OpenCL?

OpenCL is an open standard: [http://khronos.org/OpenCL](http://khronos.org/OpenCL)

- Write a **kernel**: a single portable OpenCL C function that can be run on heterogeneous CPUs, GPUs, and other computing devices.
- Setup an OpenCL environment and manage kernels and data from a Host Program
- Your will be able to use all resources in the heterogeneous platform, in parallel!
OpenCL API

To set-up and manage OpenCL computation, your application has access to the OpenCL API. It is available through C and C++ interfaces:

C:

#include <CL/cl.h>

C++:

#include <CL/cl.hpp>

To use standard exceptions using the C++ API, you have to define:

__CL_ENABLE_EXCEPTIONS
OpenCL Objects

- **cl_platform_id** – Identifier of your OpenCL SDK, you can have more than one installed on your system.
- **cl_device_id** – Identifier of your computational devices (CPU, GPU...).
- **cl_context** – OpenCL context that can be associated to one or more devices.
- **cl_command_queue** – Command queue associated to a unique context and device, that can be in-order (and serial) or out-of-order (commands may be executed concurrently). You can have more than one command queue per-device.
- **cl_program** – compiled OpenCL C source code
- **cl_kernel** – Entry point in a OpenCL program
- **cl_mem** – OpenCL memory object
- **cl_event** – OpenCL event
OpenCL Execution Model

The OpenCL host application must:

1. define the platform, its devices, context, and command queues.
2. build the OpenCL program (portable set of kernels and functions).
3. get kernel(s) from the program.
4. setup memory objects (buffers or 2D-3D images).
5. attach arguments to kernel(s).
6. enqueue commands (kernel execution, memory operations...) and bind events.
OpenCL Execution Model

- CPU
- GPU
- ...

Context

OpenCL C Program with kernel(s)

Memory
- private
- local
- global
- constant

Buffers
- Images

Command Queues
- devices[1]
- devices[0]

Compile
- OpenCL

Create and move data
- Host Application using

Enqueue for execution
- OpenCL APIs
Queries

In C, often used in two-steps:
1 – write into dataSize the amount of data the next query will write:
   
   ```c
   cl_int clGetXX(..., 0, NULL, &dataSize);
   ```

2 – allocate memory for data, then ask for a maximum of dataSize data to be written into dataPtr:
   
   ```c
   cl_int clGetXX(..., dataSize, dataPtr, NULL);
   ```

You can do it in one step when you already know the maximum size you want to be retrieved, and memory is already allocated:

   ```c
   cl_int clGetXX(..., dataSizeAsked, dataPtr, &dataSize);
   ```

These functions always return a `cl_int` error code.

Using the C++ API, one step is always enough:

   ```cpp
   XXReturnType cl::Context::getInfo<XX>(...);
   cl::Platform::get(...);
   ```
Queries in action - C++

Get the first available platform identifier:
```cpp
vector<cl::Platform> platforms;
cl::Platform::get(&platforms);
```

Get its compute devices:
```cpp
vector<cl::Device> devices;
platforms[0].getDevices(CL_DEVICE_TYPE_ALL, &devices);
```

Get device extensions:
```cpp
string deviceExts = devices[0].getInfo<CL_DEVICE_EXTENSIONS>();
```
Queries in action – C

Get the first available platform identifier:
```c
cl_platform_id platformId = NULL;
cl_int errorNum = clGetPlatformIDs(1, &platformId, NULL);
```

Get its compute devices:
```c
cl_uint numDevices=0;
cl_int errorNum = clGetDeviceIDs(platformId, CL_DEVICE_TYPE_ALL, 0, NULL, &numDevices); // get their count
cl_device_id *devices = (cl_device_id*)malloc(sizeof(cl_device_id)*numDevices); // allocate memory
errorNum = clGetDeviceIDs(platformId, CL_DEVICE_TYPE_ALL, numDevices, devices, NULL); // get them
```

Get device extensions:
```c
size_t bufSize;
cl_int errorNum = clGetDeviceInfo(devices[0], CL_DEVICE_EXTENSIONS, 0, NULL, &bufSize);
char *deviceExts=(char*)malloc(bufSize);
errorNum = clGetDeviceInfo(devices[0] (), CL_DEVICE_EXTENSIONS, bufSize, deviceExts, NULL);
```
Objects Creation

**C:**
```
cl_XX clCreateXX(..., cl_int *errorOut);
```

**C++:**
```
cl::XX(..., cl_int *errorOut);
```

Functions that return an object always take a `cl_int` pointer in last argument to return error whereas the other functions directly return a `cl_int` error.

