# OpenCL Extensions

A Collaboration Between
David Kaeli, Northeastern University
Benedict R. Gaster, AMD

# Instructor Notes

- OpenCL extensions allow for a vendor to expose device functionality without concern for the specification
  - Different categories of extensions are discussed. (Khronos approved, External extensions and Vendor specific)
- Enabling extensions done in host code by developer
  - Always check for availability of required extensions (simple C example given) before running program in order to maintain device compatibility
- This lecture is covers a wide range of OpenCL extensions
  - For teaching purposes only a subset of the extensions are necessary to provide a programmer with an idea of how extensions can be used as tools for additional performance on different devices
  - A summary table and a list of all extensions with its supported vendors is provided at the end of lecture.
    - Useful reference for extensions provided by each device

# Topics

- What are OpenCL extensions ?
- Checking for extension support in OpenCL code
- Explanation of individual OpenCL extensions
  - Khronos Approved Extensions
  - AMD Specific Extensions
  - Nvidia Specific Extensions
  - Cell BE Specific Extensions

# OpenCL Extensions

- An OpenCL Extension is a feature, which might be supported by a device but is not a part of the OpenCL specification
- Extensions allow vendors to expose device specific features without being concerned about compatibility with specification and other vendor features
- A vast number of OpenCL extensions are specified by different vendors
- In this lecture we only touch on each extension to provide a feel for its usefulness and its features

# Types of extensions:

- Approved by Khronos OpenCL Working Group:
  - “cl_khr” in extension names
  - Approved conformance tests
  - Might be promoted to required Core feature in later versions
  - e.g.: Extensions for atomic operations
- External Extensions
  - “cl_ext” in extension name
  - Developed by 2 or more members of the working group
  - No required conformance tests
  - e.g.: Device Fission (cl_ext_device_fission)
- Vendor Specific Extensions
  - Developed by a single vendor
  - e.g.: AMD printf

# Using and Checking Extensions

- OpenCL extensions have to be enabled in kernel code

  #pragma OPENCL EXTENSION `extension_name` : enable

- Initial state: All extensions **disabled**
  - Error and warning reporting done according to specification
  - Programmer's responsibility to specify what extensions his code needs

- Known target device will not be known till runtime
  - Check device and possibly have a fall-back version because code using any extension will compile as long as the pragma is added to the kernel

- Application can query device for information about extensions using `CL_DEVICE_EXTENSIONS` parameter

# Checking for Extensions

- Steps to check for the availability of an extension
- Query device using `CL_DEVICE_EXTENSIONS` parameter
- Names of extensions supported by device returned in a character array
- Search in array for required extension

```
size_t op_size;
//Get Size of Extension Array (op_size)
ciErrNum = clGetDeviceInfo (
        device, CL_DEVICE_EXTENSIONS,
        0, NULL, &op_size );
```

```
char* extensions= (char*)malloc(op_size);
//Populate array with all available extensions
ciErrNum = clGetDeviceInfo(
        device, CL_DEVICE_EXTENSIONS,
        op_size, extensions,
        &op_size);
```

```
// Search for required extension in array
if( !    strstr ( deviceExtensions,
         "cl_khr_byte_addressable_store") )
                 error("Extension Unavailable")
```

# Khronos Approved Extensions

# Atomic Operations

- Atomic operations are operations performed in memory without interference from any other threads
- Multiple threads can update the same location in memory
- Ordering of updates is undetermined but all updates are guaranteed to occur successfully
- Used to prevent race conditions in applications that involve binning like histograms.
- Atomics possible for both local and global memory
- Atomics presently supported on AMD 5000 series GPUs and on Nvidia GPUs of compute capability 1.1 and higher

# Atomic Operations in OpenCL

- Atomic operations are only guaranteed for a single device executing these atomic functions.
  - Atomics on the same location by multiple devices are not possible
- 32 Bit Integer Atomic Operations for local and device memory are specified in separate extensions.
  - cl_khr_{global | local}_int32_base_atomics
  - cl_khr_{global | local}_int32_extended_atomics
- 64 Bit Integer Atomic Operations – both local and device support expressed in one extension
  - cl_khr_int64_base_atomics
  - cl_khr_int64_extended_atomics
- The local memory atomics operate on data in local memory are atomic only within a single work group
- Atomic operations provide no ordering guarantees, they only guarantee that all operations will complete successfully

