# Программные и аппаратные средства информатики

Лекции 7-8. Вычисления на видеокарте

- Аппаратные особенности GPU
- Вычислительная модель GPU
- Порядок установки и запуска приложений на CUDA
- Расширенный Си и файлы .cu
- Работа с памятью
- Указатели
- Обработка ошибок
- Пример приложения с графическим интерфейсом
- Особенности программирования на Си

Преподаватель курса: Нефедов Денис Геннадьевич, к.т.н., доцент

# Аппаратные особенности GPU

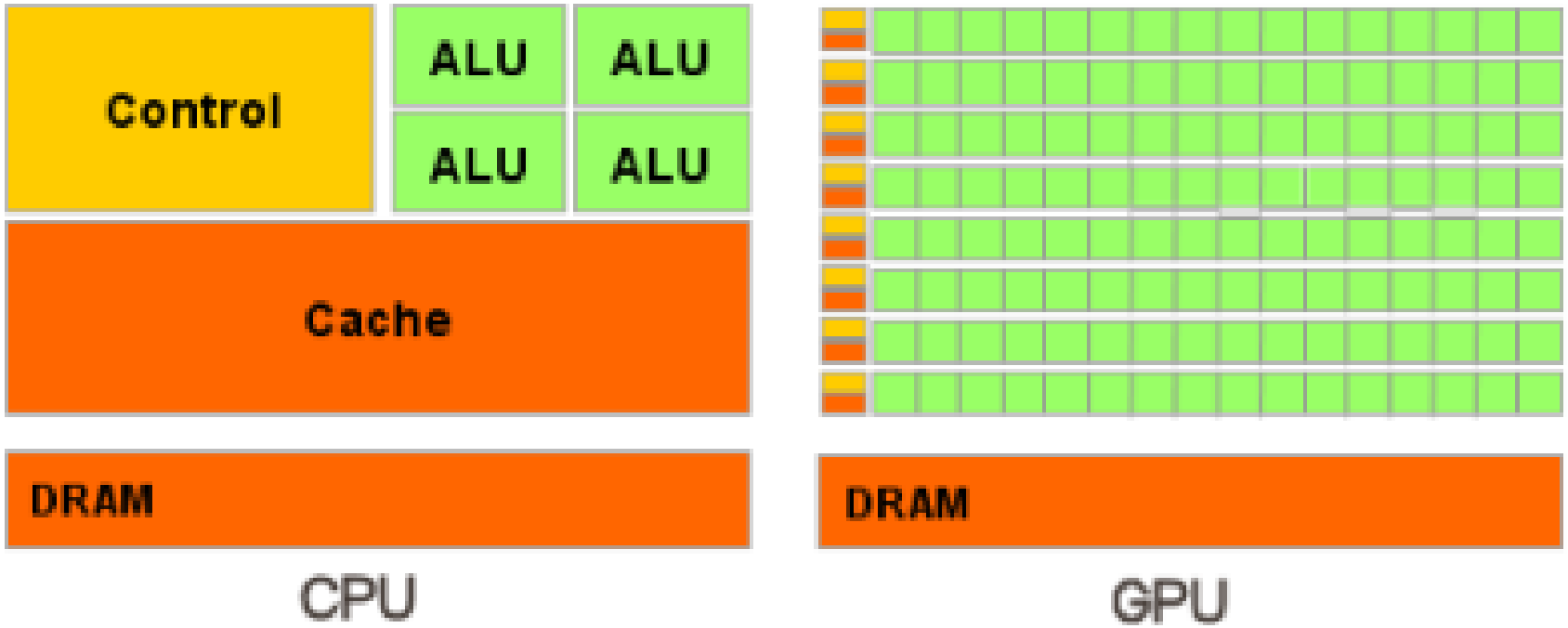



Рис. – Сравнение архитектурных особенностей CPU и GPU

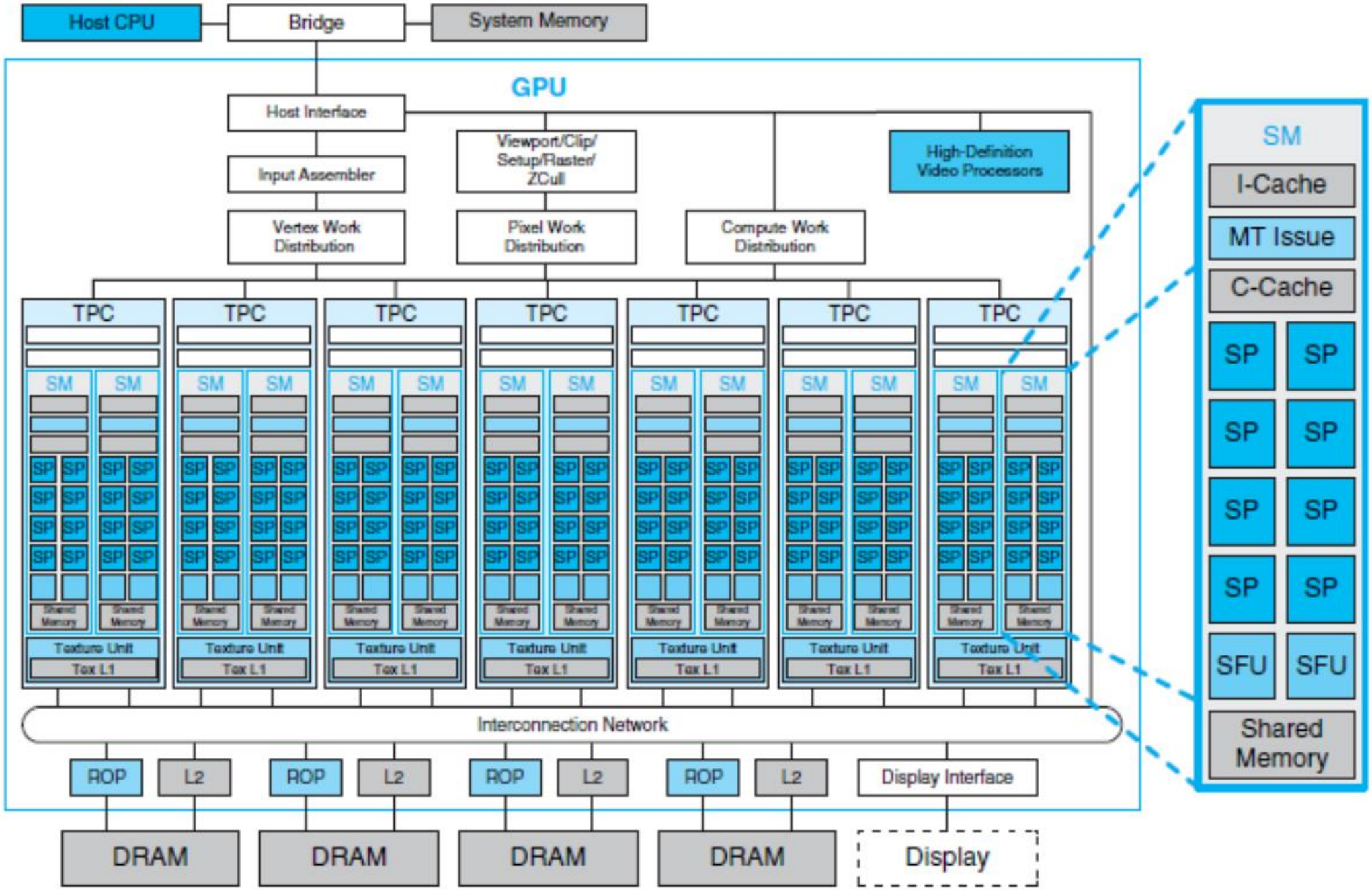



Рис. – Архитектура видеокарты GeForce 8800 GT

Центральный процессор обменивается данными с GPU через мост (Bridge). Вычислительная часть видеокарты состоит из семи текстурных процессорных кластеров (TPC), каждый из которых содержит два мультипроцессора (SM), блок текстур (Texture Unit) и совмещенный текстурный кэш и кэш первого уровня (Tex/L1).

Блоки текстур осуществляют выборку и фильтрацию текстурных данных, необходимых для построения сцены.

Мультипроцессор, содержит 8 потоковых процессоров (SP) или ядер CUDA (CUDA cores). Таким образом, общее число SP = 14 * 8 = 112. Каждый SM содержит по два блока для вычисления специальных (трансцендентных) функций (SFU), таких как sin, cos и т.д. В состав SM входит разделяемая память (Shared Memory), блок инструкций нитей (Multithreaded Instruction Unit), константный кэш (C-Cache) и кэш инструкций (L-Cache).

Вычислительная часть устройства соединяется с видеопамятью (DRAM) посредством кэша второго уровня (L2) и блоков операций растеризации (ROP). Блоки операций растеризации осуществляют операции записи рассчитанных видеокартой пикселей в буферы и операции их смешивания (блендинга). Производительность блоков ROP влияет на филлрейт (скорость заполнения пикселами).


