA.x - 1) / nTid\_MtrxA.x,

                    (N + nTid\_MtrxA.y - 1) / nTid\_MtrxA.y);

    cudaEventRecord(start, 0);

    eps\_G = 1.0;

    while (eps\_G > EPS\_G) {

        Matrix\_A <<<nBlk\_MtrxA, nTid\_MtrxA>>> (dA, dX0, N);

        AX <<<nBlk\_L, nTid\_L>>> (dAX, dA, dX0, N);

        PHI <<<nBlk\_L, nTid\_L>>> (dPhi, dAX, dF, N);

        D\_PHI <<<nBlk\_MtrxA, nTid\_MtrxA>>> (dL, dX0, N);

        cudaMemset(dV0, 1, mem\_sizeX);

        eps\_L = 1.0;

        while (eps\_L > EPS\_L) {

            Solve\_L <<<nBlk\_L, nTid\_L>>> (dL, dPhi, dV0, dV1, N);

            Eps\_L <<<nBlk\_L, nTid\_L>>> (dV0, dV1, ddV, N);

            cudaMemcpy(hdV, ddV, mem\_sizeX, cudaMemcpyDeviceToHost);

            eps\_L = 0.0;

            for (j = 0; j < N; j++) eps\_L += hdV[j];

            eps\_L /= N;

        }

        Solve\_G <<<nBlk\_L, nTid\_L>>> (dX0, dX1, dV0, TAU, N);

        Eps\_G <<<nBlk\_L, nTid\_L>>> (dX0, dX1, ddX, N);

        cudaMemcpy(hdX, ddX, mem\_sizeX, cudaMemcpyDeviceToHost);

        eps\_G = 0.0;

        for (k = 0; k < N; k++) eps\_G += hdX[k];

        eps\_G /= N;

    }

    cudaEventRecord(stop, 0);

    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&gpu\_time, start, stop);

    printf("GPU calculation time\*: %.0f ms\n", gpu\_time);

    printf("Rate: %.1f x\n", cpu\_time / gpu\_time);

    printf("\*время выполнения с учетом копирования данных «host» - «device»\n");

    // Освобождение памяти

    free(hA); free(hF); free(hX0); free(hX1); free(hV0); free(hV1);

    free(hPhi); free(hAX); free(hL); free(hdX); free(hdV);

    cudaFree(dA); cudaFree(dF); cudaFree(dX0); cudaFree(dX1); cudaFree(dV0);

    cudaFree(dV1); cudaFree(dPhi); cudaFree(dAX); cudaFree(dL); cudaFree(ddX); cudaFree(ddV);

    cudaEventDestroy(start);

    cudaEventDestroy(stop);

    return 0;

}