In C++, object destructor properly handles its destruction.

In C, you have to call explicitly `clReleaseXX(...)`.  

When using the C++ API with `__CL_ENABLE_EXCEPTIONS` set, you can directly catch `cl::Error` exceptions. In that case you don't need to specify any `cl_int`* last argument to get the error code – that argument is set to NULL by default.
Create a context:

```c
cl_context context = clCreateContext(props, numDevices, devices, NULL, NULL, &errorCode);
```

Create a command queue:

```c
cl_command_queue queue = clCreateCommandQueue(context, devices[0], CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &errorCode);
```

Create program from a C-string:

```c
cl_program program = clCreateProgramWithSource(context, 1, (const char**) &programSourceString, NULL, &errorCode);
```

Create kernel from a program:

```c
cl_kernel kernel = clCreateKernel(program, "kernelName", &errorCode);
```
Create a context:
   cl::Context context(devices);

Create a command queue:
   cl::CommandQueue queue(context, devices[0],
   CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE);

Create program from a C-string:
   cl::Program::Sources sources(1, make_pair(programCString, 0));
   cl::Program program(context, sources);

Create kernel from a program:
   cl::Kernel kernel(program, "kernelName");
Memory Objects Creation

C:          cl_mem clCreateXX(context, properties, size, hostPtr, errorNum)

C++:        cl::Memory::XX(context, properties, size, hostPtr);

XX can be Buffer, Image2D or Image3D.

Properties is a bit field that can define:
  CL_MEM_READ_ONLY, CL_MEM_WRITE_ONLY,
  CL_MEM_READ_WRITE (not available for images)

And:
  CL_MEM_COPY_HOST_PTR: data will be allocated on devices
    associated to the context, then copied from host in a blocking manner.
  CL_MEM_ALLOC_HOST_PTR: only allocation will be done on
    device, you may need to use clEnqueueWriteBuffer()
  CL_MEM_USE_HOST_PTR: memory will always be mapped to
    the same original pointer address.
Memory Objects Creation - Example

C++:
```
cl::Buffer buffer_result(context, CL_MEM_WRITE_ONLY, sizeof(float)*resultArraySize);
```

C:
```
cl_mem buffer_result = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float)*resultArraySize, NULL, &errorNum);
```
Enqueueing commands

C:       cl_int clEnqueueXX(cl_queue queue, ... , cl_uint eventsInCount, const cl_event* eventsIn, cl_event* eventOut);

C++:     cl_int cl::CommandQueue::enqueueXX(..., vector<Event>* eventsIn, Event* eventOut);

An enqueuing command always takes in last arguments:
- the number of events to wait on (only in C)
- the associated cl_events array
- the cl_event pointer to be managed by that command.
Enqueuing commands in action – C

Enqueuing a NDRange Kernel execution:

```c
errorNum = clSetKernelArg(kernel, 0, sizeof(cl_mem), &buffer_arg1); // set arg1
cl_event helloKernelEvt;
errorNum = clEnqueueNDRangeKernel(queue, kernel, workDim, NULL, globalWorkSize, NULL, /*numEventsToWaitOn=*/1, eventsToWaitOn, &kernelEvt);
```

Enqueuing a Task Kernel execution:

```c
cEnqueueTask(queue,kernel, /*numEventsToWaitOn=*/1, &eventsToWaitOn, &eventOut);
```

Enqueuing a Map request:

```c
float *result = (float*) clEnqueueMapBuffer(queue, buffer_result, /*blocking=*/CL_TRUE, CL_MAP_READ, /*offset=*/0, numBytes, /*numEventsToWaitOn=*/1, eventsToWaitOn, &eventOut, &errorCode);
```
Enqueuing commands in action – C++