Рис. – Архитектура мультипроцессора семейства Volta GV100

Каждый SM содержит четыре блока, в которых есть помимо ядер выполняющих целочисленные вычисления, ядра для работы с вещественными 32 и 64-битными числами, а также новые, появившиеся в данной линейке процессоров, тензорные ядра (Tensor Cores), разработанные специально для обучения нейросетей в задачах глубокого обучения.

Таблица – Характеристики видеокарт

| Название | GPU / кристалл | Кол-во процессоров-ядер / текстурных блоков / ROP - блоков растеризации | Частота ядра GPU / шейдерного блока / памяти в MHz | Разрядность шины памяти | Версия CUDA Capability |
|---|---|---|---|---|---|
| RTX 3090 24gb | GA102 | 10496/328/112+82 RT ядра | 1395/1995/19500 | 384 bit | 8.6 |
| GTX 1080 8gb | GP104 | 2560/160/64 | 1607/1733/10000 | 256 bit | 6.1 |
| GTX 780 Ti 3gb | GK110 | 2880/240/48 | 875/928/7000 | 384 bit | 3.5 |
| GT 630 1 gb | GF108 | 96/16/4 | 810/1620/3200 | 128 bit | 2.1 |
| 9500 GT 1 gb | G96 | 32/16/8 | 550/1400/1600 | 128 bit | 1.1 |

**Вычислительная производительность** (CUDA Compute Capability) – определяет набор функций, доступных для устройства. Выражается парой чисел - major.minor (глобальная архитектурная версия.небольшие изменения)

| Версия CUDA | Мин. CC | CC по умолчанию | Макс. CC |
|---|---|---|---|
| 5.5 | 1.0 | 1.0 | |
| 8,0 | 2.0 | 2.0 | 6.2 |
| 10.x | 3.0 | 3.0 | 7.5 |
| 11.x | 3.5 | 5.2 | 8.6 |

1.1 – атомарные операции над 32-битными целыми числами и atomicExch() над 32-битными числами с плавающей точкой в глобальной памяти;
1.2 – тоже в разделяемой памяти, также в глобальной над 64-разрядными целыми, функции голосования в варпе (warp vote functions);
1.3 – операции над числами с плавающей точкой двойной точности;
2.0 – атомарные операции над 64-битными целыми числами в разделяемой памяти и атомарное сложение 32-битных чисел с плавающей точкой, функции _ballot(), _threadfence_system(), _syncthreads_count(), _syncthreads_and(), _syncthreads_or(), поверхностные функции, трехмерная сетка блоков;
3.0 – функции варп-перемешивания (warp shuffle functions);
3.2 – сдвиги __funnelshift_lc() и __funnelshift_rc() (funnel shift);
3.5 – динамический параллелизм;
5.3 – операции над числами половинной точности;
6.0 - атомарное сложение 64-битных чисел с плавающей точкой в глобальной и разделяемой памяти

# Вычислительная модель GPU

**CUDA** (Compute Unified Device Architecture) - это технология от компании NVidia, предназначенная для разработки приложений для массивно-параллельных вычислительных устройств (для GPU, начиная с серии G80).

CUDA работает только с устройствами производства NVIDIA.

Аналогичные технологии: OpenCL, AMD FireStream, CUDAfy и PyCUDA.

**Хост** (host) – центральный процессор, CPU.

**Устройство** (device) – графический процессор, расположенный на видеокарте GPU.

**Ядро** (kernel) – функция, которая соответствует задаче, запускаемой на устройстве.

CUDA поддерживают языки программирования: **C/C++, Fortran** и **Python**.

В программу, написанную для CPU, можно добавить дополнительные элементы для работы с GPU.

Основные этапы выполнения работы в CUDA-программах:

1. Выделяется память под данные на устройстве;
2. Копируются данные из памяти хоста в память устройства;
3. На устройстве выполняются ядра;
4. Результаты пересылаются из памяти устройства в память хоста.

Особенности вычислительной модели GPU:

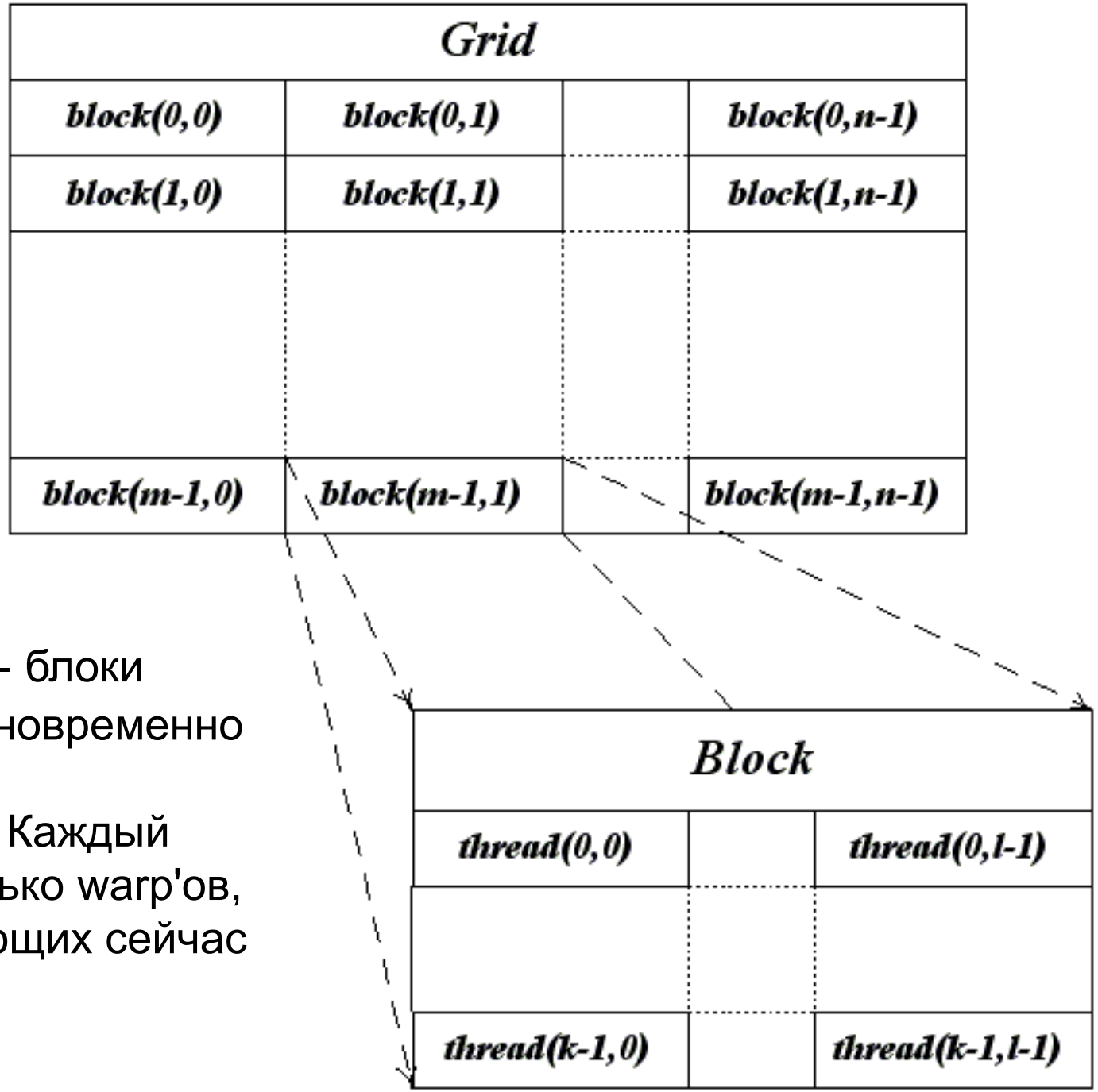
- является сопроцессором к CPU
- обладает собственной памятью (DRAM)
- обладает возможностью параллельного выполнения огромного количества отдельных нитей (threads)
- программы пишутся на "расширенном" C, при этом их параллельная часть (ядра) выполняется на GPU, а обычная часть - на CPU. CUDA автоматически осуществляет разделением частей и управлением их запуском

CUDA использует большое число отдельных нитей для вычислений, часто каждому вычисляемому элементу соответствует одна нить.

Все нити группируются в иерархию - grid/block/thread.

Grid - соответствует ядру и представляет одномерный или двухмерный массив блоков (block).

Каждый блок (block) представляет одно/двух/трехмерный массив нитей (threads).

