OpenCL и OpenGL compute shaders

Лектор:

Боресков А.В. (ВМК МГУ), steps3d@narod.ru

Основы OpenCL

OpenCL – открытый кроссплатформенный стандарт для параллельных вычислений на гетерогенных устройствах

Изначально разработан Apple, сейчас поддерживается Khronos Group,

Первая версия (1.0) – конец 2008 года

Поддерживает GPU, CPU, Cell, DSP и многие другие устройства

Основы OpenCL

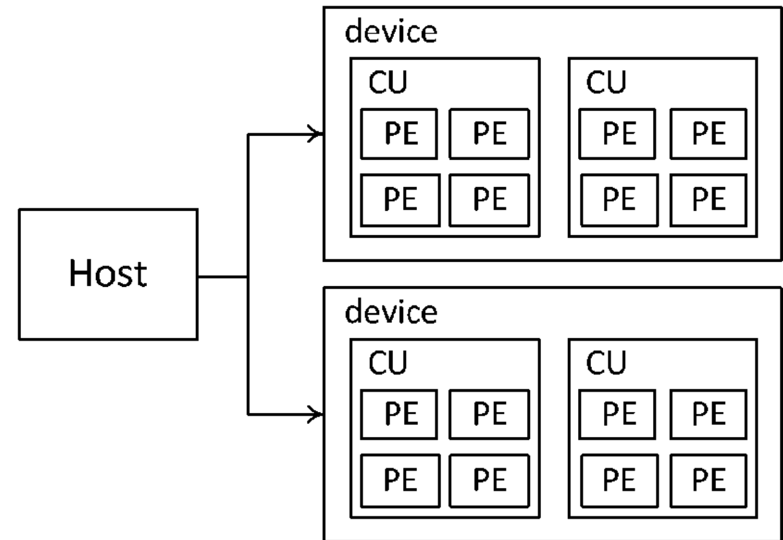
Основан на следующих обобщенных моделях

- Platform model
- Memory model
- Execution model
- Programming model

OpenCL platform model

Платформа состоит из хоста (CPU) и одного или нескольких вычислительных устройств (device)

Каждое устройство состоит из вычислительных блоков (Compute Unit), которые состоят из Processing Elements (PE)



OpenCL platform model

Получение списка доступных платформ

```
cl_int clGetPlatformIDs (
        cl_uint numEntries,
        cl_platform_id * platforms,
        cl_uint * numPlatforms );
```

Получение списка доступных платформ

```
cl_platform_id   platform;
cl_device_id     device;
cl_ uint         err ;
err = clGetPlatformIDs ( 1 , &platform , NULL ) ;
if ( err != CL_SUCCESS )
{
  p r i n t f ( ”Error obtaining OpenCL platform. \n” ) ;
  return -1;
}
e r r = clGetDeviceIDs ( platform , CL_DEVICE_TYPE_GPU, 1 , &device ,
  NULL ) ;
if ( err != CL_SUCCESS )
{
  p r i n t f ( ”Error obtaining OpenCL device. \n” ) ;
  return -1;
}
```

Получение информации о платформе

```
cl_int clGetPlatformInfo ( cl_platform_id platform ,
         cl_platform_info pname ,
         size_t valueBufSize,
         void * valueBuf,
         size_t * valueSize );
```