Enqueuing a NDRange Kernel execution:

```cpp
cl::KernelFunctor kernelCLFunc = kernel.bind(queue, /*globalRange=*/cl::NDRange(nbWorkItems), /*localRange=auto*/cl::NullRange); // create a functor bound to the OpenCL kernel
cl::Event kernelEvt = kernelCLFunc(arg1); // clEnqueueNDRangeKernel
```

Enqueuing a Task Kernel execution:

```cpp
cl::Event eventOut;
queue.enqueueTask(kernel, &eventsToWaitOn, &eventOut);
```

Enqueuing a Map request:

```cpp
float *result = (float*)queue.enqueueMapBuffer(buffer_result, CL_TRUE, CL_MAP_READ, /*offset=*/0, numBytesToRead, eventsToWaitOn, &eventOut);
```
Handling events

An event can have 4 different status:

- CL_QUEUED
- CL_SUBMITTED
- CL_RUNNING
- CL_COMPLETE

You can create user events using:

C    : cl_event event = clCreateUserEvent(context, &errorCode);

C++ : cl::UserEvent event(context);

Then use clSetUserEventStatus(CL_XX) / event.setStatus(CL_XX) to modify its status.

Don’t mix up user events with events created by enqueue commands!
OpenCL Program

Classic serial version:

```c
void add_vecs(const float *a, const float *b, float *result, size_t length){
    for(size_t i=0;i<length;i++)
        result[i] = a[i] + b[i];
}
```

OpenCL Program:

```c
kernel void add_vecs_kernel(global const float *a, global const float *b, global float *result){
    size_t gid = get_global_id(0);
    result[gid] = a[gid] + b[gid];
}
```

A kernel is an entry function in an OpenCL C program. It is designed to work on a unique point in the problem domain and should return void. Its execution will be managed from the host application.
Data Parallelism

Data parallelism is achieved using `clEnqueueNDRangeKernel()`. Kernels are executed concurrently on work-items. They have private memory, visible only for themselves, and local memory, shared among a work-group.

Problem Domain can be a 1D, 2D or 3D NDRange (N-dimensional range).

`local` memory state is not guaranteed to be consistent across the work-items: you can use `barrier()`, `mem_fence()`, `read_mem_fence()` and `write_mem_fence()` within kernels to get some synchronization.

All work-items in a work-group must encounter the same barriers. That synchronization works only across work-items sharing the same work-group.
You can query the cursor position from your kernel, using `get_XX(uint dimIndex)`

Here `get_work_dim()` will return 2

- `get_global_id(0)` -> 6
- `get_global_id(1)` -> 0

- `get_group_id(0)` -> 1
- `get_group_id(1)` -> 1

- `get_local_id(0)` -> 2

- `get_global_size(0)` -> 8
- `get_global_size(1)` -> 8

- `get_local_size(0)` -> 4
- `get_local_size(1)` -> 4
Task Parallelism

There are two ways you can achieve task parallelism:

- **Using an out of order queue:**
  Commands are executed as soon as the compute device is ready, and may run concurrently.

- **Using several queues:**
  These queues will run concurrently.

You can synchronize commands using **events**.

Before enqueuing a command that will wait for an event coming from another queue, you must call `clFlush()` on that other queue.

`clEnqueueTask()` is exactly like a `clEnqueueNDRangeKernel()` with local and global dimensions set to 1 and only 1 work-item.
OpenCL Program

Usually, an OpenCL program is written directly in a C-string in the host application code, or separately in a .cl text file whose content will be converted in a C-string by the host application.