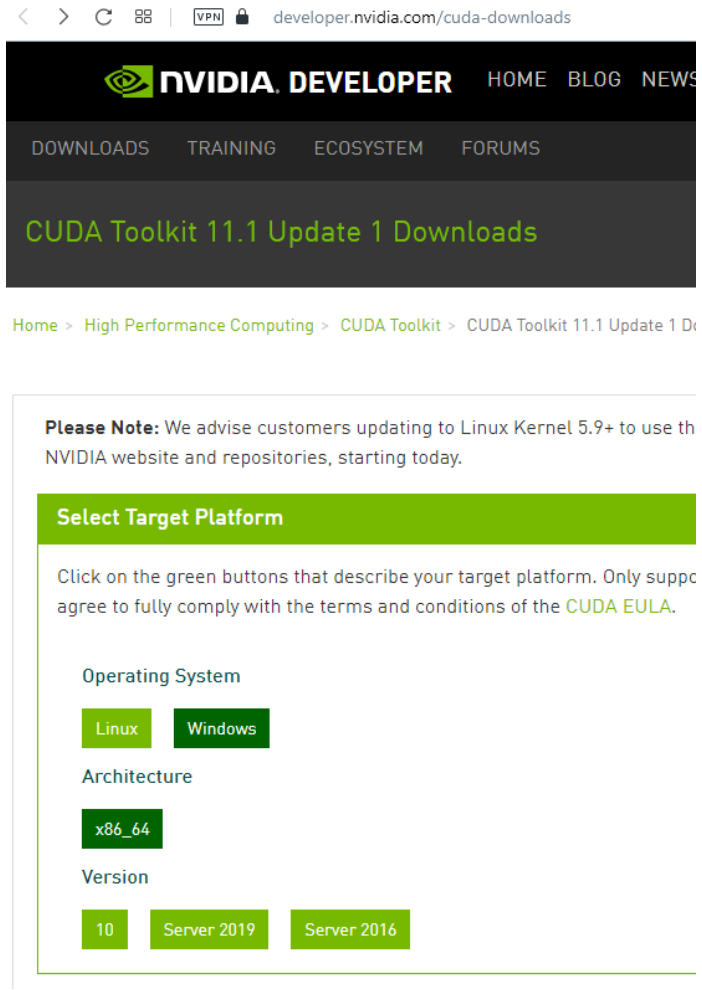


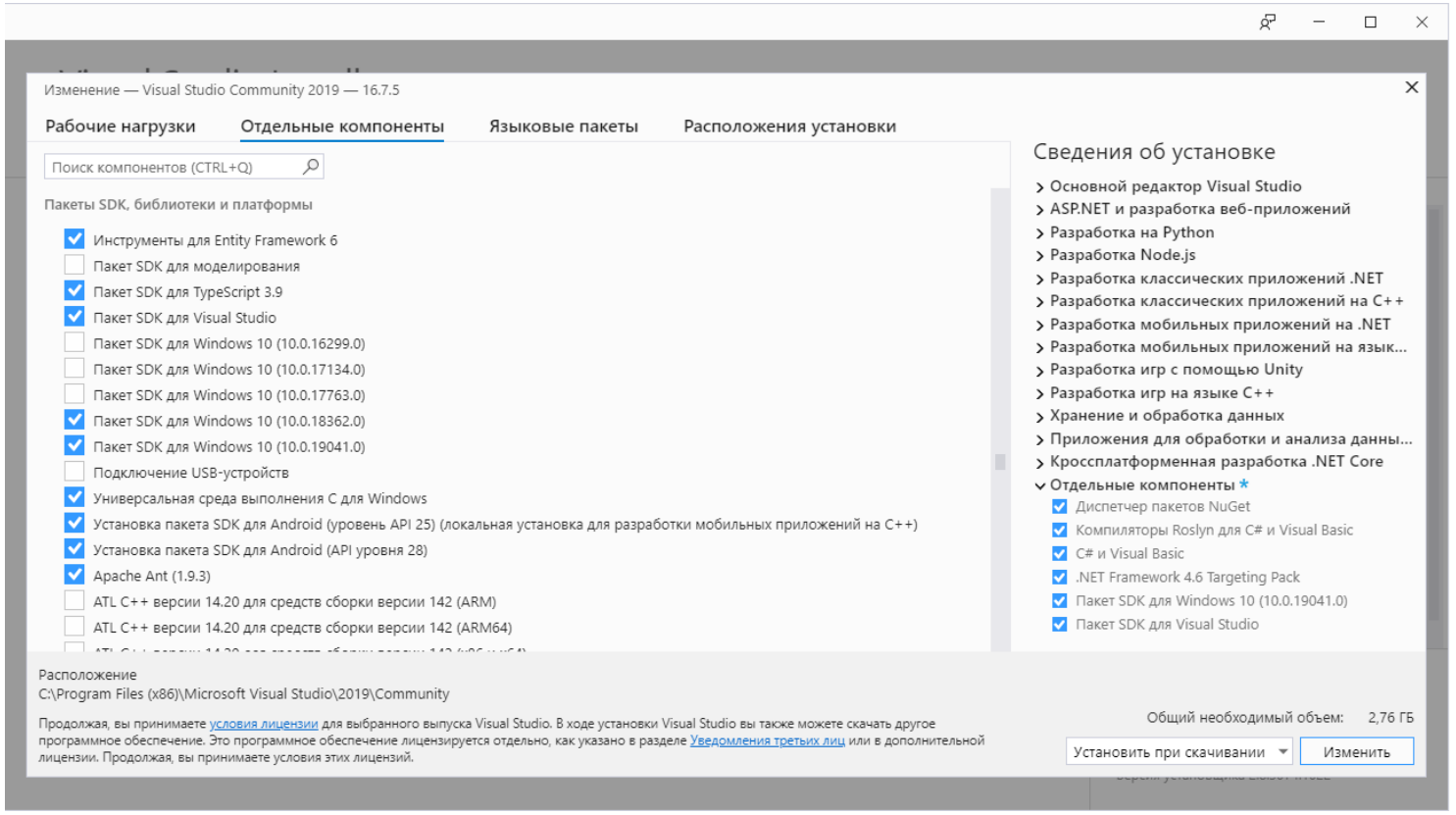

Все нити разбиваются на warp'ы - блоки подряд идущих нитей, которые одновременно (физически) выполняются и могут взаимодействовать друг с другом. Каждый блок нитей разбивается на несколько warp'ов, размер warp'а для всех существующих сейчас GPU равен 32.

# Порядок установки и запуска приложений на CUDA

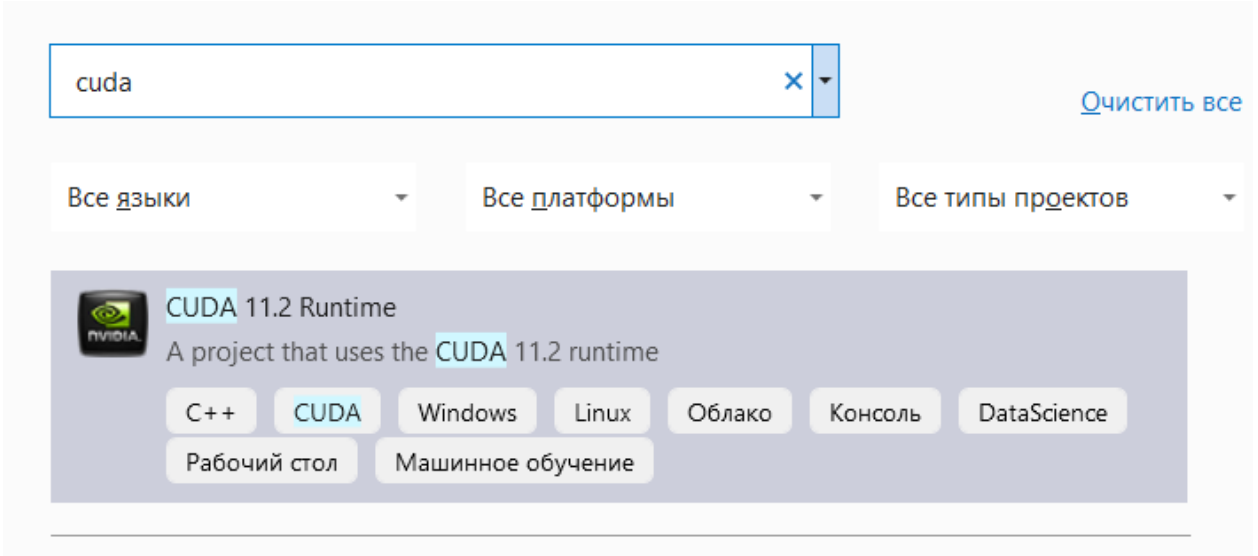
Необходимые программно-аппаратные средства