Получение устройств и информации об устройстве

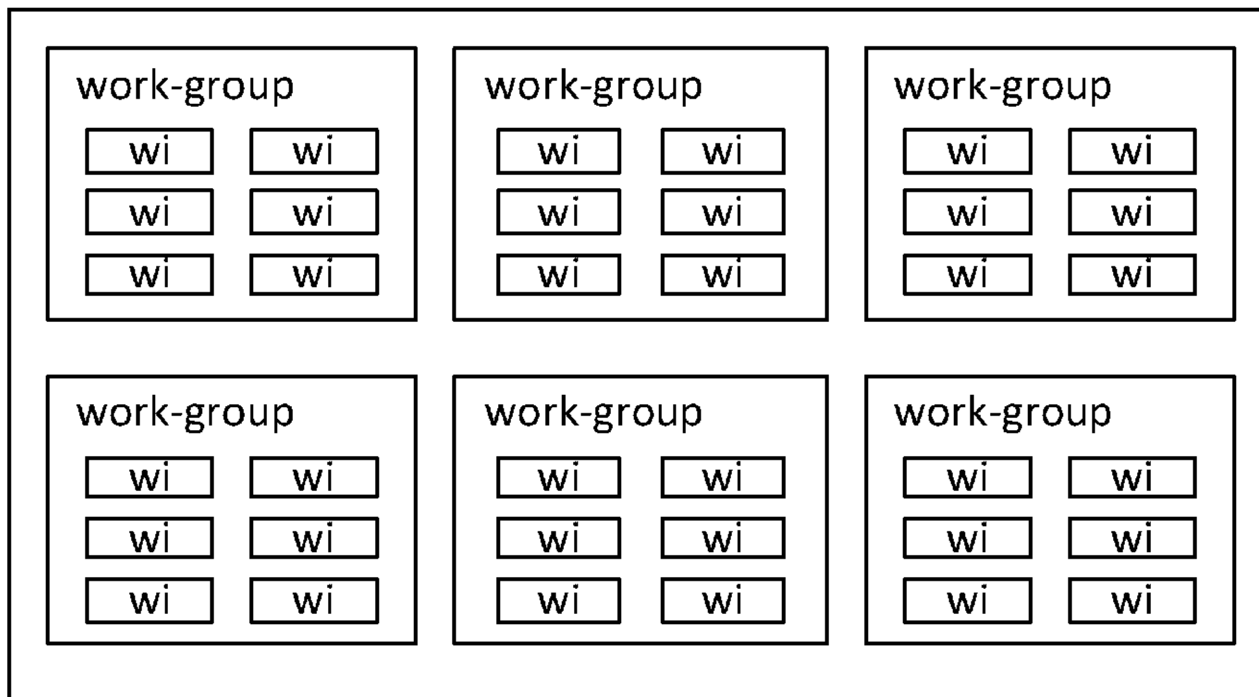
```
cl_ int clGetDeviceIDs (
     cl_platform_id platform,
     cl_device_type deviceType,
     cl_uint numEntries,
     cl_device_id * devices,
     cl_uint * numDevices );

cl_int clGetDeviceInfo (
     cl_device_id device,
     cl_deviceInfo pname,
     size_t valueBufSize,
     void * valueBuf,
     size_t * valueSize );
```