# Base Atomic Operation Set

| Function Name | Description |
|---|---|
| int atom_add (__global int * p, val) | Atomically add val to data at location pointed to by p and return old value |
| int atom_sub (__global int *p, int val) | Atomically subtract val to data at location pointed to by p and return old value |
| int atom_xchg (__global int *p, int val) | Swaps the old value pointed to by p with val |
| int atom_inc (__global int *p) | Atomically increment the 32-bit value at location pointed by p. Return the old value |
| int atom_dec (__global int *p) | Atomically decrement the 32-bit value at location pointed by p. Return the old value |
| int atom_cmpxchg (__global int *p, int cmp, int val) | Compare val with data at location p and exchange if not equal. Return old value |

# Extended Atomic Operations

| Function Name | Description |
|---|---|
| int atom_min (__global int *p, int val) | Store min(val, *p) at location pointed to by p. Returns original value |
| int atom_max (__global int *p, int val) | Store max(val, *p) at location pointed to by p. Returns original value |
| int atom_and (__global int *p, int val) | Store and(val, *p) at location pointed to by p. Returns original value |
| int atom_or (__global int *p, int val) | Store or(val, *p) at location pointed to by p. Returns original value |
| int atom_xor (__global int *p, int val) | Store xor(val, *p) at location pointed to by p. Returns original value |

- The local memory atomic operations follow exactly the same function call convention with the exception that the pointer provided is a local memory pointer
- Global atomic min and max should not be used like reductions to find minima and maxima in an array because they return the previous max / minima to work item
- Kernel would have to complete and only then can global maxima / minima be read

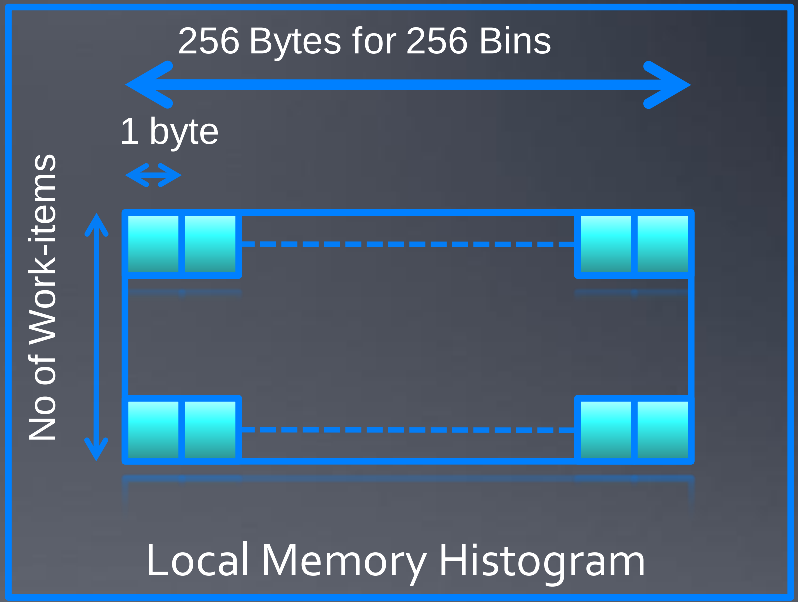
# Double and Half Precision

- OpenCL 1.0 provides double precision floating-point as an optional extension (cl_khr_fp64)
- Enable extension by using directive in kernel file
  - Double precision vectors of types double{2,4,8,16}
- AMD support for this extension is partial (as of SDK v2.2)
  - Does not guarantee that built in functions implemented would be considered conformant to the cl_khr_fp64 extension
  - Support expressed as vendor extension cl_amd_fp64
- Nvidia provides support conformant to the extension
- Half precision is defined using the extension cl_khr_fp16 but not supported by AMD or Nvidia

Source: http://www.khronos.org/registry/cl/extensions/amd/cl_amd_fp64.txt

# Byte addressable store

- In OpenCL, sub 32 bit writes to types like char are not supported.
- This extension allows writes on sub 32 bit built-in types
- Example – 256 Bin Histogram
  - The OpenCL histogram example uses 1 byte per bin when building a 256 bin per thread histogram
  - Using a uint per bin per thread would restrict the work group size to 64
- NOTE: This is a different technique from the Nvidia warp voting method discussed in Lecture 07




- Work-item handles 256 numbers and builds a per thread histogram in local memory
- After building per work-item histogram, the per workgroup histogram is built and written to device memory