1. GPU G80 и новее (с 2007).
2. CUDA Toolkit (включает Display Driver) и SDK для Visual Studio

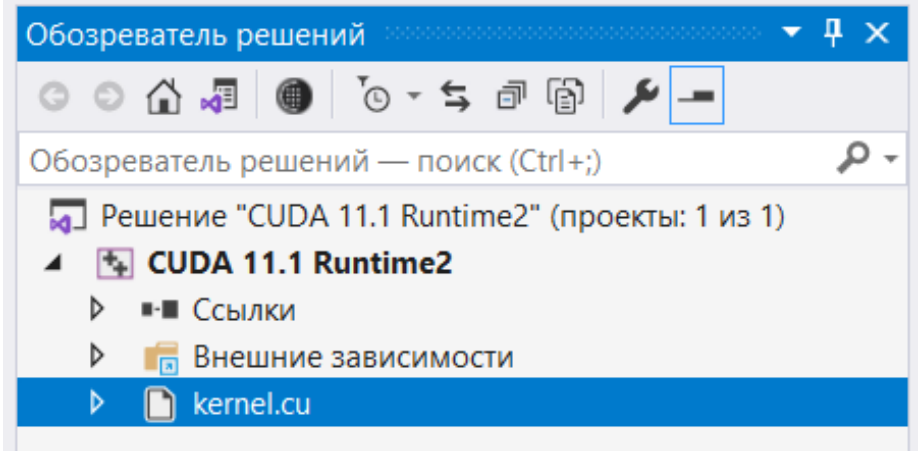


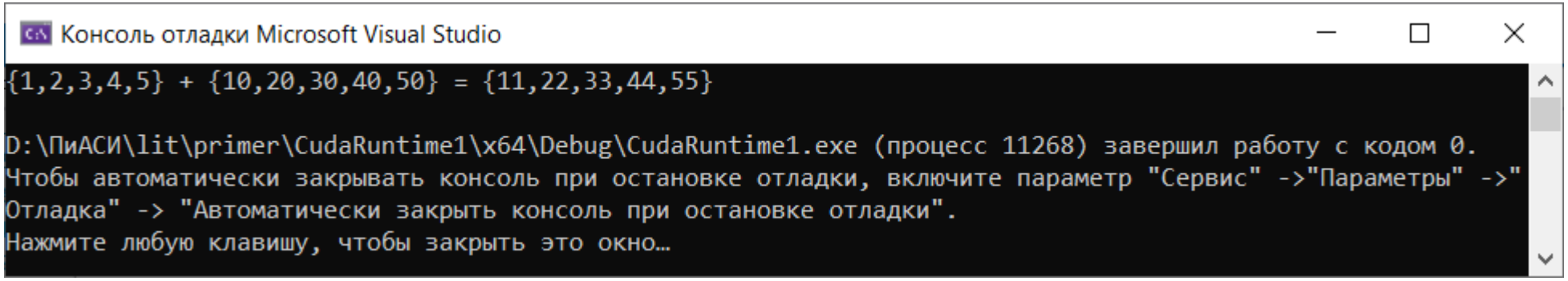
3. Создание проекта CUDA 11.x Runtime




4. Редактирование файла с расширением .cu




5. Запуск приложения



CUDA host API является связующим звеном между CPU и GPU. Подразделяется на низкоуровневое API под названием **CUDA driver API**, который предоставляет доступ к драйверу пользовательского режима CUDA, и высокоуровневое API – **CUDA runtime API**.

В CUDA runtime API входят следующие группы функций:

- Device Management – включает функции для общего управления GPU (информация о возможностях GPU, переключение между GPU при работе SLI-режиме и т.д.).
- Thread Management – управление нитями.
- Stream Management – управление потоками.
- Event Management – функция создания и управления event'ами.
- Execution Control – функции запуска и исполнения ядра CUDA.
- Memory Management – функции управлению памятью GPU.
- Texture Reference Manager – работа с объектами текстур через CUDA.
- OpenGL Interoperability – функции по взаимодействию с OpenGL API.
- Direct3D 9 Interoperability – функции по взаимодействию с Direct3D 9 API.
- Direct3D 10 Interoperability – функции по взаимодействию с Direct3D 10 API.
- Error Handling – функции обработки ошибок.

Так, информация о возможностях GPU возвращается в виде структуры cudaDeviceProp.

```
struct cudaDeviceProp
{
    char   name[256];
    size_t totalGlobalMem;
    size_t sharedMemPerBlock;
    int    regsPerBlock;
    int    warpSize;
    size_t memPitch;
    int    maxThreadsPerBlock;
    int    maxThreadsDim [3];
    int    maxGridSize   [3];
    size_t totalConstMem;
    int    major;
    int    minor;
    int    clockRate;
    size_t textureAlignment;
    int    deviceOverlap;
    int    multiProcessorCount;
}
```

```cpp
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
int main()
{
    int             deviceCount;
    cudaDeviceProp devProp;
    cudaGetDeviceCount(&deviceCount);
    printf("Found %d devices\n", deviceCount);
    for (int device = 0; device < deviceCount; device++)
    {
        cudaGetDeviceProperties(&devProp, device);
        printf("Device %d\n", device);
        printf("Compute capability     : %d.%d\n", devProp.major, devProp.minor);
        printf("Name                   : %s\n", devProp.name);
        printf("Total Global Memory    : %d\n", devProp.totalGlobalMem);
        printf("Shared memory per block: %d\n", devProp.sharedMemPerBlock);
        printf("Registers per block    : %d\n", devProp.regsPerBlock);
        printf("Warp size              : %d\n", devProp.warpSize);
        printf("Max threads per block  : %d\n", devProp.maxThreadsPerBlock);
        printf("Total constant memory  : %d\n", devProp.totalConstMem);
    }
    return 0;
}
```

```
Found 1 devices
Device 0
Compute capability     : 7.5
Name                   : GeForce GTX 1650
Total Global Memory    : 0
Shared memory per block: 49152
Registers per block    : 65536
Warp size              : 32
Max threads per block  : 1024
Total constant memory  : 65536
```

# Расширенный Си и файлы .cu

Программы для CUDA (соответствующие файлы обычно имеют расширение .cu) пишутся на "расширенном" С и компилируются при помощи команды nvcc.

Вводимые в CUDA расширения языка С состоят из

1. Спецификаторов функций, показывающих где будет выполняться функция и откуда она может быть вызвана
2. Спецификаторы переменных, задающие тип памяти, используемый для данной переменной
3. Директива, служащая для запуска ядра, задающая как данные, так и иерархию нитей
4. Встроенные переменные, содержащие информацию о текущей нити runtime, включающие в себя дополнительные типы данных

1. **Спецификаторы функций** показывают, где будет выполняться функция и откуда она может быть вызвана

| Спецификатор | Выполняется на | Может вызываться из |
|---|---|---|
| __device__ | device | device |
| __global__ | device | host |
| __host__ | host | host |

Спецификатор __global__ обозначает ядро и соответствующая функция должна возвращать значение типа void.

```
__global__ void myKernel ( float * a, float * b, float * c )
{
   int index = threadIdx.x;
   c [i] = a [i] * b [i];
}
```

На функции, выполняемые на GPU (__device__ и __global__) накладываются ограничения:

- нельзя брать их адрес (за исключением __global__ функций)
- не поддерживается рекурсия
- не поддерживаются static-переменные внутри функции
- не поддерживается переменное число входных аргументов

Для задания размещения в памяти GPU переменных используются следующие спецификаторы - __device__, __constant__ и __shared__. На их использование также накладывается ряд ограничений:

- эти спецификаторы не могут быть применены к полям структуры (struct или union)
- соответствующие переменные могут использоваться только в пределах одного файла, их нельзя объявлять как extern
- запись в переменные типа __constant__ может осуществляться только CPU при помощи специальных функций
- __shared__ переменные не могут инициализироваться при объявлении

## 2. Добавленные переменные

В язык добавлены следующие специальные переменные:

- **gridDim** - размер grid'a (имеет тип dim3)
- **blockDim** - размер блока (имеет тип dim3)
- **blockIdx** - индекс текущего блока в grid'e (имеет тип uint3)
- **threadIdx** - индекс текущей нити в блоке (имеет тип uint3)
- **warpSize** - размер warp'a (имеет тип int)

Добавленные типы

В язык добавляются 1/2/3/4-мерные вектора из базовых типов - char1, char2, char3, char4, uchar1, uchar2, uchar3, uchar4, short1, short2, short3, short4, ushort1, ushort2, ushort3, ushort4, int1, int2, int3, int4, uint1, uint2, uint3, uint4, long1, long2, long3, long4, ulong1, ulong2, ulong3, ulong4, float1, float2, float3, float2, и double2.

Обращение к компонентам вектора идет по именам - x, y, z и w. Для создания значений-векторов заданного типа служит конструкция вида make_<typeName>.

```
int2   a = make_int2   ( 1, 7 );
float3 u = make_float3 ( 1, 2, 3.4f );
```

## 3. Директива вызова ядра.

Для запуска ядра на GPU используется следующая конструкция:

kernelName <<<Dg,Db,Ns,S>>> ( args )

Здесь

- kernelName это имя (адрес) соответствующей __global__ функции,
- Dg - переменная (или значение) типа dim3, задающая размерность и размер grid'a (в блоках),
- Db - переменная (или значение) типа dim3, задающая размерность и размер блока (в нитях),
- Ns - переменная (или значение) типа size_t, задающая дополнительный объем shared-памяти, которая должна быть динамически выделена (к уже статически выделенной shared-памяти),
- S - переменная (или значение) типа cudaStream_t задает поток (CUDA stream), в котором должен произойти вызов, по умолчанию используется поток 0. Через args обозначены аргументы вызова функции kernelName.

Также в язык C добавлена функция __syncthreads, осуществляющая синхронизацию всех нитей блока. Управление из нее будет возвращено только тогда, когда все нити данного блока вызовут эту функцию. Т.е. когда весь код, идущий перед этим вызовом, уже выполнен (и, значит, на его результаты можно смело рассчитывать). Эта функция очень удобная для организации безконфликтной работы с shared-памятью.