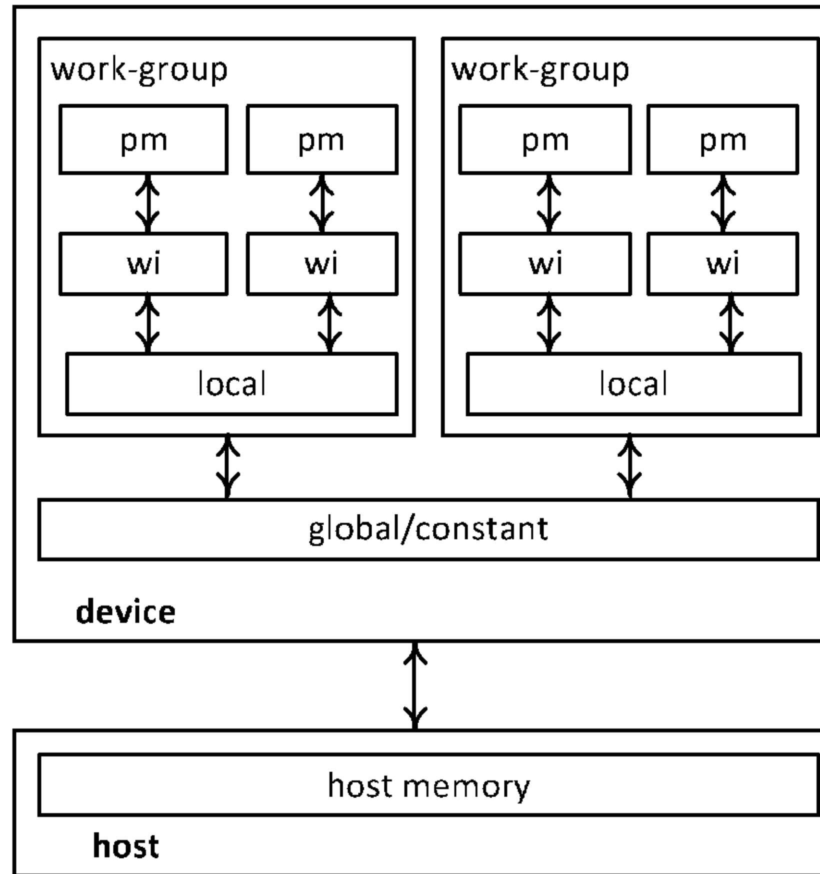
Вычислительная модель OpenCL

- Stream processing models
- Ядра (написанные на специальном основанном на C языке) запускаются на устройстве
- Ядро запускается для каждого элемента N-мерной вычислительной области (N=1,2,3) (ND-range) как work-item
- Work-item’ы группируются в work-group’ы

Вычислительная модель OpenCL



OpenCL memory model



Типы памяти – global, constant, local, private

Типы памяти в OpenCL

__global	- как в CUDA
__constant	- как в CUDA
__local	- как shared в CUDA
__private	- как локальная в CUDA

Контекст OpenCL

Контекст OpenCL содержит в себе

- Устройства
- Ядра
- Объекты программ
- Memory objects
- Command queue

Контекст OpenCL

```
cl_context clCreateContext (
    const cl_context_properties * props,
    cl_uint numDevices,
    const cl_device_id * devices,
    void (CL CALLBACK * notify) ( const char * errInfo, const void *
        privateInfo,  size_t cb, void * userData ),
    void * userData,
    cl_int * errCode );
```

Command queue

Каждый девайс должен иметь свою очередь.

В очередь помещаются запросы на

- Выполнение ядра
- Операции с памятью
- Команды синхронизации

Помещаемые команды выполняются асинхронно. Могут выполняться in-order и out-of-order

Command queue

```
cl_command_queue clCreateCommandQueue (
  cl_context_context,
  cl_device_id device,
  cl_command_queue_properties props,
  cl_ int * errCode );
```

Props

- CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE
- CL_QUEUE_PROFILING_ENABLED

Ядра

Пишутся на основанном на C99 языке из которого убраны

- Указатели на функции
- Битовые поля
- Массивы переменной длины
- Рекурсия
- Стандартные заголовки

Ядро помечается спецификатором __kernel

Ядра

Добавлены новые типы (n=2,3,4,8,16)

```
charN, ucharN (на хосте cl_charN, cl_ucharN)
shortN, ushortN (cl_shortN, cl_ushortN)
intN, uintN (cl_intN, cl_uintN)
longN, ulongN (cl_longN, cl_ulongN)
floatN (cl_floatN)
```

Пример работы с типами

```
float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
float2 f1 = f.zx;
uint4  u = (uint4) (0);
float2 lo = f.lo;      // f.xy
float2 ev = f.even;    // f.xz
float  ff = f.s3;      // f.z
```

Получение информации в ядре

Функция	Что возвращает
get_num_groups(idx)	Размер ND-range в work-group’ах
get_local_size(idx)	Размер work-group в work-item’ах
get_group_id(idx)	Глобальный индекс work-group’ы
get_local_id(idx)	Локальный индекс work-item’а в текущей work-group’е
get_global_id(idx)	Глобальный индекс work-item’а в ND-range
get_global_size(idx)	Размер ND-range в work-item’ах

Функции синхронизации

```
void barrier          ( cl_mem_fence_flags flags );
void mem_fence        ( cl_mem_fence_flags flags );
void read_mem_fence   ( cl_mem_fence_flags flags );
void write_mem_fence  ( cl_mem_fence_flags flags );
```

Пример ядра

```
__kernel void test ( __global float * a, int n )
{
  int idx = get_global_id ( 0 );
  if ( idx < n )
    a [idx] = sin ( idx * 3.1415926f / 1024.0f );
}
```

