

# Параллельное и распределенное программирование.

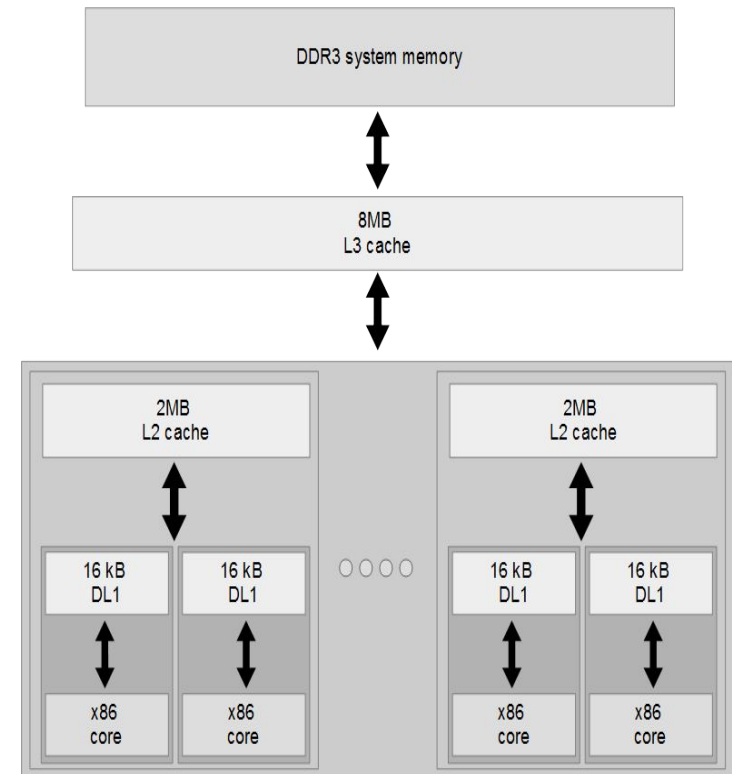
Технология программирования  
гетерогенных  
систем OpenCL.

## Лекция 3

# Архитектура GPU

# Традиционная архитектура CPU

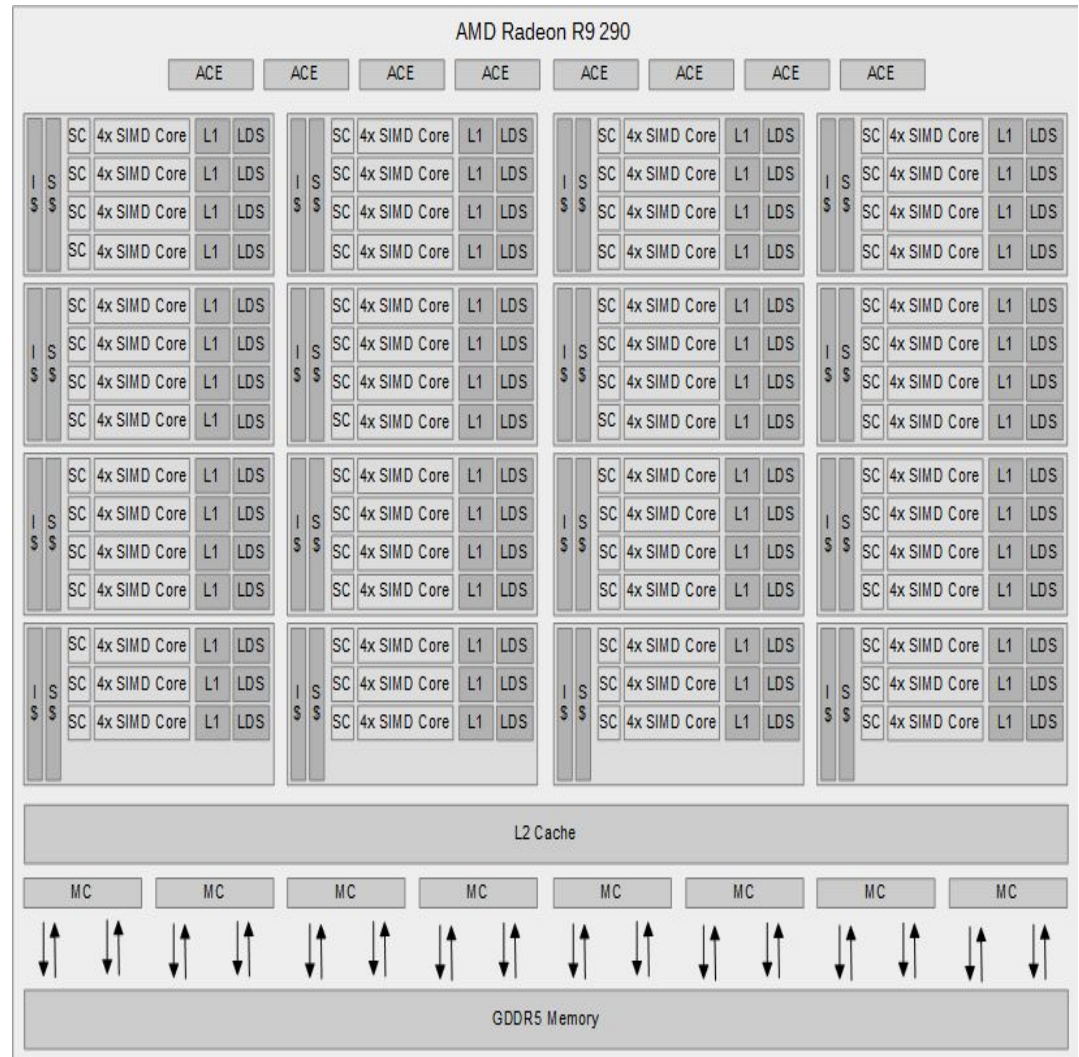
- CPU оптимизированы для минимизации задержки одного потока
  - Может эффективно обрабатывать рабочие нагрузки с интенсивным потоком управления
- Большая часть аппаратного пространства занято логикой кэширования и управления
  - Многоуровневые кэши, используемые для предотвращения задержки
- Ограниченное количество регистров из-за меньшего количества активных потоков