Также CUDA поддерживает все математические функции из стандартной библиотеки C, однако с точки зрения быстродействия лучше использовать их float-аналоги (а не double) - например sinf. Кроме этого CUDA предоставляет дополнительный набор математических функций (__sinf, __powf и т.д.) обеспечивающий более низкую точность, но заметно более высокое быстродействие чем sinf, powf и т.п.

CUDA API для CPU (host) выступает в двух формах - низкоуровневый CUDA driver API и CUDA runtime API (реализованный через CUDA driver API). В своем приложении Вы можете использовать только один из них, далее мы рассмотрим CUDA runtime API, как более простой и удобный.

Все функции CUDA driver API начинаются с префикса cu, а все функции CUDA runtime API начинаются с префикса cuda. Каждый из этих API предоставляет основной набор базовых функций, таких как перебор всех доступных устройств (GPU), работа с контекстами и потоками, работа с памятью GPU, взаимодействие с OpenGL и D3D (поддерживается только 9-я версия DirectX).

# Работа с памятью

В CUDA для GPU существует несколько различных типов памяти, доступных нитям, сильно различающихся между собой:

| № | Тип памяти | Назначение (что хранит) | Доступ | Уровень выделения | Скорость работы |
|---|---|---|---|---|---|
| 1 | **Registers** | локальные переменные | чтение/запись | per-thread | высокая (on chip) |
| 2 | **Local** | локальные переменные | чтение/запись | per-thread | низкая (DRAM) |
| 3 | **Shared** | массив данных | чтение/запись | per-block | высокая (on-chip) |
| 4 | **Global** | все данные | чтение/запись | per-grid | низкая (DRAM) |
| 5 | **Constant** | передача параметров в ядро | чтение | per-grid | высокая (on chip L1 cache) |
| 6 | **Texture** | большой массив данных | чтение | per-grid | высокая (on chip L1 cache) |

CPU имеет R/W доступ только к глобальной, константной и текстурной памяти (находящейся в DRAM GPU) и только через функции копирования памяти между CPU и GPU (предоставляемые CUDA API).

1), 2) Регистровая и локальная виды памяти выделяются на стадии компиляции

4) Глобальная память – самая медленная. Функции для работы с этой памятью

| Назначение | Синтаксис |
|---|---|
| Выделение памяти | **cudaMalloc**( void **devPtr, size_t size); |
| Освобождение памяти | **cudaFree**( void **devPtr); |
| Копирование данных | **cudaMemcpy**( void *dst, const void *src, size_t size, enum cudaMemcpyKind kind) |

Последовательность работы с памятью

```
float* devPtr;          // указатель на память устройства
                        // выделяем линейную память для 256 floats
...
cudaMalloc((void**)&devPtr, 256 * sizeof(float));
...
cudaFree(devPtr);       // освобождаем память устройства
```

Кроме линейной есть также другой тип памяти - CUDA-arrays.

Для выделения памяти под двухмерные массивы чаще используется функция cudeMallocPitch, которая осуществляет выравнивание (путем добавления к каждой строке дополнительной) строк массива для более эффективного доступа к памяти. При этом в параметре pitch возвращается размер строки в байтах.

```
float* devPtr;  // указатель на память устройтва
int  pitch; // размер строк в bytes
            // выделить линейную память для width*height 2D array of floats
. . .
   cudaMallocPitch((void**)&devPtr, &pitch, width * sizeof(float), height);
. . .
   cudaFree(devPtr); // освобождаем память устройства
```

В приведенном фрагменте кода осуществляется выделение памяти на GPU под двухмерный массив из float'ов размером width и height. Элемент с индексом [col,row] будет находиться по смещению col*sizeof(float)+row*pitch.

Рассмотренные функции управляют выделением памяти на GPU, к которой CPU не имеет непосредственного доступа. Поэтому API предоставляет функции копирования памяти как между CPU и GPU, так и в пределах GPU.

```
cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, enum
cudaMemcpyKind kind);
    cudaError_t cudaMemcpyAsync(void* dst, const void* src, size_t count,
enum cudaMemcpyKind kind, cudaStream_t stream);
```

Здесь аргументы dst и src задают адреса куда и откуда необходимо произвести копирование памяти, параметр count задает количество байт памяти, которое необходимо переписать.

Параметр kind задает тип копирования памяти и принимает одно из следующих значений - cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost и cudaMemcpyDeviceToDevice.

Пример. Программа, складывающая 2 целых числа на GPU.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

// ядро
__global__ void add(int* a, int* b, int* c)
{
    *c = *a + *b;
}
//главная функция
int main()
{
    // переменные на CPU
    int a, b, c;
    // переменные на GPU
    int* dev_a, * dev_b, * dev_c;
    int size = sizeof(int); //размерность
    // выделяем память на GPU
    cudaMalloc((void**)&dev_a, size);
    cudaMalloc((void**)&dev_b, size);
    cudaMalloc((void**)&dev_c, size);
```

```
    // инициализация переменных
    a = 2;
    b = 7;
    // копирование информации с CPU на GPU
    cudaMemcpy(dev_a, &a, size, cudaMemcpyHostToDevice
    );
    cudaMemcpy(dev_b, &b, size, cudaMemcpyHostToDevice
    );
    // вызов ядра
    add << < 1, 1 >> > (dev_a, dev_b, dev_c);
    // копирование результата работы ядра с GPU на CPU
        cudaMemcpy(&c, dev_c, size, cudaMemcpyDeviceToHost
        );
    // вывод информации
    printf("%d + %d = %d\n", a, b, c);
    // очищение памяти на GPU
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);

    return 0;
}
```

```
2 + 7 = 9
```

# Обработка ошибок

Каждая функция CUDA API (кроме запуска ядра) возвращает значение типа cudaError_t. При успешном выполнении функции возвращается cudaSuccess, в противном случае возвращается код ошибки.

| Назначение | Синтаксис |
|---|---|
| Возвращает описание ошибки в виде строки | char* cudaGetErrorString(cudaError_t error); |
| Возвращает код последней ошибки | cudaError_t cudaGetLastError(); |

В силу асинхронности выполнения многих вызовов, для получения кода ошибки лучше использовать функцию cudaThreadSynchronize, которая дожидается завершения выполнения на GPU всех переданных запросов и возвращает ошибку, если один из этих запросов привел к ошибке.

cudaError_t cudaThreadSynchronize ();

# Примеры

1.
```
cudaError_t err = cudaGetLastError();
if (err != cudaSuccess) printf("%s ",  cudaGetErrorString(err));
```

2.
```
cudaError_t piWithCuda(double* c, unsigned int size);

cudaError_t piWithCuda(double* c, unsigned int size)
{
    double* dev_c = 0;
    cudaError_t cudaStatus;
    // Choose which GPU to run on, change this on a multi-GPU system.
    cudaStatus = cudaSetDevice(0);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaSetDevice failed!  Do you have a CUDA-capable GPU
installed?");
        goto Error;
    }
Error:
    cudaFree(dev_c);

    return cudaStatus;
}
```

Для отслеживания выполнения кода на GPU в CUDA используются event'ы.

Событие – это объект типа cudaEvent_t, используемый для обозначения «точки» среды вызовов CUDA.

| Функция | Параметры | Назначение |
|---|---|---|
| cudaEventCreate | cudaEvent_t * event | Создание event'ов |
| cudaEventRecord | cudaEvent_t event, CUstream stream | Задание места, прохождение которого должен сигнализировать данный event |
| cudaEventQuery | cudaEvent_t event | Мгновенная проверкуа "прохождения" данного event'а |
| cudaEventSynchronize | cudaEvent_t event | Явная синхронизация: ожидание пока все операции для данного event'а не будут завершены |
| cudaEventElapsedTime | float * time, cudaEvent_t startEvent, cudaEvent_t stopEvent | Можно узнать время в мс, прошедшее между данными event'ами |
| cudaEventDestroy | cudaEvent_t event | Уничтожение event'ов |

Пример кода, запускающего ядро на обработку данных и замеряющего время обработки

```
// инициализируем события
cudaEvent_t start, stop;
float elapsedTime;
// создаем события
cudaEventCreate(&start);
cudaEventCreate(&stop);
// запись события
cudaEventRecord(start, 0);
// вызов ядра
KernelKunction << <blocks, threads >> >;
cudaEventRecord(stop, 0);
// ожидание завершения работы ядра
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime, start, stop);
// вывод информации
printf("Time spent executing by the GPU: %.2f millseconds\n", elapsedTime);
// уничтожение события
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

