32. Задание ядра, потоки и блоки потоков на примере перемножения двух матриц в CUDA

```
// размерность решетки блоков при запуске ядра
dim3 gridDim;
// координаты текущего блока внутри решетки блоков
uint3 blockIdx;
// размерность блока при запуске ядра
dim3 blockDim;
// координаты текущей нити внутри блока
uint3 threadIdx;
```

Пример перемножения матриц с моделью куды:

```
#include <math.h>
#include <iostream>
#include "cuda_runtime.h"
#include "kernel.h"
#include <stdlib.h>
using namespace std;

__global__ void matrixMultiplicationKernel(float* A, float* B,
float* C, int N)
{
int ROW = blockIdx.y * blockDim.y + threadIdx.y;
int COL = blockIdx.x * blockDim.x + threadIdx.x;
float tmpSum = 0;
if (ROW < N && COL < N) {
 // each thread computes one element of the block sub-matrix
  for (int i = 0; i < N; i++) {
     tmpSum += A[ROW * N + i] * B[i * N + COL];
   }
 }
C[ROW * N + COL] = tmpSum;
}

void matrixMultiplication(float *A, float *B, float *C, int N){
// declare the number of blocks per grid and the number of threads
per block
// инициализация значений блока и грида
dim3 threadsPerBlock(N, N);
dim3 blocksPerGrid(1, 1);
if (N*N > 512){
     threadsPerBlock.x = 512;
     threadsPerBlock.y = 512;
     blocksPerGrid.x = ceil(double(N)/double(threadsPerBlock.x));
     blocksPerGrid.y = ceil(double(N)/double(threadsPerBlock.y));
}
matrixMultiplicationKernel<<<blocksPerGrid,threadsPerBlock>>>(A,
B,        C, N);
}
```

33. Структура ядра и адресация на примере перемножения двух матриц в CUDA

Ядро (kernel) – функция, описывающая последовательность операций, выполняемых каждой нитью параллельно.

```
__global__ void mm_kernel(float* A, float* B, float* C, int n)
{
     int col = blockIdx.x * blockDim.x + threadIdx.x;
     int row = blockIdx.y * blockDim.y + threadIdx.y;
     if (row < n && col < n) {
          for (int i = 0; i < n; ++i) {
               C[row * n + col] += A[row * n + i] * B[i * n +
          col];
         }
     }
 }
```

Вызов ядра mm_kernel выполняется следующим образом:

```
int main(){
...
const int N = 1024 * 1024;
dim3 threadsPerBlock(1024);
dim3 numBlocks(N / threadsPerBlock.x);
mm_kernel<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C);
...
}
```

34. Синхронизация потоков, дивергенция потоков, функции голосования в CUDA. Примеры

Для синхронизации потоков блока вне зависимости от принадлежности к warp`ам существует несколько команд барьерного типа:

- **__syncthreads()** — самый верный способ. Эта функция заставит каждый поток ждать:
  - пока все остальные потоки этого блока достигнут этой точки
  - все операции по доступу к разделяемой и глобальной памяти, совершенные потоками этого блока, завершатся и станут видны потокам этого блока.
- **__threadfence_block()** - будет заставлять ждать вызвавший её поток, пока все совершенные операции доступа к разделяемой и глобальной памяти завершатся и станут видны потокам этого блока.
- **__threadfence()** - будет заставлять ждать вызвавший её поток, пока все совершенные операции доступа к разделяемой памяти станут видны потокам этого блока, а операции с глобальной памятью — всем потокам на «устройстве». Под «устройством» понимает графический адаптер.
- **__threadfence_system()** - подобна **__threadfence()**, но включает синхронизацию с потоками на CPU («хосте»), при использовании весьма удобной page-locked памяти.