Создание и компиляция программы

```
cl_program clCreateProgramWithSource (
    cl_context context,
    cl_uint count,
    const char ** strings,
    const size_t * lengths,
    cl_int * errCode );

cl_int clBuildProgram (
    cl_program program,
    cl_uint numDevices,
    const cl_device_id * devices,
    const char * options,
    void (*notify)(cl_program, void *),
    void * userData );
```

Создание ядра

```
cl_kernel clCreateKernel (
    cl_program program,
    const char * kernelName,
    cl_int * errCode );
```

Буфера

```
cl_mem clCreateBuffer (
    cl_context context,   cl_mem_flags flags,
    size_t size , void * hostPtr,
    cl_int * errCode );
cl_int clEnqueueWriteBuffer (
    cl_command_queue queue, cl_mem buffer,
    cl_bool blockingWrite,
    size_t offset, size_t numBytes,
    const void * hostPtr,
    cl_uint numEvents, const cl_event * waitList,
    cl_event * event );
```

Запуска ядра

```
cl_int clSetKernelArg ( cl_kernel kernel,
    cl_int argIndex, size_t argSize,
    const void * argPtr );

cl_int clEnqueueNDRangeKernel (
    cl_command_queue queue,
    cl_kernel kernel,
    cl_uint workDim,
    const size_t * globalWorkOffset,
    const size_t * globalSize,
    const size_t * localSize,
    cl_uint numEvents, const cl_event * waitList,
    cl_event * event );
```

C++ API

```
#define __NO_STD_VECTOR       // cl::vector вместо std::vector
#define __CL_ENABLE_EXCEPTIONS // вместо ошибок exception
#include <CL/cl.hpp>
using namespace cl;
vector<Platform> platforms;
Platform::get(&platforms);
cl_context_properties cps[3] = { CL_CONTEXT_PLATFORM,
  (cl_context_properties)(platforms[0])(), 0 };
Context context( CL_DEVICE_TYPE_GPU, cps);
vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();
CommandQueue queue = CommandQueue(context, devices[0]);
Program::Sources source(1, std::make_pair(sourceCode.c_str(),
  sourceCode.length()+1));
Program program = Program(context, source);
program.build(devices);
Kernel kernel(program, "vector_add");
Buffer bufferA = Buffer(context, CL_MEM_READ_ONLY, LIST_SIZE *
  sizeof(int));
```

C++ API

```
Buffer bufferC = Buffer(context, CL_MEM_WRITE_ONLY,
                        LIST_SIZE * sizeof(int));
queue.enqueueWriteBuffer(bufferA, CL_TRUE, 0,
                        LIST_SIZE * sizeof(int), A);
kernel.setArg(0, bufferA);
kernel.setArg(1, bufferC);

NDRange global(LIST_SIZE);
NDRange local(1);

queue.enqueueNDRangeKernel(kernel, NullRange, global, local);
queue.enqueueReadBuffer(bufferC, CL_TRUE, 0,
                        LIST_SIZE * sizeof(int), C);
```

OpenGL compute shaders

- Добавлен новый тип шейдеров GL_COMPUTE_SHADER
- Добавлен новый тип буферов в GLSL и самом OpenGL – Shader Storage Buffer Object
  - Может быть очень большим
  - Шейдеры могут читать и писать по произвольным полям/индексам
  - Можно применять атомики

OpenGL compute shaders

- Расширение ARB_compute_shaders
  - Входит в OpenGL 4.3 core
  - Новый тип буфера – GL_DISPATCH_INDIRECT_BUFFER
  - Команды для запуска грида на счет
    - glDispatchCompute
    - glDispatchComputeIndirect

OpenGL compute shaders

```
void glDispatchCompute (
            GLuint numGroupsX,
            GLuint numGroupsY,
            GLuint numGroupsZ );

void glDispatchComputeIndirect (
            GLintptr offset );
```

OpenGL compute shaders

Задание размера рабочей группы -

```
layout ( local_size_x = 32,
         local_size_y = 16 ),
         local_size_z = 2 ) in;
```