# Пример приложения с графическим интерфейсом

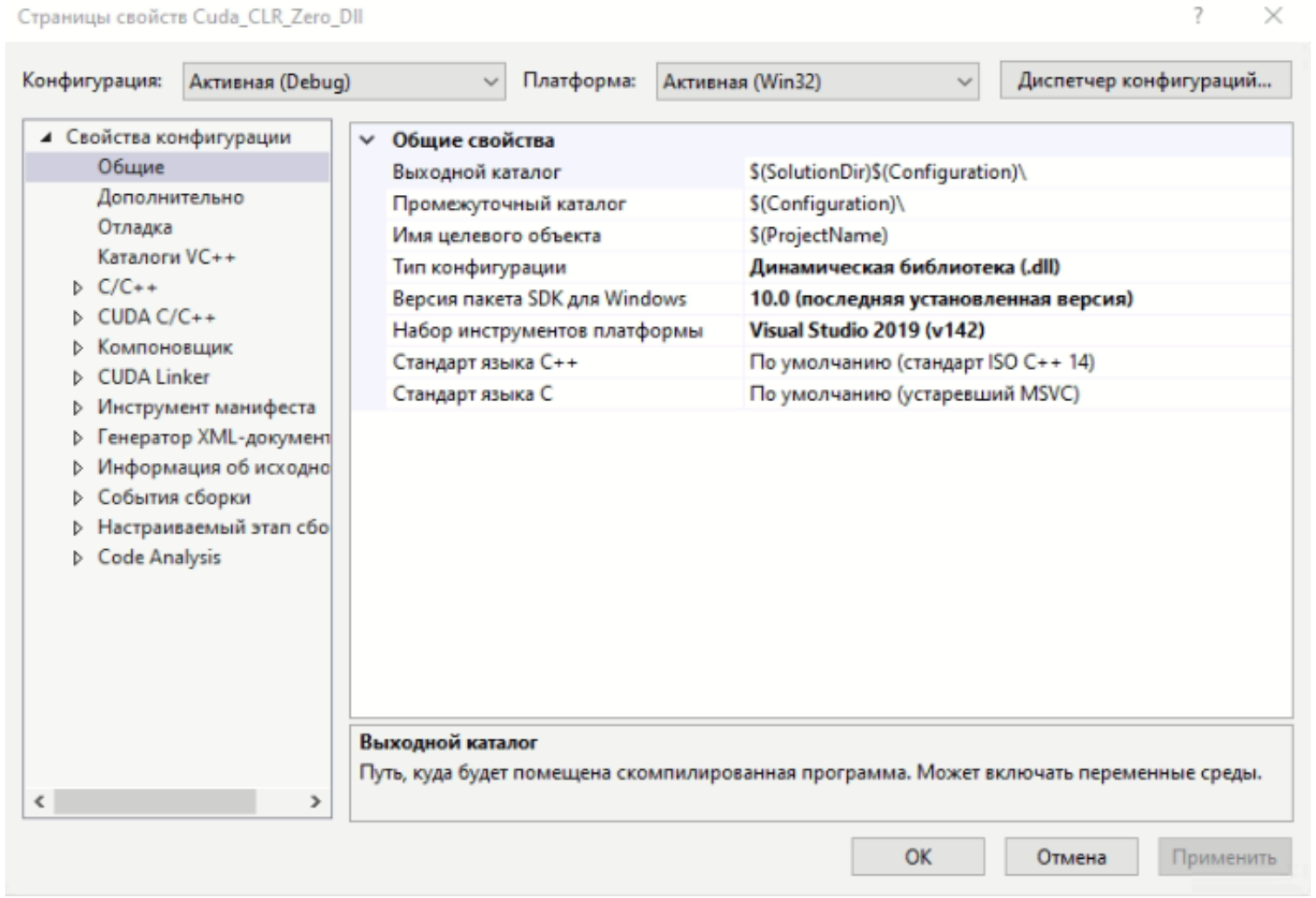
## 1. Создадим библиотеку на C++.

Для этого выбираем тип проекта Пустой, язык C++. Назовем Cuda_CLR_Zero_Dll.

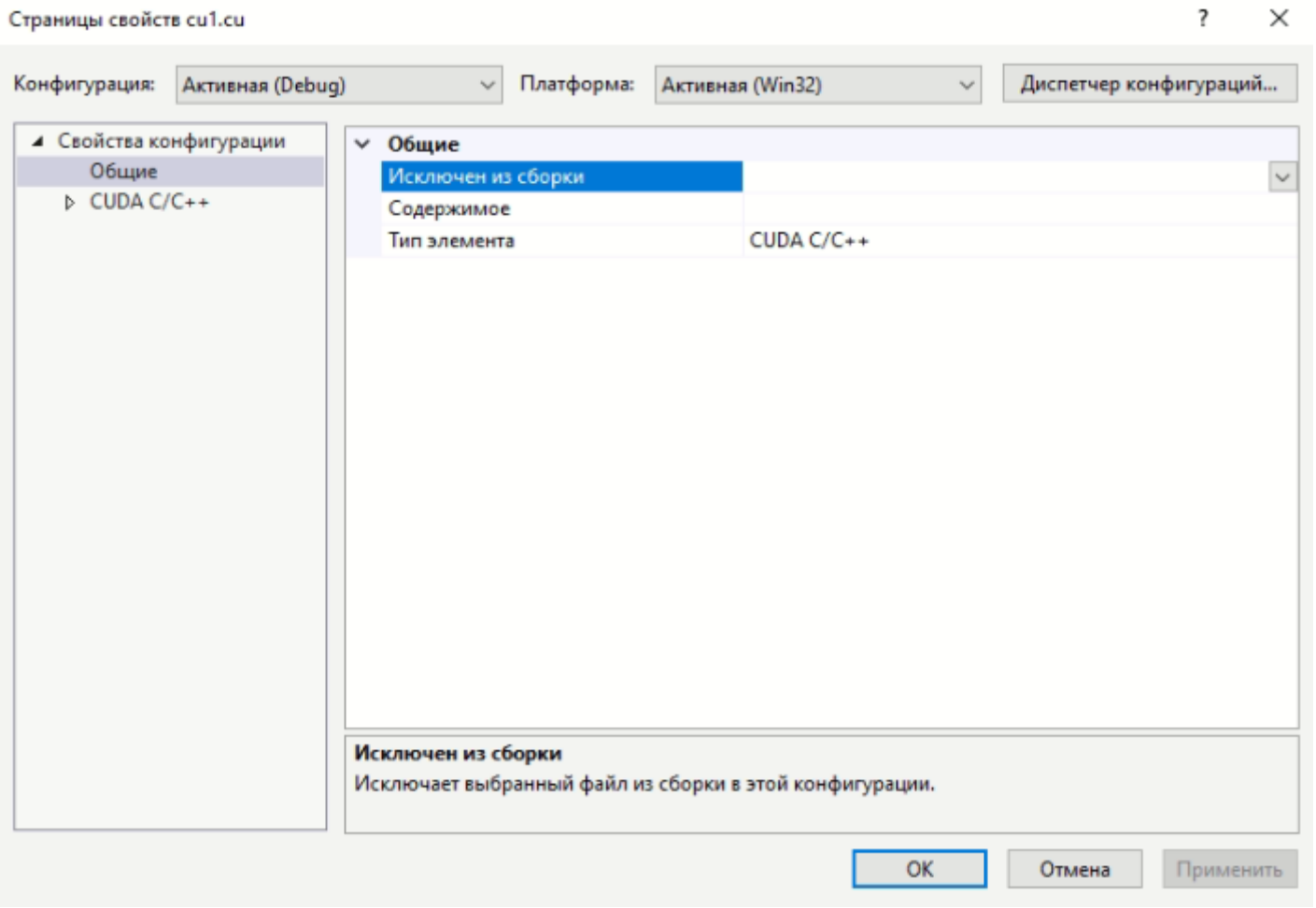
После создания проекта в его свойствах указываем

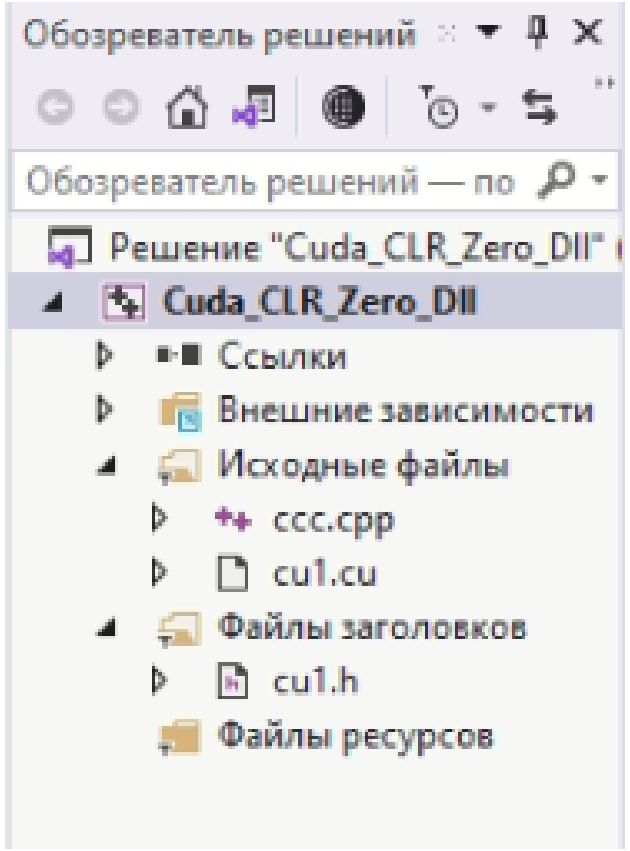
- Тип конфигурации в Динамическая библиотека
- и Поддержка общеязыковой среды в Поддержка CLR.