Example Piledriver-based CPU architecture

# Современная архитектура GPGPU

- Массив независимых «ядер», называемых Compute Units (AMD) или потоковых мультипроцессоров (NVIDIA)
- Высокая пропускная способность, кэшированные L2 кэши и основная память
  - Банки позволяют осуществлять множественный доступ параллельно
  - 100 ГБ / с
- Память и кэши, как правило, не являются когерентными



# Современная архитектура GPGPU

- Вычислительные устройства основаны на оборудовании SIMD
  - Как у AMD, так и у NVIDIA есть 16-ти SIMD-секции
- Большие файлы регистров используются для быстрого переключения контекста
  - Состояние сохранения / восстановления
  - Данные постоянны для выполнения всего потока
- Оба производителя (AMD и NVIDIA ) имеют комбинацию автоматического кеша L1 и управляемой локальной памятью
  - Scratchpad называется локальной коллекцией данных (LDS) от AMD и разделяемой памятью от NVIDIA
- Блок Scratchpad имеет большую плотность и очень высокую пропускную способность ( $\sim \text{TB} / \text{s}$ )

# Современная архитектура GPGPU

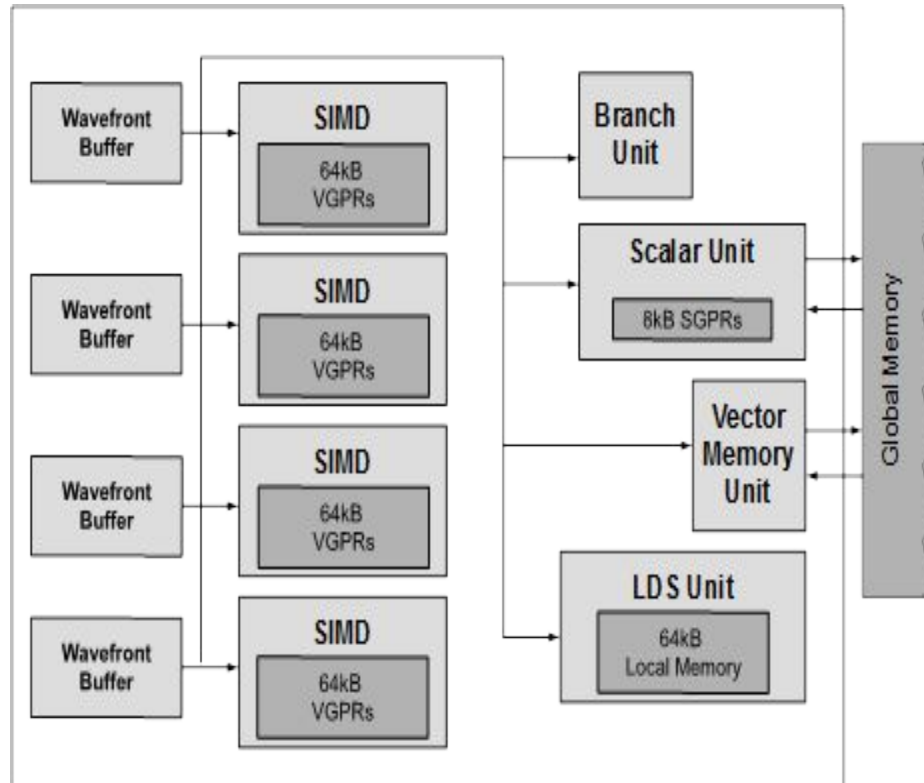
- Рабочие элементы автоматически группируются в аппаратные потоки, называемые «wavefronts» (AMD) или «warps» (NVIDIA)
  - Один поток команд, выполняемый на SIMD-оборудовании
  - 64 рабочих элемента в wavefront, 32 в warp
- Управляющий поток обрабатывается путем маскировки SIMD-секции

# SIMT и SIMD

- NVIDIA придумала «Single Instruction Multiple Threads» (SIMT) для обозначения нескольких (программных) потоков, совместно используемых потоком команд
- Хотя каждый рабочий элемент имеет свой собственный счетчик программ (ПК), они выполняют блокировку на SIMD-оборудовании
  - Несколько потоков программного обеспечения выполняются на одном аппаратном потоке
- Расхождение между потоками, обрабатываемыми с помощью маскирования или предикации
  - Дивергенция между рабочими элементами прозрачна для модели OpenCL и кажется, что рабочий элемент имеет свой собственный вычислитель
- Производительность сильно зависит от адаптации рабочих элементов под SIMD-обработку