OpenGL compute shaders

Встроенные переменные

```
uvec3 gl_NumWorkGroups;
uvec3 gl_WorkGroupSize;
uvec3 gl_WorkGroupID;
uvec3 gl_LocalInvocationID;
uvec3 gl_GlobalInvocationID;
uint gl_LocalInvocationIndex;
```

OpenGL compute shaders

Разделяемая память

```
shared float buf
  [gl_WorkGroupSize.x+2][gl_WorkGroupSize.y+2];
```

```
void memoryBarrier ();
void memoryBarrierBuffer ();
void memoryBarrierShared ();
void groupMemoryBarrier ();
```

OpenGL SSBO

ARB_shader_storage_buffer_object

OpenGL 4.3 core GL_SHADER_STORAGE_BUFFER

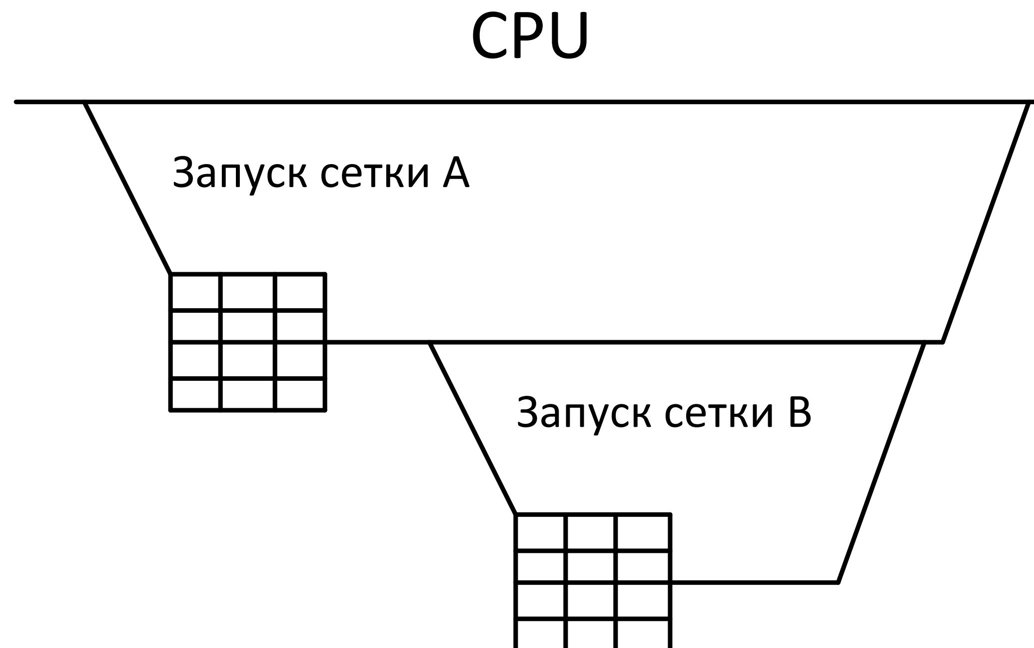
```
#version 430 layout( local_size_x = 1000 ) in;
layout(std430, binding = 0) buffer Pos
{
    vec4 position [];
};
layout(std430, binding = 1) buffer Vel {
    vec4 velocity [];
};
```

Динамический параллелизм

- Рассмотренная ранее модель вычислений предполагала запуск сеток со стороны CPU
  - Во многих случаях этого достаточно
  - Но не всегда, например при использовании регулярных сеток
- Динамический параллелизм – это возможность запускать ядра непосредственно на стороне GPU
  - Каждая нить может запустить сетку и не одну
  - Конструкция для запуска сетки точно такая же как для CPU
  - Запускающая сетка называется *родительской*
  - Запускаемая сетка называется *дочерней*
  - А дочерняя сетка также может запускать новые сетки и т.д.