Также ПКМ на названии проекта, выбираем Настройки сборки и указываем CUDA 11.2



Добавляем в проект 3 файла с указанными названиями и расширениями. Для файла с расширением .cu тип элемента должен быть CUDA C/C++



Заполняем файлы содержимым

```
//  cu1.h
int Add2(char* s1,char* s2,char* s3,int * a,int * b, int * c,int d);
// ccc.cpp
#include "cu1.h"
extern "C"
 __declspec (dllexport) int  __cdecl   Add(char* s1,char* s2,char* s3,int * a,
int * b, int * c,int d)
   { return  Add2(s1,s2,s3, a, b, c, d) ;}

//  cu1.cu
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size)
;
__global__ void addKernel(int *c, const int *a, const int *b)
{
int i = threadIdx.x;
c[i] = a[i] + b[i] ;
}
```

```
////////////////////////////////////////////////////////
 int   Add2(char* s1,char* s2,char* s3,int * aa,int * bb, int * cc,int dd){
 char* s11;char* s22;char* s33;
 s11=s1; s22=s2;s33=s3;
  while(*s11){*s33=*s11;s33++;s11++;}
  while(*s22){*s33=*s22;s33++;s22++;}
  *s33=0;
  cudaError_t cudaStatus = addWithCuda(cc, aa, bb, dd );
  cudaStatus = cudaDeviceReset();
  return 0;
  };
 ////////////////////////////////////////////////////////
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size)
{
int *dev_a = 0;
int *dev_b = 0;
int *dev_c = 0;
cudaError_t cudaStatus;
```

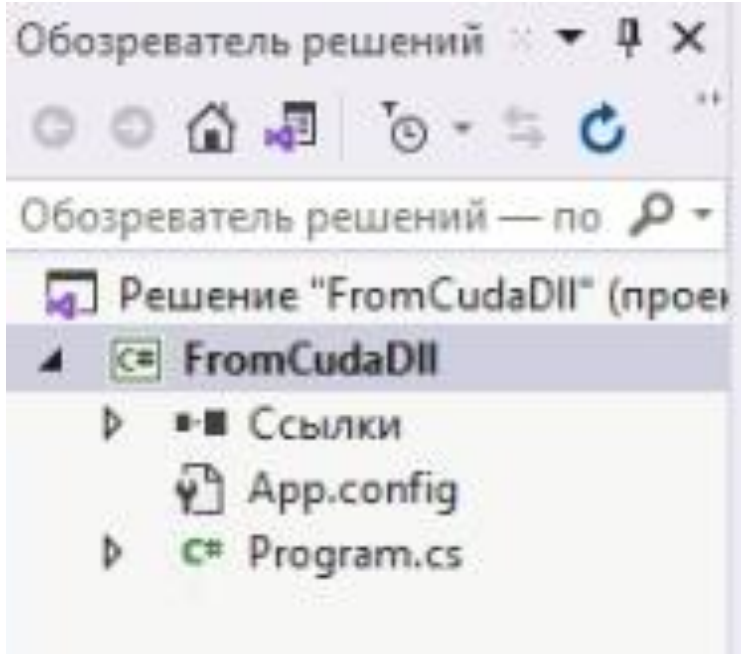
```
cudaStatus = cudaSetDevice(0);
cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

addKernel<<<1, size>>>(dev_c, dev_a, dev_b);
cudaStatus = cudaDeviceSynchronize();
cudaStatus = cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
cudaFree(dev_c); cudaFree(dev_a); cudaFree(dev_b);
return cudaStatus;
}
```

В главном меню вызываем Сборка/Собрать решение, что приведет к построению библиотеки. Динамическая библиотека Cuda_CLR_Zero_Dll.dll расположится в подпапке Debug или Release в зависимости от того строился отладочный или окончательный вариант и готова к использованию

## 2. Создание основной исполняемой программы

Для этого выбираем тип проекта Пустой, язык C#. Назовем FromCudaDll.
Добавим в проект элемент типа Файл с текстом программы, назовем его Program.




В Обозревателе решений добавился пункт Program.cs, щелкаем по нему мышкой и вводим текст программы.

```
using System;
using System.Windows.Forms;
using System.Reflection;
using System.Drawing;
using System.Runtime.InteropServices;
using System.Text;
public class Program
{
[DllImport("Cuda_CLR_Zero_Dll.dll", CallingConvention = CallingConvention.Cde
cl)]
unsafe public extern static System.Int32 Add(char* s1,char* s2,char* s3,int*
a, int* b, int* c, int d);
static ListBox listBox1;
//////////////////////////////////////////////////////////////

public static void Main()
{
  Form my_form = new Form1();
  Application.Run(my_form);
}
```

```
public class Form1 : Form
{
  public Form1()
  {
   InitializeComponent();
  }
  void InitializeComponent()
  {
   MetodsButton MB1 = new MetodsButton();
   MB1.InicializeButton(this, 1, "Незанятая кнопка");
   MB1.InicializeButton(this, 2, "Сумма векторов");

   listBox1 = new ListBox(); listBox1.Size = new System.Drawing.Size(400, 400)
;
   listBox1.Location = new System.Drawing.Point(150, 10);
   this.Controls.Add(listBox1);
  }
  public void Button1(object sender, EventArgs e) { }
  public void Button2(object sender, EventArgs e) { Test1(); }
}
```

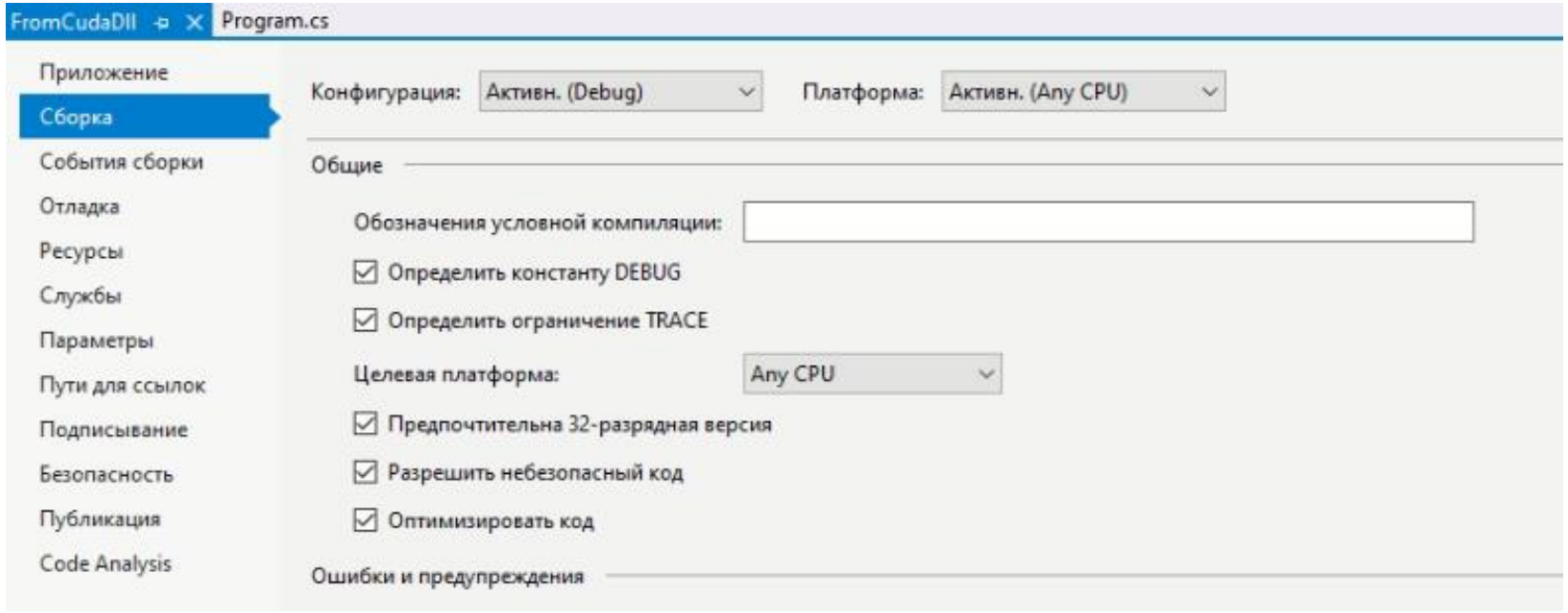
```
public class MetodsButton
{
  delegate void my_delegate_button_type(object sender, EventArgs e);
  MethodInfo buttons_metod_info;
  public void InicializeButton(System.Windows.Forms.Form My_Form1, int i, str
ing name)
  {
   Button My_Button = new System.Windows.Forms.Button();
   My_Button.Location = new Point(0, 27 * (i - 1));
   My_Button.Text = name;
   My_Button.Width = 140;
   My_Form1.Controls.Add(My_Button);
   string Button_Number = "Button";
   Button_Number += i.ToString();
   buttons_metod_info = (typeof(Form1)).GetMethod(Button_Number);
   Delegate My_Event = Delegate.CreateDelegate(typeof(my_delegate_button_type
), null, buttons_metod_info);
  My_Button.Click += new EventHandler((my_delegate_button_type)My_Event);
  }
}// public class MetodsButton
```

```
unsafe static void Test1()
{
  int[] a = new int[10]; a[0] = 10; a[1] = 20; a[2] = 30; a[3] = 40; a[4] = 50;
  int[] b = { 10, 20, 30, 40, 50 };
  int[] c = new int[10];
  string str1 = "Сумма ";
  string str2 = "векторов: ";
  char[] str3 = new char[255];
  string str4;
  IntPtr stringPointer1 = (IntPtr)Marshal.StringToHGlobalAnsi(str1);
  fixed (int* pa = a, pb = b, pc = c)
  fixed (char* s1=str1,s2=str2,s3=str3)
  {
   char* stringPointer2 = (char *)Marshal.StringToHGlobalAnsi(str2);
   int i = Add((char *)stringPointer1,stringPointer2,s3, pa, pb, pc, 5);
   str4 =c[0].ToString() + ", " + c[1].ToString() + ", " + c[2].ToString() + ", "
+ c[3].ToString() + ", " + c[4].ToString() ;
   String myString = Marshal.PtrToStringAnsi((IntPtr)s3);
   listBox1.Items.Add(myString + str4);
  }
}
}
```