# AMD R9 290X

- Архитектура вычислений



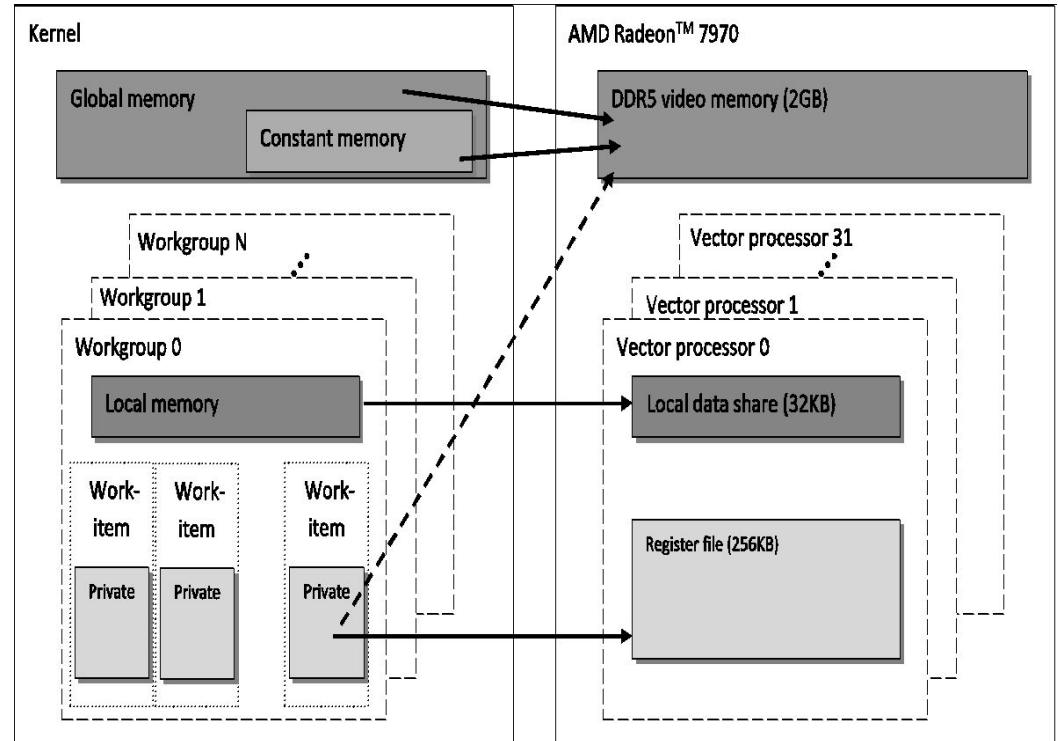


# AMD R9 290X

- Wavefronts связаны с модулем SIMD и подмножеством векторных регистров
  - С каждым SIMD может быть связано до 10 wavefronts
  - 4 SIMD
  - 40 wavefronts могут быть активны на единицу измерения
- Все аппаратные блоки, за исключением SIMD, совместно используются всеми wavefronts на вычислительном блоке

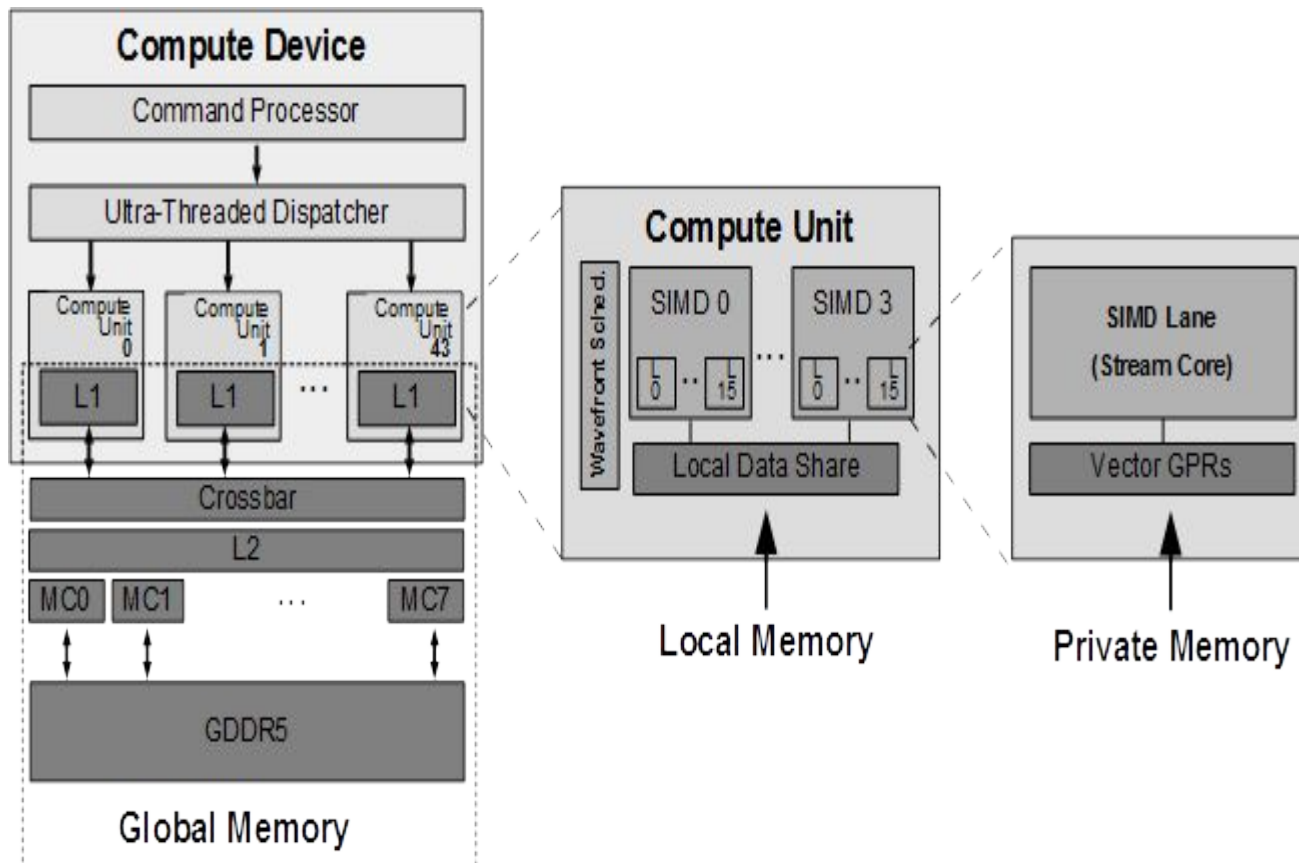
# Модель памяти в OpenCL

- Глобальная память
  - для кэширования
- Основная память GDDR5
  - Константная память
  - Память скалярного блока
  - Локальная память
- Карты в LDS
  - Общие данные между рабочими элементами рабочей группы
  - Доступ с высокой пропускной способностью для SIMD
- Закрытая память
  - Регистровая память



# AMD GPU архитектура in OpenCL

- R9 290X в OpenCL



# Идеальное ядро GPGPU

- Идеальное ядро для GPU
  - Имеет тысячи самостоятельных вычислительные единицы
    - Использует все доступные вычислительные единицы
    - Позволяет переключать контекст для скрытия латентности
  - Подходит для обмена потоками команд
    - Локальная память для выполнения SIMD предотвращает расхождение между рабочими элементами
    - Обладает высокой арифметической интенсивностью
  - Временное соотношение математических операций с доступом к памяти является высоким
    - Не ограничено полосой пропускания памяти