Дивергенция варпов - ситуация, возникающая когда в нитях одного варпа одна и на же инструкция условного перехода отрабатывает по разному (например, для одной нити отработал блок *if*, а для другой нити отработал блок *else*). Другими словами, в одной группе нитей переход выполняется а в другой нет. Часто возникает в циклах, содержащих специфические условия.

Из-за того, что инструкции всех тредов внутри варпа выполняются синхронно, то исполнение всех тредов, которые выполнили прямой переход будет приостановлено, до того момента пока остальные треды не достигнут той же инструкции.

Дивергенция приводит к тому, что время исполнения одного варпа определяется суммарным временем исполнения всех выполненных ветвей.

Функции голосования (начиная с CUDA 2.x) - набор встроенных функций, которые предоставляют возможность “общения нитей” в пределах варпа.

- **__all(bool value)** - возвращает 1, если все нити в варпе имеют значение 1, иначе 0.
- **__any(bool value)** - Возвращает 1, если хотя бы одна нить в варпе имеет значение 1, иначе 0.
- **__ballot(bool(неточно) value)** - Возвращает 32-битное значение, где каждый бит соответствует одной нити в варпе и содержит ее значение.

Пример синхронизации есть в вопросе про редукцию (40).

Пример дивергенции варпов (ГПТ)

```
__global__ void kernel(int *data, int n) {
     int tid = threadIdx.x + blockIdx.x * blockDim.x;

     if (tid < n) {
          if (data[tid] > 0) {
   // Операции, выполняемые только для положительных элементов
                    data[tid] *= 2;
          } else {
  // Операции, выполняемые только для отрицательных элементов
                    data[tid] /= 2;
          }}}
```

Пример использования функций голосования (ГПТ)

Задача: найти минимальный элемент числового массива.

```
#include <cuda_runtime.h>

__global__ void findMin(int *data, int *minVal, int n) {
     int tid = threadIdx.x + blockIdx.x * blockDim.x;

     // Проверяем границы массива
     if (tid < n) {
          int myVal = data[tid];

          // Инициализируем локальное минимальное значение
          int localMin = myVal;

          // Итеративно находим минимальное значение в варпе
          for (int s = warpSize / 2; s > 0; s >>= 1) {
              int neighbor = myVal + s;
              if (neighbor < n) {
                   localMin = __min(localMin, data[neighbor]);
              }
          }

          // Все нити в варпе записывают свое локальное минимальное
     значение
          data[tid] = localMin;

          //  Используем функцию голосования для определения
     глобального минимума
          *minVal = __all(data[tid] == localMin) ? localMin :
     *minVal;
     }
}
```

35. Архитектура современного GPU

Графический процессор (GPU) — это специализированное устройство для параллельной обработки больших объемов данных.

Основное назначение современных GPU — выполнение задач, связанных с графикой, вычислениями в реальном времени, машинным обучением, научными вычислениями и другими интенсивными вычислительными нагрузками.

Архитектура GPU разработана с учетом максимальной эффективности выполнения параллельных вычислений.

Основные компоненты архитектуры GPU:

1. Множество ядер (SIMD-архитектура): GPU состоит из сотен или тысяч вычислительных ядер. Каждое ядро способно выполнять одну и ту же инструкцию на разных данных (Single Instruction, Multiple Data, или SIMD).

2. Многоуровневая организация вычислительных блоков:

    - Streaming Multiprocessors (SMs): GPU состоит из нескольких блоков SM, каждый из которых содержит множество ядер, специализированные блоки для операций с плавающей запятой (FP32/FP64) и целыми числами, тензорные ядра для ускорения матричных операций, используемых в машинном обучении, RT-ядра для аппаратного ускорения трассировки лучей (в графических задачах).

    - Warp-архитектура: Потоки объединяются в группы (обычно 32 потока), которые выполняются синхронно. Это позволяет эффективно распределять вычисления.

