OpenCL
Parallel Computing for Heterogeneous Devices

Ofer Rosenberg
Visual Computing Software Division, Intel

Based on Khronos OpenCL Overview by Aaftab Munshi
Welcome to OpenCL

- With OpenCL you can...
- Leverage CPUs, GPUs, other processors such as Cell/B.E. processor and DSPs to accelerate parallel computation
- Get dramatic speedups for computationally intensive applications
- Write accelerated portable code across different devices and architectures
What You’ll Learn

- **What is OpenCL?**
  - Design Goals
  - The OpenCL Platform, Execution and Memory Model

- **How to use OpenCL**
  - Setup
  - Resource Allocation
  - Execution and Synchronization

- **Programming with OpenCL C**
  - Language Features
  - Built-in Functions
What Is OpenCL?
Design Goals of OpenCL

- Use all computational resources in system
  - CPUs, GPUs, and other processors as peers
  - Data- and task- parallel compute model
- Efficient parallel programming model
  - Based on C99
  - Abstract the specifics of underlying hardware
- Specify accuracy of floating-point computations
- Desktop and Handheld Profiles
OpenCL Platform Model

- A host connected to one or more OpenCL devices
- An OpenCL device is
  - A collection of one or more compute units (cores)
  - A compute unit is composed of one or more processing elements
  - Processing elements execute code as SIMD or SPMD
OpenCL Execution Model

- **Kernel**
  - Basic unit of executable code - similar to a C function
  - Data-parallel or task-parallel

- **Program**
  - Collection of kernels and other functions
  - Analogous to a dynamic library

- **Applications queue kernel execution instances**
  - Queued in-order
  - Executed in-order or out-of-order
Expressing Data-Parallelism in OpenCL

- Define N-dimensional computation domain (N = 1, 2 or 3)
  - Each independent element of execution in N-D domain is called a work-item
  - The N-D domain defines the total number of work-items that execute in parallel
- E.g., process a 1024 x 1024 image: **Global problem dimensions**: 1024 x 1024 = 
  **1 kernel execution per pixel**: 1,048,576 total kernel executions

```c
void scalar_mul(int n,
    const float *a,
    const float *b,
    float *result)
{
    int i;
    for (i=0; i<n; i++)
        result[i] = a[i] * b[i];
}

// execute scalar_mul over n work-items
```

```c
kernel void
dp_mul(global const float *a,
    global const float *b,
    global float *result)
{
    int id = get_global_id(0);
    result[id] = a[id] * b[id];
}
// execute dp_mul over “n” work-items
```
Expressing Data-Parallelism in OpenCL

• Kernels executed across a global domain of work-items
  - Global dimensions define the range of computation
  - One work-item per computation, executed in parallel

• Work-items are grouped in local workgroups
  - Local dimensions define the size of the workgroups
  - Executed together on one core
  - Share local memory and synchronization

• Caveats
  - Global work-items must be independent: no global synchronization
  - Synchronization can be done within a workgroup
Global and Local Dimensions

- Global Dimensions: 1024 x 1024 (whole problem space)
- Local Dimensions: 128 x 128 (executed together)

Choose the dimensions that are “best” for your algorithm

Synchronization between work-items possible only within workgroups: barriers and memory fences

Cannot synchronize outside of a workgroup

* Other names and brands may be claimed as the property of others. Copyright © 2009, Intel Corporation.
Expressing Task-Parallelism in OpenCL

- Executes as a single *work-item*
- A kernel written in OpenCL C
- A native C / C++ function
OpenCL Memory Model

- **Private Memory**
  - Per work-item

- **Local Memory**
  - Shared within a workgroup (16Kb)

- **Local Global/Constant Memory**
  - Not synchronized

- **Host Memory**
  - On the CPU

Memory management is explicit
You must move data from host → global → local and back
Compilation Model

- OpenCL uses dynamic compilation model (like DirectX and OpenGL)
- Static compilation:
  - The code is compiled from source to machine execution code at a specific point in the past.
- Dynamic compilation:
  - Also known as runtime compilation
  - Step 1: The code is compiled to an Intermediate Representation (IR), which is usually an assembler of a virtual machine.
  - Step 2: The IR is compiled to a machine code for execution. This step is much shorter.