Буфер  
ы

# План

- Создание объектов памяти (например, буферов)
  - Параметры флага памяти
- Буферы записи и чтения
- Управление объектами памяти
- Перенос объектов памяти
- Память, доступная для хоста

# Буферы

- Объекты памяти используются для передачи больших структур данных в ядра OpenCL
- Наиболее прямым объектом является буфер
  - Буфер представляет собой непрерывную последовательность адресных элементов, подобных массиву C
- На основе флагов памяти предоставляется опциональный указатель хоста для инициализации буфера или даже для хранения буфера
- Объект-буфер создается с помощью следующей функции:

```
cl_mem buffer = clCreateBuffer (  
    cl_context context,    // Context object  
    cl_mem_flags flags,   // Memory flags  
    size_t size,          // Bytes to allocate  
    void *host_ptr,       // Host data  
    cl_int *errcode)     // Error code
```

# Флаги памяти

- Поле флага памяти в `clCreateBuffer ()` позволяет нам определять атрибуты буферного объекта:

Флаг	Поведение
<code>CL_MEM_READ_WRITE</code>	Указывает типы доступа, разрешенные ядром
<code>CL_MEM_WRITE_ONLY</code>	
<code>CL_MEM_READ_ONLY</code>	
<code>CL_MEM_USE_HOST_PTR</code>	Используйте указатель хоста в качестве хранилища для буфера. В памяти устройства используется кешированная копия во время выполнения ядра
<code>CL_MEM_ALLOC_HOST_PTR</code>	Выделяет буферное хранилище в доступной памяти хоста.
<code>CL_MEM_COPY_HOST_PTR</code>	Инициализировать буфер данными из ссылки <code>host_ptr</code> .
<code>CL_MEM_HOST_WRITE_ONLY</code>	Указывает типы доступа, разрешенные хостом для объекта памяти.
<code>CL_MEM_HOST_READ_ONLY</code>	
<code>CL_MEM_HOST_NO_ACCESS</code>	



# Буферы

- Следующий код создает буфер только для чтения и инициализирует его данными из массива хоста
  - Предполагается, что контекст является допустимым контекстом OpenCL

```
cl_int err;
int a[16];

cl_mem newBuffer = clCreateBuffer(
    context,
    CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
    16*sizeof(int),
    a,
    &err);

if( err != CL_SUCCESS ) {
    // Handle error as necessary
}
```

# Буферы для записи

- OpenCL предоставляет команды для чтения или записи данных из буфера
  - Использование командной очереди также позволяет среде выполнения копировать буфер на устройство.
  - Создает событие для зависимостей или может быть использован блокирующий вызов
- Как только команда будет завершена, указатель хоста может быть повторно использован
  - Программист может указать, что хранилище данных буфера объекта находится на устройстве после завершения вызова, хотя это явно не требуется в спецификации OpenCL

```
cl_int clEnqueueWriteBuffer (  
    cl_command_queue queue,           // command queue  
    cl_mem buffer,                   // buffer object  
    cl_bool blocking_write,          // blocking flag  
    size_t offset,                   // offset into buffer to write  
    size_t cb,                       // size of data to write  
    void *ptr,                       // pointer to source data  
    cl_uint num_in_wait_list,        // number of events to wait for  
    const cl_event * event_wait_list, // array of events to wait for  
    cl_event *event)                 // event for this command
```

# Буферы для записи

- Пример, показывающий создание и инициализацию буфера

```
cl_int err;
int a[16];

for (int i = 0; i < SIZE; i++) {
    a[i] = i;
}

// Create the buffer
cl_mem buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, SIZE*sizeof(int), a, &err);

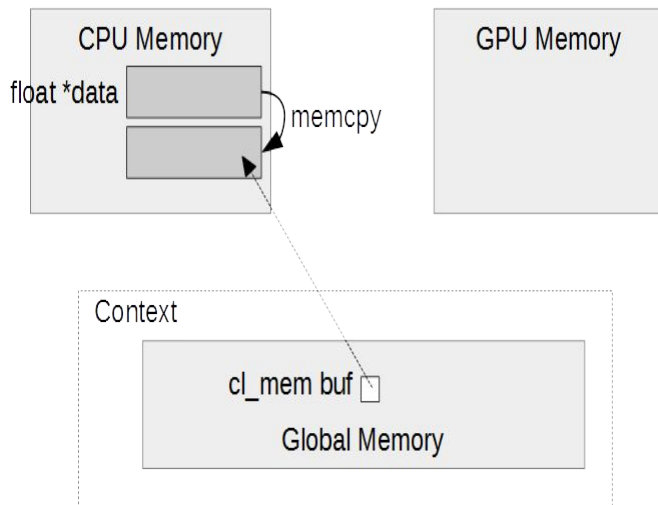
if( err != CL_SUCCESS ) { // Handle error as necessary }

// Initialize the buffer
err = clEnqueueWriteBuffer (
    queue,
    buffer, // destination
    CL_TRUE, // blocking write
    0, // don't offset into buffer
    SIZE*sizeof(int), // number of bytes to write
    a, // host data
    0, NULL, // don't wait on any events
    NULL); // don't generate an event
```