Динамический параллелизм

- Запускаемые сетки являются вложенными
  - Гарантированно завершаются перед завершением родительской сетки



Динамический параллелизм

- Родительская сетка может явно дождаться завершения дочерний через вызов cudaDeviceSynchronize
  - Если сразу несколько нитей запускают дочерние сетки, то убедитесь, что они все запущены через __syncthreads
  - Это дорогостоящий вызов, выполните из одной нити
- Гарантируется, что при запуске дочерней сетки и при ее завершении они одинаково видят глобальную память
  - Записанное одной сеткой будет видно другой
- Но это не верно во время работы дочерней сетки

Динамический параллелизм

- Нельзя передавать в дочернюю сетку указатели на
  - Разделяемую память
  - Локальные переменные
- Дочерние сетки запускаются асинхронно, т.е. ставятся в очередь на выполнение
- Дочерние сетки выполняются последовательно
  - Иначе используйте потоки
  - Потоки необходимо создавать и уничтожать на GPU
  - cudaStreamCreateWithFlags (&stream, cudeStreamNonBlocking);
  - Потоки, созданные в разных блоках, считаются разными

Динамический параллелизм

- Есть ограничения по глубине рекурсии и глубине синхронизации
- Для сборки необходимо указывать дополнительные флаги
  - nvcc test.cu --machine 64 --debug --gpu-architecture=sm_35 -rdc=true -lcudadevrt -o test

```
__global__ void kernel ( int level, float a )
{
   printf ( "Thread %d level %d value %f\n", threadIdx.x, level, a );
   if ( level == 0 )
      kernel <<<1, 5>>> ( level + 1, a*a );
}
int main () {
   kernel<<<1, 5>>> ( 0, 3 );
   cudaDeviceSynchronize ();
   return 0;
}
```

Поддержка printf

- На стороне GPU поддерживается printf
  - Выводит в кольцевой буфер данные
  - Дополнительное форматирование на стороне CPU
  - Размер буфера по умолчанию 1Мб, задается как
    - cudaDeviceSetLimit (cudaLimitPrintfFifoSize, newSize);
  - Содержимое буфера печатается при
    - Запуске ядра
    - Явной синхронизации (cudaDevice/Stream/EventSynchronize)
    - Любом блокирующем копировании памяти через cudaMemcpy*
    - cudaDeviceReset
    - Перед вызовом callback-функции для потока
  - Поддерживается до 32 аргументов и почти все форматы

Поддержка выделения памяти на стороне GPU

- Поддерживаются функции malloc, free, memcpy и memset для выделения, освобождения и копирования памяти на GPU
- Память выделяется из кучи, по умолчанию ее размер 1 Мб
  - cudaDeviceSetLimit (cudaLimitMallocHeapSize, size);
  - Нельзя динамически менять размер кучи
- Выделенную malloc можно освободить только через free
- Через free можно освободить только выделенную через malloc
- Полученные адреса нельзя передавать в вызове API на стороне CPU
- Выделенная память выровнена как минимум по 16 байтам

Описатель __launch_bounds__

- Можно отдельно для каждого ядра задать компилятору пожелания по запуску нескольких блоков на мультипроцессоре
  - maxThreadsPerBlock – наибольшее число нитей в блоке
  - minBlocksPerMultiProcessor – необязателен, желаемое число блоков на мультипроцессор
  - Полезно использовать __CUDA_ARCH__ для подбора этих параметров

```
__global__void _launch_bounds__ ( maxThreadsPerBlock,
      minBlockPerMultiProcessor ) kernel ()
{
   . . .
}
```

Дополнительные функции синхронизации блоков

```
int __syncthreads_count ( int predicate );
int __syncthreads_and   ( int predicate );
int __syncthreads_or    ( int predicate );
void __syncwarp         ( unsigned int mask = 0xFFFFFFFF );
```

- Значение параметра predicate вычисляется перед вызовом и определяет возвращаемое значение
- Выполняет синхронизацию нитей в блоке
- `__syncthreads_count` - число нитей, для которых predicate != 0
- `__syncthreads_and` - для всех нитей predicate != 0
- `__syncthreads_count` - хотя бы для одной нити predicate != 0
- `__syncwarp` - Синхронизация для варпа, объединение нитей после ветвления

Функции для работы на уровне отдельных варпов

```
int __all_sync    ( unsigned int mask, int predicate );
int __any_sync    ( unsigned int mask, int predicate );
int __ballot_sync ( unsigned int mask, int predicate );
int __activemask  ();

unsigned int __match_any_sync ( unsigned mask,
                 T value );
unsigned int __match_all_sync ( unsigned mask, T value,
                 int * pred );
```