In dynamic compilation, step 1 is done usually once, and the IR is stored. The App loads the IR and does step 2 during the App’s runtime.
Using OpenCL
OpenCL Objects

- **Setup**
  - **Devices** — GPU, CPU, Cell/B.E.
  - **Contexts** — Collection of devices
  - **Queues** — Submit work to the device

- **Memory**
  - **Buffers** — Blocks of memory
  - **Images** — 2D or 3D formatted images

- **Execution**
  - **Programs** — Collections of kernels
  - **Kernels** — Argument-execution instances

- **Synchronization/profiling**
  - **Events**
OpenCL Framework

```
__kernel void
dp_mul(global const float *a,
global const float *b, 
global float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
```

Compile code
Create data & arguments
Send to execution
Setup

1. Get the device(s)
2. Create a context
3. Create command queue(s)
   - Choose type: In-Order or Out-of-order

```c
cl_uint num_devices_returned;
cl_device_id devices[2];
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &devices[0], &num_devices_returned);
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &devices[1], &num_devices_returned);
cl_context context;
context = clCreateContext(0, 2, devices, NULL, NULL, &err);
cl_command_queue queue_gpu, queue_cpu;
queue_gpu = clCreateCommandQueue(context, devices[0], 0, &err);
queue_cpu = clCreateCommandQueue(context, devices[1], 0, &err);
```
Setup: Notes

- **Devices**
  - Multiple cores on a CPU or a GPU are presented as a single device
  - OpenCL executes kernels across all cores in a data-parallel manner

- **Contexts**
  - Enable sharing of memory between devices
  - To share between devices, both devices must be in the same context

- **Queues**
  - All work submitted through queues
  - Each device must have a queue
Choosing Devices

- A system may have several devices—which is best?
- The “best” device is algorithm- and hardware-dependent

Query device info with: `clGetDeviceInfo(device, param_name, *value)`
  - Number of compute units `CL_DEVICE_MAX_COMPUTE_UNITS`
  - Clock frequency `CL_DEVICE_MAX_CLOCK_FREQUENCY`
  - Memory size `CL_DEVICE_GLOBAL_MEM_SIZE`
  - SP/DP Precision Capability `CL_DEVICE_SINGLE_FP_CONFIG`  
    `CL_DEVICE_DOUBLE_FP_CONFIG`
  - Extensions (double precision, atomics, etc.)

- Pick the best device for your algorithm
Memory Resources

- **Buffers**
  - Simple chunks of memory
  - Kernels can access however they like (array, pointers, structs)
  - Kernels can read and write buffers

- **Images**
  - Opaque 2D or 3D formatted data structures
  - Kernels access only via `read_image()` and `write_image()`
  - Each image can be read or written in a kernel, but not both
Image Formats and Samplers

- **Formats**
  - Channel orders: CL_A, CL_RG, CL_RGB, CL_RGBA, etc.
  - Channel data type: CL_UNORM_INT8, CL_FLOAT, etc.
  - `clGetSupportedImageFormats()` returns supported formats

- **Samplers (for reading images)**
  - Filter mode: linear or nearest
  - Addressing: clamp, clamp-to-edge, repeat or none
  - Normalized: true or false

- **Benefit from image access hardware on GPUs**
Allocating Images and Buffers

```c
cl_image_format format;
format.image_channel_data_type = CL_FLOAT;
format.image_channel_order = CL_RGBA;

cl_mem input_image;
input_image = clCreateImage2D(context, CL_MEM_READ_ONLY, &format,
                                image_width, image_height, 0, NULL, &err);

cl_mem output_image;
output_image = clCreateImage2D(context, CL_MEM_WRITE_ONLY, &format,
                                image_width, image_height, 0, NULL, &err);

cl_mem input_buffer;
input_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY,
                               sizeof(cl_float)*4*image_width*image_height, NULL, &err);

cl_mem output_buffer;
output_buffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                                sizeof(cl_float)*4*image_width*image_height, NULL, &err);
```
Explicit commands to access memory object data

- Read from a region in memory object to host memory
  - `clEnqueueReadBuffer(queue, object, blocking, offset, size, *ptr, ...)`
- Write to a region in memory object from host memory
  - `clEnqueueWriteBuffer(queue, object, blocking, offset, size, *ptr, ...)`
- Map a region in memory object to host address space
  - `clEnqueueMapBuffer(queue, object, blocking, flags, offset, size, ...)`
- Copy regions of memory objects
  - `clEnqueueCopyBuffer(queue, srcobj, dstobj, src_offset, dst_offset, ...)`

Operate synchronously (`blocking = CL_TRUE`) or asynchronously
Compilation and Execution of Kernels
Program and Kernel Objects

- Program objects encapsulate ...
  - a program source or binary
  - list of devices and latest successfully built executable for each device
  - a list of kernel objects

- Kernel objects encapsulate ...
  - a specific kernel function in a program - declared with the `kernel` qualifier
  - argument values
  - kernel objects created after the program executable has been built
Executing Code