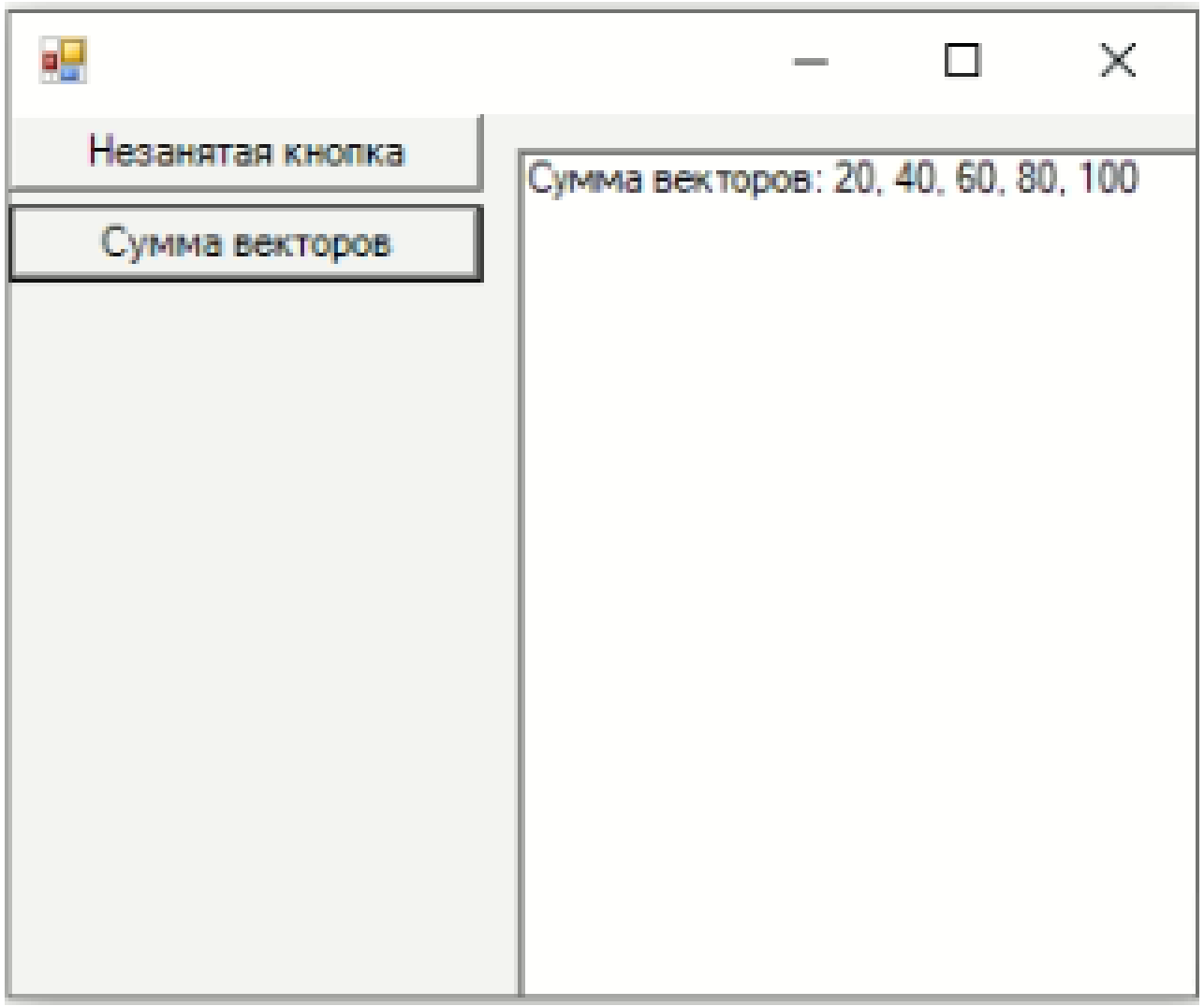
В обозревателе решений нажимаем Ссылки, выбираем пункт Добавить ссылку и в раскрывшемся окне вызываем меню .NET.

Поочередно добавим к проекту ссылки на компоненты System, System.Windows.Forms, System.Drawing. Далее, в том же окне выбираем меню Обзор, вспоминаем где расположена динамическая библиотека и пройдя путь до нее, также добавим к проекту файл Cuda_CLR_Zero_Dll.dll.

В приложении присутствует ключевое слова unsafe, что означает применение небезопасного кода для работы с указателями. Поэтому в Обозревателе решений ПКМ выбираем имя проекта, нажимам  пункт Свойства, в раскрывшемся окне выбираем пункт Сборка и ставим галочку в опции Разрешить небезопасный код.



В главном меню выбираем пункт Отладка и подпункт Начать отладку. После построения появляется пользовательская программа, все что остается, так это нажать клавишу Сумма векторов и в окне прочитать результат сложения a и b.



При возникновении ошибки

```
Двухэтапный поиск по имени не поддерживается для C++/CLI и
C++/CX; используйте /Zc:twoPhase-
```

- Открыть project's Property Pages диалог.
- Выбрать Configration Properties > C/C++ > Command Line property.
- Изменить Additional Options property добавив туда /Zc:twoPhase- и нажать OK.

## Ограничения и возможности при работе с GPU

Ограничения

- Если мы выполняем расчет на GPU, то не можем выделить только одно ядро, выделен будет целый блок ядер (32 для NVIDIA).
- Все ядра выполняют одни и те же инструкции, но с разными данными, такие вычисления называются Single-Instruction-Multiple-Data или SIMD.
- Из-за относительно простого набора логических блоков и общих регистров, GPU очень не любит ветвлений, да и в целом сложной логики в алгоритмах. Не поддерживается рекурсия.

Возможности:

- Собственно, ускорение тех самых SIMD-вычислений.

# Работа с разделяемой памятью

**Разделяемая память** обладает высокой, по сравнению с глобальной памятью, скоростью. Основной особенностью разделяемой памяти, является то, что она оптимизирована под обращение к ней одновременно половины количества нитей в блоке (16 нитей). Вся разделяемая память разбита на 32 банка, состоящих из 32 битовых слов, причем, последовательно идущие слова попадают в последовательно идущие блоки. Обращение к каждому банку происходит независимо. Второй важной особенностью разделяемой памяти является то, что она выделяется на блок.

Для задания разделяемой памяти используется спецификатор __shared__.

Часто использование разделяемой памяти требует явной синхронизации внутри блока. При работе с разделяемой памятью необходимо обеспечить синхронизацию потоков, что в CUDA довольно просто – она предоставляет примитивный барьер, функцию __syncthreads().

При ее вызове потоки приостанавливаются до того момента, пока все потоки не достигнут этой точки. Важно, чтобы вызов __syncthreads() у всех потоков находился в одном и том же месте (в частности, был безусловным, т. е. не находился внутри оператора if и т.п.), в противном случае результаты могут быть непредсказуемыми, в частности, может возникнуть "мертвая блокировка" (deadlock).

# Пример. Программа для вычисления скалярного произведения векторов.

```
__global__ void scalMult(const BASE_TYPE* A, const BASE_TYPE* B, BASE_TYPE* C, int numElem)
{
    // Переменная для хранения суммы элементов
    BASE_TYPE sum = 0.0;
    // Создание массивов в разделяемой памяти
    __shared__ BASE_TYPE ash[BLOCK_SIZE];
    __shared__ BASE_TYPE bsh[BLOCK_SIZE];
    // Копирование из глобальной памяти
    ash[threadIdx.x] = A[blockIdx.x * blockDim.x + threadIdx.x];
    bsh[threadIdx.x] = B[blockIdx.x * blockDim.x + threadIdx.x];
    // Синхронизация нитей
    __syncthreads();
    // Вычисление скалярного произведения
    if (threadIdx.x == 0)
    {
        sum = 0.0;
        for (int j = 0; j < blockDim.x; j++)
            sum += ash[j] * bsh[j];
        C[blockIdx.x] = sum;
    }
}
```