# Буферы для записи

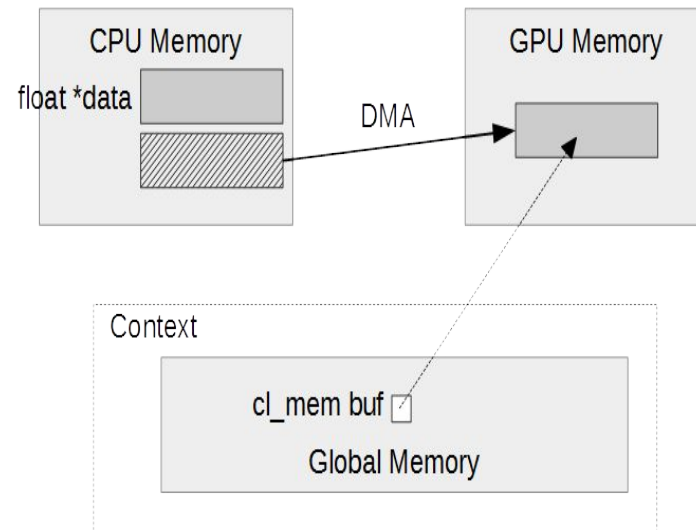
- В следующем примере показано создание и инициализация буфера, использование его в ядре без явного написания буфера

```
cl_mem buf = clCreateBuffer(...,  
    CL_MEM_COPY_HOST_PTR,data,...);
```



а) Создание и инициализация буфера в памяти хоста (инициализация выполняется с использованием CL\_MEM\_COPY\_HOST\_PTR).

```
clSetKernelArg(..., &buf);  
clEnqueueNDRangeKernel (gpuQueue,...);
```

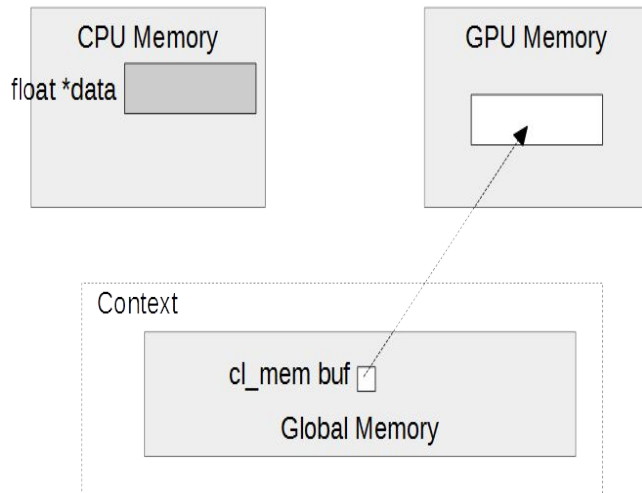


б) Неявная передача данных с хоста на устройство до выполнения ядра. Среда выполнения также может выбрать, чтобы устройство получало доступ к буферу непосредственно из памяти хоста.

# Буферы для записи

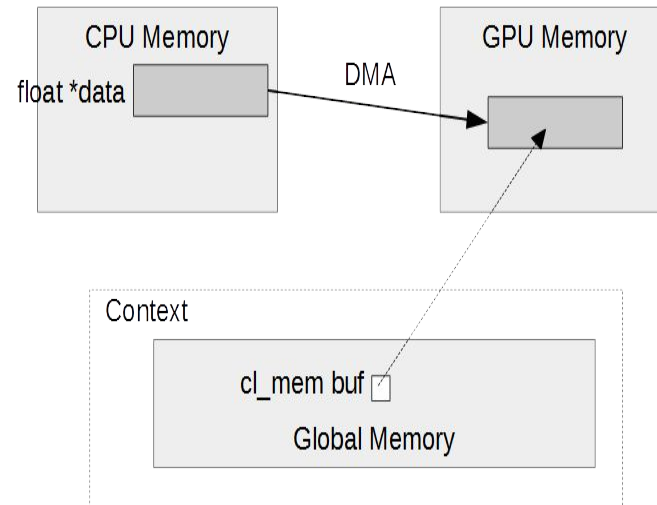
- В качестве альтернативы среда выполнения может решить создать буфер непосредственно в памяти устройства

```
cl_mem buf = clCreateBuffer(...);
```



а) Создание буфера в памяти устройства (по усмотрению среды выполнения)

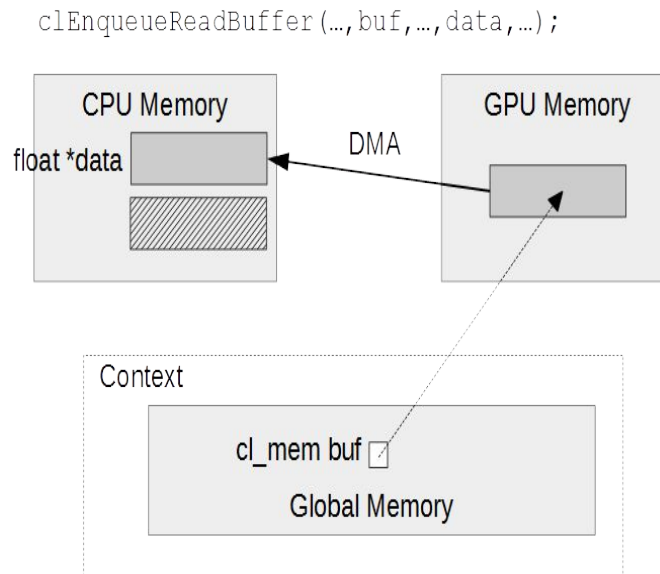
```
clEnqueueWriteBuffer(..., buf, ..., data, ...);
```



б) Копирование данных хоста в буфер непосредственно в памяти графического процессора

# Буферы для чтения

- Бесплатный вызов для записи буфера (буфер считывается обратно в память хоста)
  - Следующая диаграмма предполагает, что данные буфера были перенесены на устройство по времени выполнения



с) Чтение выходных данных из буфера обратно в память хоста (продолжение с предыдущего слайда)



# Память, доступная с хоста

- При создании объекта памяти флаги могут указывать, что объект должен быть создан в доступной для хоста памяти
  - требует выделения в месте, которое может быть отображено в адресное пространство хоста
- `CL_MEM_ALLOC_HOST_PTR`
  - сообщает рабочей среде о создании буфера в доступной для хоста памяти
- `CL_MEM_USE_HOST_PTR`
  - Сообщает, что среда выполнения использует указатель на хост-носитель в качестве хранилища для буфера
- Для обоих вариантов реализация будет иметь доступ к буферу из памяти хоста (ЦП)
  - Это обычно называют памятью с нулевой копией
  - Это явно не требуется в спецификации OpenCL