- Programs build executable code for multiple devices
- Execute the same code on different devices

Kernel Code

code

data

Program

Compile for GPU

Compile for CPU

GPU code

CPU code

* Other names and brands may be claimed as the property of others.  Copyright © 2009, Intel Corporation.
Executing Kernels

1. Set the kernel arguments
2. Enqueue the kernel

```c
size_t global[3] = {image_width, image_height, 0};
err = clEnqueueNDRangeKernel(queue, kernel, 2, NULL, global, NULL, 0, NULL, NULL);
```

- **Note:** Your kernel is executed *asynchronously*
  - Nothing may happen — you have just enqueued your kernel
  - Use a blocking read `clEnqueueRead*(... CL_TRUE ...)`
  - Use events to track the execution status
Synchronization Between Commands

- Each *individual* queue can execute in order or out of order
  - For in-order queue, all commands execute in order
  - Behaves as expected (as long as you’re enqueuing from one thread)

- You must *explicitly synchronize between queues*
  - Multiple devices each have their own queue
  - Use events to synchronize

- Events
  - Commands *return events* and *obey waitlists*
  - `clEnqueue*(..., num_events_in_waitlist, *event_waitlist, *event_out)"
Synchronization: One Device/Queue

- Example: Kernel 2 uses the results of Kernel 1

Kernel 2 waits in the queue until Kernel 1 is finished
Synchronization: Two Devices/Queues

- Explicit dependency: Kernel 1 must finish before Kernel 2 starts
Synchronization: Two Devices/Queues

Kernel 2 starts before the results from Kernel 1 are ready

Kernel 2 waits for an event from Kernel 1 and does not start until the results are ready
Using Events on the Host

• `clWaitForEvents(num_events, *event_list)`
  – Blocks until events are complete
• `clEnqueueMarker(queue, *event)`
  – Returns an event for a marker that moves through the queue
• `clEnqueueWaitForEvents(queue, num_events, *event_list)`
  – Inserts a “WaitForEvents” into the queue
• `clGetEventInfo()`
  – Command type and status
    `CL_QUEUED, CL_SUBMITTED, CL_RUNNING, CL_COMPLETE`, or error code
• `clGetEventProfilingInfo()`
  – Command queue, submit, start, and end times
Programming Advice

Performance and Debugging
Performance: Overhead

- Compiling programs can be expensive
  - Reuse programs or pre-compiled binaries
- Moving data to/from some devices may be expensive
  - e.g. Discrete GPU over PCIe
  - Keep data on device
- Starting a kernel can be expensive
  - Do a lot of work for each execution
- Events can be expensive on some devices
  - Only use events where needed
Performance: Kernels and Memory

- Large global work sizes help hide memory latency and overheads
  - 1000+ work-items preferred
- Trade-off math precision and performance with half_ and native_
- Divergent execution can be bad on some devices
  - All work-items in a work-group should take very similar control flow paths
- Handle data reuse through local memory when available
- Access memory sequentially across work-items
  - Enables hardware memory coalescing
  - Dramatic bandwidth improvements
Debugging

- Start on the CPU
- Be very careful about reading/writing out of bounds on the GPU
  - Use explicit address checks around reads and writes if a kernel is crashing to locate problems
- Play nicely with other apps
  - GPUs are not preemptively scheduled
- Use extra output buffers/images to record intermediate values
- Set a context call-back function to report API errors
OpenCL C Language

- Derived from ISO C99
  - No standard C99 headers, function pointers, recursion, variable length arrays, and bit fields
- Additions to the language for parallelism
  - Work-items and workgroups
  - Vector types
  - Synchronization
- Address space qualifiers
- Optimized image access
- Built-in functions
Kernel

- A data-parallel function executed for each work-item

```c
kernel void square(global float* input, global float* output) {
    int i = get_global_id(0);
    output[i] = input[i] * input[i];
}
```

<table>
<thead>
<tr>
<th>Input</th>
<th>6 1 1 0 9 2 4 1 1 9 7 6 1 2 2 1 9 8 4 1 9 2 0 0 7 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>36 1 1 0 81 4 16 1 1 81 49 36 1 4 4 1 81 64 16 1 81 4 0 0 49 64</td>
</tr>
</tbody>
</table>
Work-Items and Workgroup Functions

get_work_dim = 1

get_global_size = 26

get_num_groups = 2

get_group_id = 0

get_local_size = 13

get_local_id = 8

get_global_id = 21
Data Types

- **Scalar data types**
  - char, uchar, short, ushort, int, uint, long, ulong
  - bool, intptr_t, ptrdiff_t, size_t, uintptr_t, void, half (storage)