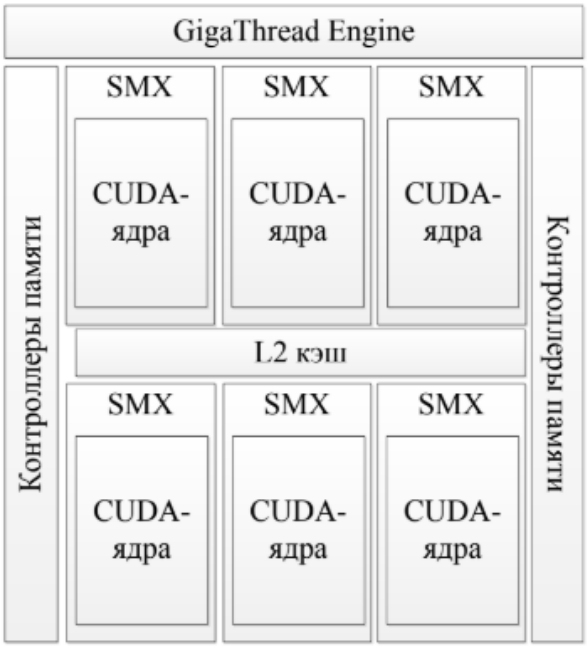
3. Память: GPU имеет сложную иерархию памяти, оптимизированную для высокой пропускной способности: регистры, Shared memory, Global memory, Texture и Constant memory, Кэш-память.

Современные GPU поддерживают гибкую программируемость благодаря платформам:

- CUDA (от NVIDIA) — для программирования общего назначения.
- OpenCL — стандартный фреймворк для работы с GPU различных производителей.
- DirectX, Vulkan и OpenGL — графические API для работы с 3D-графикой.
- ROCm — платформа для GPU от AMD.

Особенности современных GPU:

1. Гетерогенные вычисления. GPU часто используются в связке с CPU (гетерогенные системы). CPU управляет задачами, требующими сложной логики, а GPU выполняет параллельные вычисления.
2. Акселерация AI:
3. Ускорение графики. RT-ядра и специализированные технологии позволяют воспроизводить сложные графические сцены с реалистичными эффектами, такими как трассировка лучей.
4. Поддержка виртуализации. Современные GPU поддерживают виртуализацию, что делает их полезными для облачных вычислений.



Упрощенная структура гпу

36. Понятие occupancy в CUDA. Пример расчета

В CUDA термин «occupancy» (заполняемость) описывает степень использования вычислительных ресурсов графического процессора (GPU). Он определяется как отношение числа активных варпов (warps) на одном потоковом мультипроцессоре (SM) к максимальному числу варпов, которые могут быть одновременно активны на этом SM.

Для расчета occupancy (загруженности) GPU необходимы следующие параметры:

Параметры устройства (GPU)
Максимальное количество активных потоков на мультипроцессор (SM)
Максимальное количество активных блоков на мультипроцессор (SM)
Максимальное количество активных warp на мультипроцессор
Общее количество регистров на мультипроцессор
Размер единицы выделения регистров
Объём разделяемой памяти (Shared Memory) на мультипроцессор
Гранулярность выделения warp
Гранулярность выделения регистров
Гранулярность выделения разделяемой памяти

Параметры ядра (Kernel)

1. Количество потоков в блоке.
2. Число регистров, используемых каждым потоком.
3. Объём разделяемой памяти (Shared Memory), используемый блоком.

Пример : На устройстве с вычислительной мощностью 7.0 рассмотрим ядро с блоками из 128 потоков, где каждый поток использует 37 регистров.

Из раздела **"Физические ограничения для вычислительной мощности GPU"** известно, что максимальное количество активных warps равно 64 для вычислительной мощности 7.0.