# 3D Image Write Extensions

- OpenCL provides support for 2D Image read and write
- Writes to a 3D image memory object is not allowed in OpenCL 1.0 specification
- The `cl_khr_3d_image_writes` extension implements writes to 3D image memory objects
- Reads and writes to the same 3D image memory object are not allowed in a kernel.
- RGBA AND BGRA channel images only supported

Source: http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/supportedImageFormats.html

Source: http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/cl_khr_3d_image_writes.html

# OpenGL Interoperability

- The extension `cl_KHR_gl_sharing` allows applications to use OpenGL buffer, texture and render-buffer objects as OpenCL memory objects
- An OpenCL context may be created from an OpenGL context using this extension
- An OpenCL image object may be created from an OpenGL texture or render-buffer object
- An OpenCL buffer object may be created from an OpenGL buffer object
- For MacOS the extension is known as `cl_apple_gl_sharing`

Source: http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/gl_sharing.html

# AMD Specific Extensions

# Device Fission

- An external extension developed by AMD, Apple, IBM and Intel (`cl_ext_device_fission`)
- Provides an interface for sub-dividing a device into multiple sub-devices
- Presently available for AMD / Intel multicore CPUs and the Cell Broadband Engine

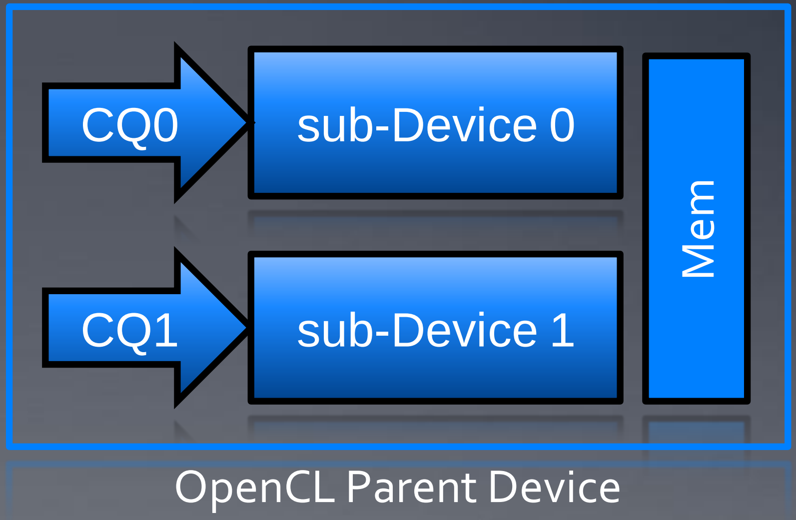
Source: http://www.khronos.org/registry/cl/extensions/ext/cl_ext_device_fission.txt

# Applications of Device Fission

- Possible uses as per specification documentation
  - Reserve a part of the device for use for high-priority/latency-sensitive tasks
  - Control for the assignment of work to individual compute units
  - Subdivide compute devices along some shared hardware feature like a cache
- Other uses of device fission include enforcing “scheduling” of work groups onto compute units of sub-devices by assigning work to sub-devices

# Device Fission

- Single OpenCL device partitioned into sub-devices
- A sub-device must have its corresponding command queue
- Partitioning a device requires knowledge of underlying architecture
- Launch different work groups to both queues



# Partitioning a Device

- Multiple ways to subdivide a device defined by extension
- CL_DEVICE_PARTITION_EQUALLY_EXT
  - Equal Partitions  of compute units in device to each sub-device
- CL_DEVICE_PARTITION_BY COUNTS_EXT
  - This property is used for an uneven distribution of numbers of compute units for each subdevice
- CL_DEVICE_PARTITION_BY NAMES_EXT
  - This property is used to create sub-devices using a list of compute unit names for each sub-device
- CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN_EXT
  - Used to partition a device as per its cache hierarchy

# Expressing Partitions

- Device Fission Extension defines a type known as "`cl_device_partition_property_ext`"
- Partitions expressed in terms of arrays of type "`cl_device_partition_property_ext`" which are a combination of constants and the partition properties
- Example: To create a three compute unit sub-device using compute units, { 0, 1, 3 } i.e. (partition by name)
  - { CL_DEVICE_PARTITION_BY_NAMES_EXT, 0, 1, 3, CL_PARTITION_BY_NAMES_LIST_END_EXT, CL_PROPERTIES_LIST_END_EXT }