- **Image types**
  - image2d_t, image3d_t, sampler_t

- **Vector data types**
Vector Types

- Portable
- Vector length of 2, 4, 8, and 16
- char2, ushort4, int8, float16, double2, ...
- Endian safe
- Aligned at vector length
- Vector operations and built-in functions
Vector Operations

- **Vector literal**
  
  ```c
  int4 vi0 = (int4) -7;
  int4 vi1 = (int4)(0, 1, 2, 3);
  ```

- **Vector components**
  
  ```c
  vi0.lo = vi1.hi;
  int8 v8 = (int8)(vi0, vi1.s01, vi1.odd);
  ```

- **Vector ops**
  
  ```c
  vi0 += vi1;
  vi0 = abs(vi0);
  ```
Address Spaces

- **Kernel** pointer arguments must use *global*, *local* or *constant*

  ```c
  kernel void distance(global float8* stars, local float8* local_stars)
  kernel void sum(private int* p)  // Illegal because is uses private
  ```

- Default address space for arguments and local variables is *private*

  ```c
  kernel void smooth(global float* io) {
      float temp;
      ...
  }
  ```

- *image2d_t* and *image3d_t* are always in *global* address space

  ```c
  kernel void average(read_only global image_t in, write_only image2d_t out)
  ```
Address Spaces

- Program (global) variables must be in constant address space

```c
constant float bigG = 6.67428E-11;
global float time; // Illegal non constant
kernel void force(global float4 mass) { time = 1.7643E18f; }
```

- Casting between different address spaces is undefined

```c
kernel void calcEMF(global float4* particles) {
    global float* particle_ptr = (global float*) particles;
    float* private_ptr = (float*) particles; // Undefined behavior -
    float particle = * private_ptr; // different address
}
```
Conversions

- Scalar and pointer conversions follow C99 rules
- No implicit conversions for vector types
  
  ```
  float4 f4 = int4_vec; // Illegal implicit conversion
  ```
- No casts for vector types (different semantics for vectors)
  
  ```
  float4 f4 = (float4) int4_vec; // Illegal cast
  ```
- Casts have other problems
  
  ```
  float x;
  int i = (int)(x + 0.5f); // Round float to nearest integer
  ```
  - Wrong for:
    - 0.5f - 1 ulp (rounds up not down)
    - negative numbers (wrong answer)
- There is hardware to do it on many devices
Conversions

- Explicit conversions: `convert_destType<_sat><_roundingMode>`
  - Scalar and vector types
  - No ambiguity

```c
uchar4 c4 = convert_uchar4_sat_rte(f4);
```

<table>
<thead>
<tr>
<th>f4</th>
<th>-5.0f</th>
<th>254.5f</th>
<th>254.6f</th>
<th>1.2E9f</th>
</tr>
</thead>
<tbody>
<tr>
<td>c4</td>
<td>0</td>
<td>254</td>
<td>255</td>
<td>255</td>
</tr>
</tbody>
</table>

- Saturate to 0
- Round down to nearest even
- Round up to nearest value
- Saturated to 255
Reinterpret Data: `as_typen`

- Reinterpret the bits to another type
- Types must be the same size
- OpenCL provides a `select` built-in

```c
// f[i] = f[i] < g[i] ? f[i] : 0.0f
float4 f, g;
int4 is_less = f < g;
f = as_float4(as_int4(f) & is_less);
```

<table>
<thead>
<tr>
<th></th>
<th>f</th>
<th>g</th>
<th>is_less</th>
<th>as_int</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>-5.0f</td>
<td>254.5f</td>
<td>254.6f</td>
<td>1.2E9f</td>
</tr>
<tr>
<td>g</td>
<td>254.6f</td>
<td>254.6f</td>
<td>254.6f</td>
<td>254.6f</td>
</tr>
<tr>
<td>is_less</td>
<td>ffffffff</td>
<td>ffffffff</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>as_int</td>
<td>c0a00000</td>
<td>42fe0000</td>
<td>437e8000</td>
<td>4e8f0d18</td>
</tr>
<tr>
<td>&amp;</td>
<td>c0a00000</td>
<td>42fe0000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>f</td>
<td>-5.0f</td>
<td>254.5f</td>
<td>0.0f</td>
<td>0.0f</td>
</tr>
</tbody>
</table>
Built-in Math Functions

- IEEE 754 compatible rounding behavior for single precision floating-point
- IEEE 754 compliant behavior for double precision floating-point
- Defines maximum error of math functions as ULP values
- Handle ambiguous C99 library edge cases
- Commonly used single precision math functions come in three flavors
  - eg. log(x)
    - Full precision <= 3 ulps
    - Half precision/faster. half_log—minimum 11 bits of accuracy, <= 8192 ulps
    - Native precision/fastest. native_log: accuracy is implementation defined
  - Choose between accuracy and performance
Built-in Work-group Functions