Количество регистров, необходимых для одного warp, определяется по формуле:

*Необходимые регистры для одного warp = (число потоков в warp×число регистров на поток / 256) * 256*

$$\left\lceil \frac{37 \times 32}{256} \right\rceil \times 256 = 1280$$

Максимальное количество активных warp на мультипроцессоре (SM) при заданной гранулярности выделения warp рассчитывается как:

Максимум активных warp = (Общее количество регистров / Необходимые регистры для одного warp / 4) * 4

$$\left\lfloor \frac{65536/1280}{4} \right\rfloor \times 4 = 48$$

Occupancy=Максимум активных warp/общее количество warp.

48/64 = 75%

37. Типы памяти в CUDA. Примеры создания и организации доступа

Глобальная память:
Предназначена для хранения данных, доступных всем нитям на устройстве. Высокая латентность доступа. Кешируется в уровнях L1 и L2.
Динамически с хоста:

```
int *deviceArray;
cudaMalloc(&deviceArray, size);
cudaMemcpy(deviceArray, hostArray, size, cudaMemcpyHostToDevice);
```

Статически:

```
__device__ int deviceArray[100];
```

Разделяемая (shared) память
Объем ограничен. Используется для обмена данными между нитями одного блока. Каждому блоку нитей выделяется свой участок памяти.
Пример:

```
__global__ void kernel() {
__shared__ int sharedArray[256];
sharedArray[threadIdx.x] = threadIdx.x;
__syncthreads();
}
```

Константная память
кешируется в read-only кеш Uniform Cache. Хранение неизменяемых данных, общих для всех нитей.
Пример:

```
__constant__ float constData[256];
```

Копирование с хоста:

```
cudaMemcpyToSymbol(constData,hostData, sizeof(hostData));
```

Регистровая память
Объем ограничен, делится между всеми активными нитями. Хранение временных переменных для каждой нити.
Пример:

```
__global__ void kernel() {
int localVar = threadIdx.x; – размещается в регистрах}
```

Локальная память
Хранение данных, специфичных для каждой нити, когда регистров недостаточно.
Пример:

```
__global__ void kernel() {
int largeArray[1024]; – размещено в локальной памяти
}
```

38. Механизм транзакций в CUDA. Пример

Транзакция – выполнение загрузки из глобальной памяти сплошного отрезка данных размерностью 128 байт с адреса кратного 128. Размер кэш-линии – 128 байт, на которые проецируется соответственно 128-байтный выравненный сегмент в глобальной памяти.

В случае если размер слова, запрашиваемого нитью, больше 4 байт, запрашиваемый варпом, блок данных разделяется на 128-байтные запросы, обрабатываемые независимо. Транзакции по 128 байт будут формироваться до тех пор, пока все нити варпа не получат необходимые данные.

```
cudaMalloc(%d_src, 1024*sizeof(int));
…

__global__ void kernel(int *src, int size) {
          offset = … ;
          int a = src[offset];
          // действия
}
```

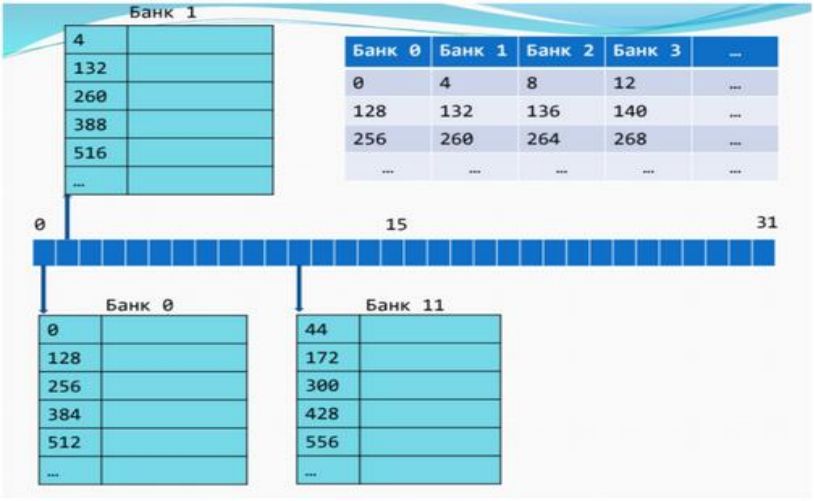
```
Если offset = blockIdx.x * blockDim.x + threadIDx.x, транзакции +
Если offset = blockIdx.x * blockDim.x + threadIDx.x * 3,
транзакции x3
```

39. Конфликт по банкам в разделяемой памяти в CUDA. Пример

Для увеличения полосы пропускания блок, на котором расположена общая память, разделен на подмодули (банки).