Функции для работы на уровне отдельных варпов

- Функция **__all_sync** возвращает ненулевое значение, если для всех нитей варпа из *mask* и не завершивших свое выполнение значение параметра *predicate* не равно нулю
- Функция **__any_sync** возвращает ненулевое значение, если хотя бы для одной из нитей, указанных в параметре *mask* и не завершивших свое выполнение значение *predicate* не равно нулю
- Функция **__ballot_sync** возвращает целое число? *N*-ый бит которого равен 1, если нить *N* еще не завершила выполнение (активна), указана в *mask* и для нее *predicate* не равен нулю
- Функция **__activemask** возвращает битовую маску, задающую какая из нитей варпа до сих пор активна (т.е. не завершила свое выполнение)

Функции для работы на уровне отдельных варпов

- Функция **__match_any_sync** (доступна для CC 7.0 и выше) возвращает маску нитей варпа, у которых одно и тоже значение *value*
- Функция **__match_all_sync** (доступна для CC 7.0 и выше) возвращает *mask*, если для всех нитей варпа, заданных в *mask*, переданное значение равно *value*, иначе возвращается 0. Значение *pred* устанавливается истинным, если все нитей из *mask* имеют одно и то же значение *value*
- В этих функциях в качестве типа T может выступать любой из следующих типов - **int**, **unsigned int**, **long**, **unsigned long**, **long long**, **unsigned long long**, **float** или **double**

Команда Shuffle

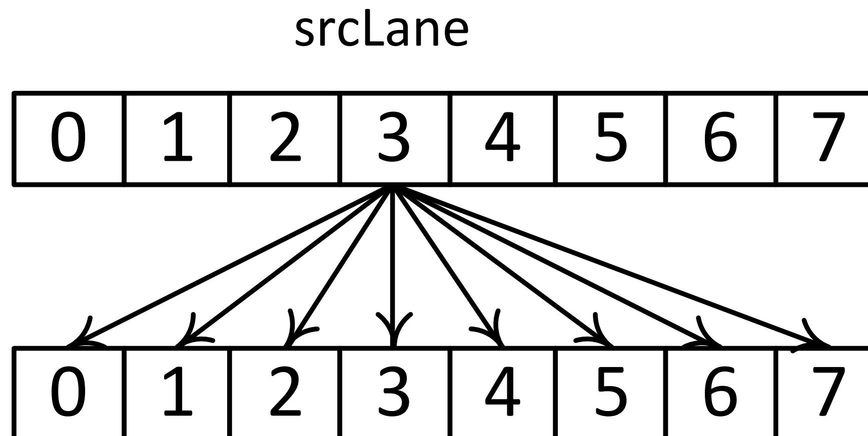
- Команды для быстрого обмена данными между нитями одного варпа

```
T __shfl_sync ( unsigned int mask, T var, int srcLane,
      int width = warpSize );
T __shfl_up_sync ( unsigned int mask, T var,
      unsigned int delta, int width = warpSize );
T __shfl_down_sync ( unsigned int mask, T var,
      unsigned int delta, int width = warpSize );
T __shfl_xor_sync ( unsigned int mask, T var, int laneMask,
      int width = warpSize );
```

`T` – int, unsigned int, long, unsigned long, long long, unsigned long long, float и double.

Команда Shuffle

- Lane – номер нити внутри варпа
- Width – размер группы, может быть равен 1, 2, 4, 8, 16 и 32
- __shfl_sync – возвращает значение переменной из нити srcLane % width (broadcast значения между всеми нитями варпа)



Команда Shuffle

- __shfl_down_sync – вычисляет номер нити, из которой будет взято значение, путем прибавления delta к номеру текущей нити
- __shfl_up_sync – вычисляет номер нити, из которой будет взято значение, путем вычитания delta из номера текущей нити
- __shfl_xor_sync – вычисляет номер нити, из которой будет взято значение, путем побитовой операции XOR с параметром laneMask к номеру текущей нити