Source: http://www.khronos.org/registry/cl/extensions/ext/cl_ext_device_fission.txt

# Device Fission - New Device Types

```
cl_int clCreateSubDevicesEXT(
        cl_device_id in_device,                          //Parent Device to sub-divide
        const cl_device_partition_property_ext * properties,
        cl_uint num_entries,                             //Size of out_devices (no of devices)
        cl_device_id * out_devices,                      //Pointer to list of sub-devices
        cl_uint * num_devices );                         // No of subdivided devices returned
```

- This function allows us to create sub-devices using a parent device
- Sub-device: An OpenCL device after being subdivided
- Root-device: A root device is a device that has not been sub-divided
- Parent-device: The device used to produce a sub-device

# Using Device Fission

- Other functionality has been added to OpenCL spec functions like clGetDeviceInfo that allows us to query sub-device specific properties
  - e.g.: CL_DEVICE_PARENT_DEVICE_EXT: Passed to clGetDeviceInfo to get cl_device_id of parent device
  - Many different selectors available to check for affinity settings, partition style etc
- clCreateCommandQueue: When called on a sub-device, checks if parents were used to create the context

# GPU - Printf

- GPU printf – An AMD specific extension
- Can write a format string to print GPU data to stdout
- Constraints – Format string needs to be resolved at compile time
- This extension is useful for debugging
- Note: Kernel does have to complete before the debug output can be seen, Thus printf cannot be used in the case where a kernel crashes midway

# AMD Media Operations

- Provides support for AMD media operations in OpenCL (`cl_amd_media_ops`)
- Operations commonly used in multimedia applications
  - Operations work on OpenCL vector types
- Supported Operations
  - Pack and Unpack Operations
  - Bit and Byte alignment Operations
  - Interpolation Operations
  - Sums of absolute differences

# Media Operations Summary

| Operation | Description |
|---|---|
| uint dst = amd_pack (float4 src) | Combines individual components of a float4 vector into a unsigned int by choosing the most significant 8 bits of each float |
| floatn dst = amd_unpack{i} (uintn src) | Moves 8 bits of the uint denoted by “i” to MSB and save to the $0^{th}$ element of the float vector dst. Value of i={0,1,2,3} |
| uintn amd_bytealign (uintn s0, uintn s1, uintn s2) | Bit alignment for each element of vector |
| uintn amd_bitalign (uintn s0, uintn s1, uintn s2) | Byte alignment for each element of vector |
| uintn  amd_lerp (uintn s0, uintn s1, uintn s2) | Linear interpolation |
| uintn amd_sad (uintn s0, uintn s1, uintn s2)<br>uintn amd_sadhi (uintn s0, uintn s1, uintn s2) | Calculates sums of absolute differences of each component of src0 and src1. The result of SAD is added to src2 and returned |

Source: http://www.khronos.org/registry/cl/extensions/amd/cl_amd_media_ops.txt

# Device Query and Event Handling

- The AMD device query extension (`cl_amd_device_attribute_query`) provides a means to query AMD specific device attributes.
  - Adds parameter `CL_DEVICE_PROFILING_TIMER_OFFSET_AMD` to clGetDeviceInfo.
- Using this parameter in clGetDeviceInfo returns the offset in nano-seconds between an event timestamp and Epoch.
- The `cl_amd_event_callback` extension provides more functionality for OpenCL events
  - This extension is discussed in the timing lecture

Source: http://www.khronos.org/registry/cl/extensions/amd/cl_amd_device_attribute_query.txt

# Nvidia Specific Extensions

# Nvidia Specific Extensions

- Nvidia's OpenCL extensions can be grouped into
  - Compiler Options
  - Interoperability Extensions
  - Device Query Extension

# Nvidia OpenCL Compiler Options

- Compiler Extensions in OpenCL provide a set of Nvidia platform specific options for the OpenCL compiler

- Passed from option field of clBuildProgram to the PTX assembler allowing greater control over code generation.
  - -cl-nv-maxrregcount=<N>
    - Maximum number of registers that can be use, It is a tradeoff between using more registers and better occupancy enabled by less register pressure
  - -cl-nv-opt-level=<N>
    - N = 0 indicates no optimization. The default value: 3
  - -cl-nv-verbose
    - Output will be reported in the build log

Source:
http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/OpenCL_Extensions/cl_nv_compiler_options.txt