- **Synchronization**
  - **Barrier**

- **Work-group functions**
  - Encountered by all work-items in the work-group
  - With the same argument values

```c
kernel read(global int* g, local int* shared) {
    if (get_global_id(0) < 5)
        barrier(CLK_GLOBAL_MEM_FENCE);
    else
        k = array[0];
}
```

Illegal since not all work-items encounter barrier
Built-in Work-group Functions

- async_work_group_copy
  - Copy from global to local or local to global memory
  - Use DMA engine or do a memcpy across work-items in work-group
  - Returns an event object
- wait_group_events
  - wait for events that identify async_work_group_copy operations to complete
Built-in Functions

- **Integer functions**
  - abs, abs_diff, add_sat, hadd, rhadd, clz, mad_hi, mad_sat, max, min, mul_hi, rotate, sub_sat, upsample

- **Image functions**
  - read_image[f | i | ui]
  - write_image[f | i | ui]
  - get_image_[width | height | depth]

- **Common, Geometric and Relational Functions**

- **Vector Data Load and Store Functions**
  - eg. vload_half, vstore_half, vload_halfn, vstore_halfn, ...
Built-in Functions

Math Functions

gentype acos (gentype x)
gentype acosh (gentype x)
gentype acoshf (gentype x)
gentype acosinf (gentype x)
gentype acosip (gentype x)
gentype asin (gentype x)
gentype asinf (gentype x)
gentype asinip (gentype x)
gentype atan (gentype x)
gentype atanf (gentype x)
gentype atan2 (gentype x, gentype y)
gentype atan2f (gentype x, gentype y)
gentype atan2pi (gentype x, gentype y)
gentype cbrt (gentype x)
gentype ceil (gentype x)
gentype copy (gentype x)
gentype copyf (gentype x)
gentype copyi (gentype x)
gentype copyi16 (gentype x)
gentype copyi32 (gentype x)
gentype copyi64 (gentype x)
gentype copyq (gentype x)
gentype copyq (gentype x, gentype y)
gentype copyq (gentype x, float y)
gentype copyq (gentype x, gentype y, float y)
gentype copysign (gentype x, gentype y)
gentype erf (gentype x)
gentype erfc (gentype x)
gentype exp (gentype x)
gentype exp10 (gentype x)
gentype exp2 (gentype x)
gentype exp2 (gentype x, gentype y)
gentype expm1 (gentype x)
gentype expm1 (gentype x, gentype y)
gentype expm1 (gentype x, float y)
gentype expm1 (gentype x, gentype y, float y)
gentype expm1 (gentype x, float y, float y)
gentype expm1 (gentype x, float y, float y, float y)

Relational Ops

int isgreater (float x, float y)
int isgreater (float x, float y, float z)
gentype isgreater (gentype x, gentype y)
gentype isgreater (gentype x, gentype y, gentype z)
gentype isgreater (gentype x, float y)
gentype isgreater (float y, float x)

Vector Loads/Store Functions

gentype vloadn (size_t offset, const __local gentype *p)
gentype vloadn (size_t offset, const global gentype *p)

Integer Ops

gentype bitselect (gentype a, gentype b, gentype c)
gentype bitselect (gentype a, gentype b, float c)
gentype bitselect (float a, float b, float c)
gentype bitselect (float a, float b, gentype c)
gentype bitselect (gentype a, float b, float c)
gentype bitselect (float a, gentype b, float c)
gentype bitselect (float a, float b, gentype c)

Other names and brands may be claimed as the property of others. * Copyright © 2009, Intel Corporation.
Extensions

- Double Precision
- Atomic functions to global and local memory
  - add, sub, xchg, inc, dec, cmp_xchg, min, max, and, or, xor
  - 32-bit/64-bit integers
- Select rounding mode for a group of instructions at compile time
  - For instructions that operate on floating-point or produce floating-point values
  - #pragma opencl_select_rounding_mode rounding_mode
  - All 4 rounding modes supported
- Extension: Check clGetDeviceInfo with CL_DEVICE_EXTENSIONS
Summary

- Portable and high-performance framework
  - Ideal for computationally intensive algorithms
  - Access to all compute resources
  - Portable across different devices
  - Well-defined computation/memory model

- Efficient parallel programming language
  - C99 with extensions for task and data parallelism
  - Rich set of built-in functions

- Defines hardware and numerical precision requirements

- Open standard for heterogeneous parallel computing