Пусть n - число банков, m - количество последовательных байтов, которое один банк может отдать за 1 цикл.

Адресное пространство общей памяти разделено на n непересекающихся подмножеств, расположенных в разных банках. Банки работают независимо друг от друга и могут вместе выдать максимум n*m байтов за один цикл.



Если хотя бы два нужных варпу слова расположены в одном банке, такая ситуация называется «банк конфликтом». Такое обращение аппаратно разбивается на серию обращений, не содержащих банк конфликтов. Если число обращений, на которое разбит исходный запрос, равно n, то такая ситуация называется банк-конфликтом порядка n. Пропускная способность при этом падает в n раз. Наиболее оптимальный способ обращения к разделяемой памяти - когда каждая нить в пределах варпа оперирует данными в одном банке.

Латентность доступа к глобальной памяти крайне высока, поэтому в правильно спроектированном алгоритме чтение и запись элемента данных должно осуществляться только один раз, при этом должны формироваться транзакции. Поэтому, как правило, ядро начинается с загрузки данных из глобальной в разделяемую память и заканчивается сохранением результата работы из разделяемой в глобальную память. Латентность доступа к разделяемой памяти в сравнении с глобальной гораздо ниже, однако она не нулевая. Поэтому при использовании одной нитью значений из разделяемой памяти более одного раза может оказаться эффективной программная реализация дополнительного кэша, построенного на уровне регистров (локальных переменных). В данные переменные размещаются наиболее часто используемые данные и в дальнейшем могут использоваться для аккумулирования промежуточного результата.

Пример конфликта банков (ГПТ)

```
__shared__ float sharedData[1024];
__global__ void kernel(float *data, int n) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    // Каждый поток записывает в свою ячейку, но с шагом 1
    sharedData[tid] = data[tid];

    __syncthreads();

    // Каждый поток считывает соседнее значение
    float sum = sharedData[tid] + sharedData[tid + 1];
    // ...
}
```

Более хороший код:

```
__shared__ float sharedData[1024];

__global__ void kernel(float *data, int n) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    // Каждый поток записывает в свою ячейку, но с шагом, кратным числу банков
    int bankSize = 32; // Предположим, что в банке 32 элемента
    sharedData[tid * bankSize] = data[tid];
    __syncthreads();

    // Каждый поток считывает значение из соседнего банка
    float sum = sharedData[tid * bankSize] + sharedData[(tid + 1) * bankSize];
    // ...
}
```

Во втором примере каждый поток обращается к разным банкам, что минимизирует конфликты.

40. Алгоритм редукции в CUDA. Пример

Редукция – сумма значений всех элементов массива.

```
__global__ void reduceBase(int* inputData, int* outputData) {
extern __shared__ int smemData[];
// загружаем данные в разделяемую память
const unsigned tid = threadIdx.x;
smemData[tid] =

inputData[blockIdx.x * blockDim.x+ threadIdx.x];

// синхронизируемся
__syncthreads();
// выполняем редукцию над элементами массива
for( unsigned s = 1; s < blockDim.x; s <<= 1 ) {
if( tid % ( s << 1 ) == 0 ) {
smemData[tid] += smemData[tid + s];
}
// синхронизируемся, чтобы гарантировать
// корректность входных данных
// на следующей итерации цикла
__syncthreads();
}
// сохраняем результат вычислений
if ( tid == 0 )
outputData[blockIdx.x] = smemData[0];
}
```