# Loop Unrolling

- The programmer can control loop unrolling using the `cl_nv_pragma_unroll` extension
- Allows us to point out loops to be unrolled fully or partially using the “#pragma unroll k” directive
  - k denotes unrolling factor, It is only a hint and can be ignored.
  - Using only #pragma unroll specifies full unrolling
- The pragma must be specified before the respective loop as shown below. The trip count can be a variable like “n” below

```
#pragma unroll 4
for (int i = 0; i < n; i++) {
        ...
}
```

Source:
http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/OpenCL_Extensions/cl_nv_pragma_unroll.txt

# Device Query

- Device query extension `(cl_nv_device_attribute_query)` can be used to query device attributes specific to NVIDIA hardware
- Enables the programmer to optimize kernels based on the specifics of the hardware
  - clGetDeviceInfo called with parameters specific to Nvidia hardware order to query the device attributes
  - Example use case could be to use textures vs. cache for newer hardware like Fermi or to query the warp size
- Example query parameters include Nvidia GPU specific characteristics like compute capability, kernel execution timeout

Source:
http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/OpenCL_Extensions/cl_nv_device_attribute_query.txt

# Interoperability Extensions

- Interoperability extensions provided by Nvidia to support sharing of buffers and texture objects with OpenCL and DirectX in a manner similar to OpenGL interoperability extensions
- Different versions of DirectX need to be explicitly enabled
  - Direct3D implementation supporting sharing of buffer and texture objects with OpenCL is required
  - Extensions named cl_nv_d3d{9,10,11}_sharing as per version
- Allows creating special OpenCL contexts for DirectX interoperability
- Provides enhancements for OpenCL event types to handle acquiring and releasing objects

Source:http://www.khronos.org/registry/cl/extensions/nv/cl_nv_d3d9_sharing.txt

# Cell BE Extensions

# Cell Broadband Engine Extensions

- Device Fission – Discussed previously in AMD's extensions
- Subdividing a device is only possible using:
  - CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN_EXT
  - CL_AFFINITY_DOMAIN_NUMA_EXT
- Cell BE supports the byte addressable store extension
- Cell BE provides another external extension (developed by IBM and Apple) known as Memory Object Migration

Source: OpenCL Development Kit for Linux on Power – Users Guide v0.2 June, 2010

# Memory Object Migration

- Migrate memory object extension `cl_ext_migrate_memobject` defines a mechanism for assigning which device an OpenCL memory object resides
- Defines an function clEnqueueMigrateMemObjectEXT to initiate migration of an object to its compute unit
- Device fission allows an application to divide a device into sub-devices along affinity domains.
  - Fission can be used with clEnqueueMigrateMemObjectEXT to influence the association of the memory object with the specific domain
- Expressing affinity and overlapping explicitly initiated migration with other commands leads to latency hiding and improved performance in the memory bus of the Cell BE

# Extension Support Summary

| Extension Name | AMD - GPU | AMD - CPU | Nvidia | Cell |
|---|---|---|---|---|
| Atomics Local | ✔ | ✔ | ✔ | |
| Atomics Global | ✔ | ✔ | ✔ | |
| Byte Addressable | ✔ | ✔ | ✔ | ✔ |
| GPU-printf | ✔ | ✔ | | |
| Device Fission | | ✔ | | ✔ |
| Migrate Object | | | | ✔ |
| Media Operations | ✔ | ✔ | | |
| Event operations | ✔ | ✔ | | |
| Image Write | ✔ | ✔ | ✔ | |
| FP64 | ✔ | ✔ | ✔ | |
| Compiler Options | | | ✔ | |
| DirectX Interop | | | ✔ | |

**Note:** Extension support for GPUs specified for AMD 5870 with Stream SDK 2.2 and Nvidia GTX 480 with CUDA toolkit 3.1

# Summary

- AMD, Nvidia and IBM provide a vast set of OpenCL extensions which can help in a variety of applications
- Extensions allow us to take advantage of architectural features of devices that are not a part of the core OpenCL specification

# List of All OpenCL Extensions

- Khronos Approved Extensions
  - Atomic Operations
  - Sub 32 bit read-write
  - Double and half precision
  - 3D Image Write
  - OpenGL Interop
- AMD Specific Extensions
  - Device Fission
  - Media Operations
  - GPU-printf
  - Device Query and Event Callbacks
- Nvidia Specific Extensions
  - Compiler and Assembler Extensions
  - DirectX Interoperability
  - Device Attributes
- Cell BE
  - Device Fission
  - Memory Object Migration