# Память, доступная с хоста

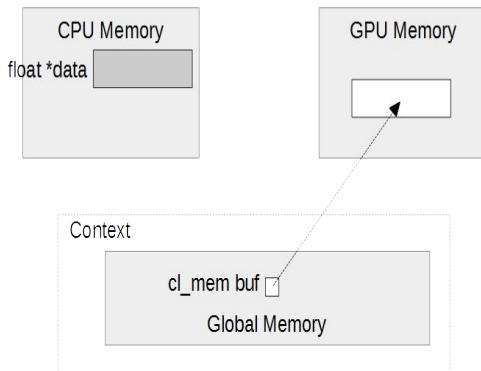
- Специальная обработка флагов (AMD)
- CL\_MEM\_ALLOC\_HOST\_PTR и CL\_MEM\_USE\_HOST\_PTR
  - Если устройства поддерживают виртуальную память, хранилище будет создано в виде закрепленной памяти хоста и доступ к ней как к нулевым копиям
  - Без виртуальной памяти на устройстве будет выделено хранилище
- CL\_MEM\_USE\_PERSISTENT\_MEM\_AMD
  - Специфическое для AMD расширение
  - Доступ к этому объекту памяти с хоста происходит непосредственно из памяти устройства
- Когда указываете, где должны храниться данные, необходимо оценить все последствия



# Отображение данных в память хоста

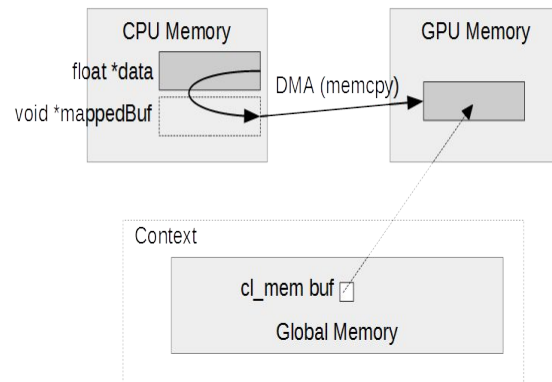
- В приведенном ниже примере показан один возможный сценарий для функций `map` и `unmap`

```
cl_mem buf = clCreateBuffer(...);
```



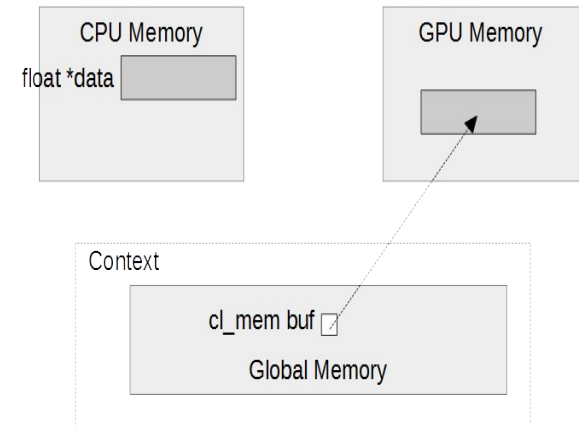
a) Создание неинициализированного буфера в памяти устройства

```
void *mappedBuf = clEnqueueMapBuffer(..., buf, ...);  
memcpy(data, mappedBuf, ...);
```



b) Отображение буфера в адресное пространство хоста. Получение доступа к буферу с помощью указателя узла.

```
clEnqueueUnmapMemObject(..., buf, mappedBuf, ...);
```



c) Unmap буфера из адресного пространства хоста

# Выводы

- Объекты памяти создаются и управляются хостом с помощью вызовов OpenCL API
- Параметры, предоставленные во время создания, могут использоваться для описания намерений программиста во время выполнения
- Расположение данных влияет на время выполнения

Изображения и  
каналы

Images и Pipes  
в OpenCL 2.0

# План

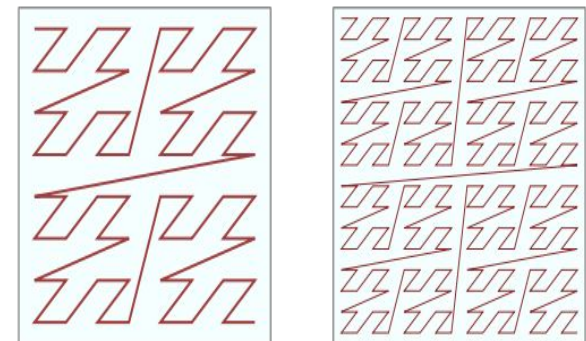
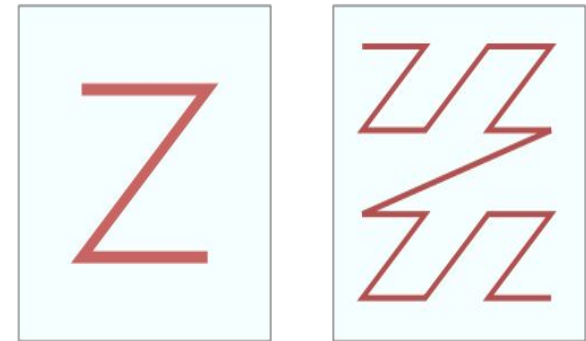
- Объекты памяти, введенные в спецификации OpenCL 2.0
  - Изображения
  - Каналы (pipes)
- Объекты памяти специального назначения и отличаются от C-подобных буферов
  - Изображения - абстрактная модель памяти для обеспечения простоты и оптимизации
  - Каналы используются для отправки данных между экземплярами ядра в FIFO порядке

# Изображения

- Массивы C / C ++ и объекты буфера OpenCL (cl\_mem) обеспечивают 1D локальность
- Графические процессоры содержат аппаратную поддержку для:
  - Кэширование и чтение многомерных данных (текстур)
  - Рисование интерполированных текстурных вершин
- Аппаратная поддержка этих функций предоставляется программистам через OpenGL-изображения
  - Изображения OpenGL - это объекты памяти, оптимизированные для 2D-локации
- Смежные элементы не гарантируют непрерывности в памяти
  - Структура кривой Z в текстурах в памяти обеспечивает двумерную локальность данных