41. Алгоритм свертки в CUDA. Пример

```
__global__ void convolution2D(float *input, float *output, float
*kernel, int width, int height, int kernelWidth, int kernelHeight)
{
     // Индексы текущего элемента
     int x = blockIdx.x * blockDim.x + threadIdx.x;
     int y = blockIdx.y * blockDim.y + threadIdx.y;

     // Индексы в shared memory
     int tx = threadIdx.x;
     int ty = threadIdx.y;

     // Размеры shared memory
     extern __shared__ float sharedData[];

     int sharedWidth = blockDim.x + kernelWidth - 1;
     int sharedHeight = blockDim.y + kernelHeight - 1;

     // Загрузка данных в shared memory
     for (int i = ty; i < sharedHeight; i += blockDim.y) {
          for (int j = tx; j < sharedWidth; j += blockDim.x) {
               int globalX = blockIdx.x * blockDim.x + j -
kernelWidth / 2;
               int globalY = blockIdx.y * blockDim.y + i -
kernelHeight / 2;

               if (globalX >= 0 && globalX < width && globalY >= 0
&& globalY < height) {
                    sharedData[i * sharedWidth + j] =
input[globalY * width + globalX];
               } else {
                    sharedData[i * sharedWidth + j] = 0.0f; //
Паддинг
               }
          }
     }
     __syncthreads();

     // Вычисление свертки
     if (x < width && y < height) {
          float sum = 0.0f;
          for (int i = 0; i < kernelHeight; ++i) {
                    for (int j = 0; j < kernelWidth; ++j) {
                         sum += sharedData[(ty + i) * sharedWidth
     + (tx + j)] *
                                kernel[i * kernelWidth + j];
                    }
          }
          output[y * width + x] = sum;
     }
}
```

42. Алгоритм операции инклюзивного scan в CUDA. Пример

Базовый алгоритм последовательного scan (на CPU для понимания), f - функция (например, сложение, min, max, произведение)

```
output[0] = input[0];
for(i = 1; i < N; ++i) {
     output[i] = f(output[i-1], input[i]);
}
```

```
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
__shared__ float buffer_s[BLOCK_DIM];
buffer_s[threadIdx.x] = input[i];
__syncthreads();
for(unsigned int stride = 1; stride <= BLOCK_DIM/2; stride *=
2) {
    float v;
    if(threadIdx.x >= stride) {
        v = buffer_s[threadIdx.x - stride];
    }
    __syncthreads();
    if(threadIdx.x >= stride) {
        buffer_s[threadIdx.x] += v;
    }
    __syncthreads();
}
if(threadIdx.x == BLOCK_DIM - 1) {
    partialSums[blockIdx.x] = buffer_s[threadIdx.x];
}
output[i] = buffer_s[threadIdx.x];
```

43. Алгоритм операции эксклюзивного scan в CUDA. Пример

Базовый алгоритм последовательного scan (на CPU для понимания),
f - функция (например, сложение, min, max, произведение),
IDENTITY - обычно просто 0

```
output[0] = IDENTITY;
for(i = 1; i < N; ++i) {
     output[i] = f(output[i-1], input[i-1]);
}
Exclusive Scan Code

unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
__shared__ float buffer1_s[BLOCK_DIM];
__shared__ float buffer2_s[BLOCK_DIM];
float* inBuffer_s = buffer1_s;
float* outBuffer_s = buffer2_s;
if(threadIdx.x == 0) {
    inBuffer_s[threadIdx.x] = 0.0f;
}
else {
    inBuffer_s[threadIdx.x] = input[i - 1];
}
__syncthreads();

for(unsigned int stride = 1; stride <= BLOCK_DIM/2; stride *=
2) {
        if(threadIdx.x >= stride) {
            outBuffer_s[threadIdx.x] = inBuffer_s[threadIdx.x] +
inBuffer_s[threadIdx.x - stride];
        }
        else {
            outBuffer_s[threadIdx.x] = inBuffer_s[threadIdx.x];
        }
        __syncthreads();
        float* tmp = inBuffer_s;
        inBuffer_s = outBuffer_s;
        outBuffer_s = tmp;
    }
    if(threadIdx.x == BLOCK_DIM - 1) {
        partialSums[blockIdx.x] = inBuffer_s[threadIdx.x] +
input[i];
    }
    output[i] = inBuffer_s[threadIdx.x];
```