C/C++ 1D locality (row major) layout



Z Curve - 2D locality in layout

# Изображения

- Изображения специально разработаны для управления объектами графических данных
- Элементы изображений не могут быть доступны непосредственно из ядра
- Изображения отличаются от буферов данных следующими признаками:
  - Непрозрачный тип данных, который нельзя просматривать напрямую из устройства
  - Многомерные структуры
  - Ограничено набором типов данных, относящихся к графике
- Специальные ВУ обеспечивают различные операции, такие как преобразование данных и фильтрация
- В ядрах OpenCL вместо функции простого индексирования используется вызов функции `read_image {type}`, используя `[]`
  - Использование `read_image` и `write_image` описано дальше



# Преимущества изображений

- Интерполяция
  - Доступ к изображениям осуществляется с помощью координат с плавающей запятой
  - Возвращается любой ближайший пиксель или выполняется линейная интерполяция
    - Указано в объекте `cl_sampler`
    - `CL_FILTER_NEAREST` (без интерполяции)
    - `CL_FILTER_LINEAR` (линейная интерполяция)
- Нормализованные типы данных
  - Уменьшает объем используемой памяти, поскольку эти типы данных сохраняют float как 16- или 8-битное целое число в текстуре
  - Использовать float в нормализованном диапазоне [0.0-1.0] (беззнаковые типы), [-1.0-1.0] (знаковые типы)

# Преимущества изображений

- Обработка исключений
  - Поведение доступа за пределами границ обрабатывается аппаратными средствами
  - Флаги, указанные при создании `cl_sampler`
    - Примеры
    - `CLK_ADDRESS_CLAMP` - возврат 0
    - `CLK_ADDRESS_CLAMP_TO_EDGE` - вернуть цвет пикселя, ближайшего к местоположению вне пределов

# Преимущества изображений

- Каналы в изображениях OpenCL относятся к основным цветам, которые составляют изображение
  - Каждый пиксель в текстуре может содержать от 1 до 4 каналов (от R до RGBA)
  - RGBA: красный, зеленый, синий, альфа
  - Информация о цвете хранится как данные с плавающей запятой / целыми числами
- При упаковке нескольких значений (каналов) в пикселе это может улучшить использование полосы пропускания памяти.
- Количество каналов определяется при создании изображения

# Создание изображений

- Декларации изображений состоят из дескрипторов и форматов
- Использование изображений в ядрах требует объявления объекта сэмплера изображения

```
cl_mem clCreateImage (  
    cl_context context,           // OpenCL Context  
    cl_mem_flags flags,         // Memory Flags  
    const cl_image_format *image_format, // Image format  
    const cl_image_desc *image_desc, // Image descriptor  
    void *host_ptr,             // Host Pointer  
    cl_int *errcode_ret)       // Error code
```

# Дескрипторы и форматы изображений

- Свойства изображения, указанные в структуре `cl_image_desc`, включая
  - Тип изображения - 1D, 2D или 3D изображение
  - Размер изображения - ширина, высота и глубина
  - Шаг строки и среза
- Формат изображения, описанный в структуре `cl_image_format`, определяет свойства канала и тип данных каждого элемента в канале
  - Свойства канала: количество каналов и макет памяти, в которых каналы хранятся в изображении
- Более подробный синтаксис можно найти в спецификации

# Сэмплер изображения

## (Image sampler )

- Image sampler описывает, как получить доступ к объекту изображения
- Сэмплеры указывают:
  - Тип системы координат для доступа к изображению
  - Параметры для обращения за пределами доступа
  - Параметры интерполяции, если доступ лежит между несколькими индексами
- Сэмплер передается ядру так же, как и обычные аргументы ядра, или может быть объявлен в ядре
- Создание сэмплера на хосте выполняется с помощью вызова `clCreateSampler`

```
cl_sampler  clCreateSampler (
    cl_context context,           // OpenCL context
    cl_bool  normalized_coords,  // Use normalized coords?
    cl_addressing_mode addressing_mode, // Out-of-bounds access behavior
    cl_filter_mode filter_mode,  // Interpolation behavior
    cl_int  *errcode_ret)       // Error code
```

# Использование изображений в ядре

- Форматы изображений не такие, как базовые типы OpenCL (int, float, char и т. д.).
- Квалификаторы типов, используемые для изображений: `image1d_t`, `image2d_t` и `image3d_t`