44. Асинхронное и синхронное копирование в CUDA. Pinned память. Способы выделения

Основная разница во времени выполнения асинхронных и синхронных версий команд копирования проявляется при работе с pinned памятью в качестве входного параметра. (также влияет неявный вызов синхронизации в синхронной версии, поэтому и основная).

Pinned-память - page-locked память, т.е. страницы виртуальной памяти становятся жёстко привязанны к физической. С такой памятью возможны асинхронные операции с перемещением памяти.

Копирование такой памяти происходит быстрее, т.к. при копировании страничной адресации памяти под капотом происходит перенос её содержимого в pinned буфер. С такой памятью работает dma, из-за чего и достигается мгновенное возвращение из функций асинхронного копирования. Её возможно не только выделять, но и делать таковой уже выделенную страничной адресации память.

Для выделения памяти в pinned-режиме достаточно выделить память в ОЗУ с применением следующей функции:

```
cudaError_t cudaMallocHost (
void** ptr, // указатель на выделяемую память в ОЗУ
size_t size // объем выделяемой памяти
}
```

Заранее выделенная системная память может быть также зарегистрирована в pinned-режиме с использованием CUDA API:

```
cudaError_t cudaHostRegister (
void * ptr, // указатель на память ОЗУ
size_t size, // размер регистрируемой памяти в байтах
unsigned int flags
)
```

45. CUDA Stream. Создание, инициализация и синхронизация

Технология *CUDA stream* позволяет при соблюдении ряда условий организовать параллельное выполнение некоторых задач. Особенно удобно при работе с более чем одним gpu.

Создание потока:

```
cudaStream_t stream;
cudaStreamCreate(&stream);
...
cudaStreamDestroy(stream);
```

Поток привязывается к текущему активному устройству: перед отправлением команды нужно переключаться на устройство, к которому привязан поток, если попробовать отправить в него команду при другом активном устройстве, будет ошибка.

Инициализация stream'а выглядит следующим образом:

```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
// асинхронное копирование на устройство
cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice,
stream1);
// запуск ядра
kernelStream<<<1, N, 0, stream1>>>(d_a);
... // дальнейшая работа со stream1, CPU
cudaStreamDestroy(stream1);
```

Для *синхронизации*, например, со stream1 введена функция `cudaStreamSynchronize(stream1)`, которая блокирует хост до тех пор, пока не будет полностью выполнены все команды для stream1. Для того чтобы узнать завершены команды stream'а или нет без блокировки хоста используется функция `cudaStreamQuery(stream)`.

Синхронизация на GPU:

```
cudaError_t cudaStreamWaitEvent;
(cudaStream_t stream, cudaEvent_t event, unsigned int
flags ).
```

Команды, отправленные в stream, начнут выполняться после наступления события event

— Синхронизация будет эффективно выполнена на GPU;

— При stream == NULL будут отложены все команды всех потоков.

Событие event может быть записано на другом GPU: - синхронизация между GPU.

Синхронизация по потоку:

```
cudaError_t cudaStreamQuery (cudaStream_t stream);
```

cudaSuccess -> если выполнены все команды в stream, иначе cudaErrorNotReady.

```
cudaError_t cudaStreamSynchronize (cudaStream_t stream);
```

— Возвращает управление хостовой нити, когда завершится выполнение всех команд, отправленных в поток stream.