# Использование изображений в ядре

```
__kernel
void rotation(
    __read_only image2d_t inputImage, __write_only image2d_t outputImage,
    int imageWidth, int imageHeight, float theta)
{
    // Declaring sampler
    __constant sampler_t sampler =
        CLK_NORMALIZED_COORDS_FALSE | CLK_FILTER_LINEAR | CLK_ADDRESS_CLAMP;

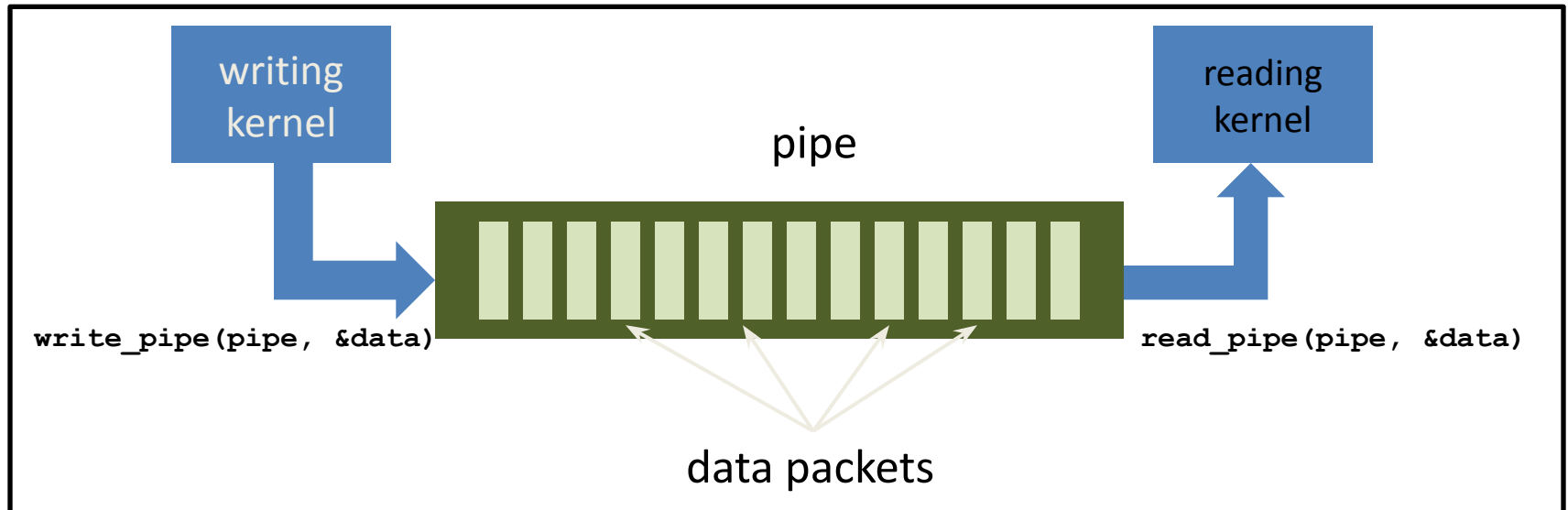
    /* Read the input image */
    float value;
    value = read_imagef(inputImage, sampler, readCoord).x;

    /* Write the output image */
    write_imagef(outputImage, (int2)(x, y), (float4)(value, 0.f, 0.f, 0.f));
}
```



# Каналы (Pipes)

- Новый объект памяти, введенный в спецификации OpenCL 2.0
- Данные организованы как пакеты в структуре FIFO
- У Pipes есть один экземпляр ядра, который является писателем и другим экземпляром ядра, который является читателем
  - Одно и то же ядро не может писать и читать канал
  - Согласование памяти выполняется в точках синхронизации



# Каналы (Pipes)

- Данные в канале организованы как пакеты
  - Пакет может иметь тип, поддерживаемый OpenCL C
  - Глубина канала определяется как количество пакетов, поддерживаемых каналом
- К каналу можно получить доступ через код ядра на устройстве
- Хост создает объект pipe, используя `clCreatePipe()`
  - Передается как обычный аргумент ядра
  - Хост не разрешает читать или записывать данные в объект pipe

```
clCreatePipe(  
    cl_context context,           // Context  
    cl_mem_flags flags,         // Flags same as buffer  
    cl_uint pipe_packet_size,   // Packet size  
    cl_uint pipe_max_packets,   // Pipe depth  
    const cl_pipe_properties *properties, // Pipe properties  
    cl_int *errcode_ret)        // Error code
```

# Каналы (Pipes)

- В ядре каналы должны быть объявлены с использованием ключевого слова `pipe`, квалификатора доступа (`read_only` или `write_only`) и тип данных пакетов
- Пример сигнатуры ядра, который считывает `pipe` типа `int` и записывает в `pipe` типа `float4`:

```
kernel
void foo(read_only pipe int input_pipe,
         write_only pipe float4 output_pipe)
```

# Каналы (Pipes)

- Подобно изображениям, каналы представляют собой непрозрачные объекты
- Доступ к каналу в ядре выполняется с помощью встроенных функций OpenCL C
- Существует несколько способов доступа к каналу; самым основным является использование функций
  - `int write_pipe (pipe p, gtype * ptr)`
  - `int read_pipe (pipe p, gtype * ptr)`
- Оба вызова принимают объект `pipe` и указатель в качестве параметра
- Оба вызова возвращают 0 при успешном завершении

# Pipes – идентификатор

## резервирования

- Существуют функции OpenCL для резервирования места в канале заранее
  - Доступ гарантируется
  - Эти функции возвращают идентификаторы резервирования (`reserve_id_t`), которые определяют местоположения в пределах канала
- Несколько пакетов могут быть зарезервированы для одного и того же идентификатора резервирования, используя параметр `num_packets`

```
reserve_id_t  
reserve_read_pipe(  
    pipe gentype p,  
    uint num_packets)|
```

```
reserve_id_t  
reserve_write_pipe(  
    pipe gentype p,  
    uint num_packets)
```

# Pipes – идентификатор резервирования

- Идентификаторы резервирования передаются в перегруженные версии `read_pipe ()` и `write_pipe ()`
  - дополнительно должен быть указан индекс, указывающий местоположение пакета в зарезервированном пространстве
- Когда используются идентификаторы резервирования, для обеспечения успешного завершения операции требуется дополнительный блокирующий вызов (`commit_read_pipe ()` или

```
cor int
```

```
read_pipe(  
    pipe gentype p,  
    reserve_id_t reserve_id,  
    uint index,  
    gentype *ptr)
```

```
void
```

```
commit_read_pipe(  
    pipe gentype p,  
    reserve_id_t reserve_id)
```

# Каналы (Pipes)

- Дополнительные версии вызовов резервирования и фиксации каналов существуют в детализации рабочей группы
  - `work_group_reserve_read_pipe ()` и `work_group_reserve_read_pipe ()`
  - `work_group_commit_read_pipe ()` и `work_group_commit_read_pipe ()`
- Существуют также вызовы OpenCL C для определения количества пакетов в канале (`get_pipe_num_packet ()`) и размера канала (`get_pipe_max_packets ()`)

# Выводы

- Изображение представляет собой тип данных, который позволяет эффективно выполнять операции
  - интерполирование
  - нормализации
  - Bounds-обращение
- Каналы упрощают передачу данных между экземплярами ядра для определенных классов алгоритмов
- Изображения и каналы являются непрозрачными типами, доступными только по внутренним функциям OpenCL C