It can also be cached in a device dependant binary file and loaded from it, but that topic is not covered in that presentation.
OpenCL C is derived from C99.
Some features are added:

- vector types (int8, float4...)
- synchronization
- work-item and workgroups interfaces (get_global_id()...)
- address space qualifiers (kernel, global...)
- built-in math and image manipulation functions
OpenCL C - 2/2

But some others are lacking:

- standard C99 headers (string.h, stdio.h...)
- function pointers
- recursion
- variable length arrays
- bit fields
- variadic templates

and OpenCL Image types can only be types of function arguments
Specific Kernel Restrictions

There are restrictions on OpenCL kernel arguments:

- Pointers must be global, constant or local.
- Pointers to pointers are not supported.
- They cannot be: bool, half, size_t, ptrdiff_t, intptr_t, uintptr_t, even_t.
OpenCL memory objects (cl_mem)

Buffer:
- classic 1D array of bytes

Images:
- 2D or 3D
- Opaque type handled by the device
- Writable or readable by a kernel, not both.
- Need a cl_sampler object to be read (object used to define sampling and bounds behaviors)
## Scalar-only data types

<table>
<thead>
<tr>
<th>OpenCL C</th>
<th>OpenCL API</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>true (1) or false (0)</td>
</tr>
<tr>
<td>half</td>
<td>cl_half</td>
</tr>
<tr>
<td>size_t</td>
<td></td>
</tr>
<tr>
<td>ptrdiff_t</td>
<td></td>
</tr>
<tr>
<td>intptr_t</td>
<td></td>
</tr>
<tr>
<td>uintptr_t</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td></td>
</tr>
</tbody>
</table>
## Vector/Scalar Data Types

These types are vectors when \( n \) equals to 2, 3, 4, 8 or 16, and scalars without \( n \) :

<table>
<thead>
<tr>
<th>OpenCL C</th>
<th>OpenCL API</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>char( n )</td>
<td>cl_charn</td>
<td>8-bit signed</td>
</tr>
<tr>
<td>uchar( n )</td>
<td>cl_ucharn</td>
<td>8-bit unsigned</td>
</tr>
<tr>
<td>short( n )</td>
<td>cl_shortn</td>
<td>16-bit signed</td>
</tr>
<tr>
<td>ushort( n )</td>
<td>cl_ushortn</td>
<td>16-bit unsigned</td>
</tr>
<tr>
<td>int( n )</td>
<td>cl_intn</td>
<td>32-bit signed</td>
</tr>
<tr>
<td>uint( n )</td>
<td>cl_uintn</td>
<td>32-bit unsigned</td>
</tr>
<tr>
<td>long( n )</td>
<td>cl_longn</td>
<td>64-bit signed</td>
</tr>
<tr>
<td>ulong( n )</td>
<td>cl_ulongn</td>
<td>64-bit unsigned</td>
</tr>
<tr>
<td>half( n )</td>
<td>cl_halfn</td>
<td>16-bit half (if cl_khr_fp16 on)</td>
</tr>
<tr>
<td>float( n )</td>
<td>cl_floatn</td>
<td>32-bit float</td>
</tr>
<tr>
<td>double( n )</td>
<td>cl_doublen</td>
<td>64-bit double (if cl_khr_fp64 on)</td>
</tr>
</tbody>
</table>
Accessing vector components

Typical components can be accessed using two addressing modes:

- `vec.x, y, z, or w`
- `vec.s0,1,2...a/A...f/F`
- `vec.s[0,1,2...](only from OpenCL API)`

You can even get a vector of these components (only in OpenCL C):

- `vec.yw`
- `vec.s13`
- `vec.s1w`: these addressing modes cannot be mixed

And there are some special suffixes:

- `vec.lo` or `vec.low`: lower half of vec
- `vec.hi` or `vec.high`: upper half of vec
- `vec.odd`: odd components of vec (OpenCL C only)
- `vec.even`: even components of vec (OpenCL C only)
Vector initialization

Using OpenCL API:

```c
cl_float4 vec = {{1.0f,2.0f,3.0f,4.0f}};
```

In OpenCL C:

```c
float4 vec = (float4)(1.0f,2.0f,3.0f,4.0f);
```

You can even use other vectors:

```c
float4 vec = (float4)(1.0f,(float2)(2.0f,3.0f),4.0f);
```
Hello World – C++ 1/3

```cpp
#include <iostream>

#define __CL_ENABLE_EXCEPTIONS // enable use of exceptions by the OpenCL API
#include <CL/cl.hpp>
#include <stdexcept>

using namespace std;

int main(int argc, char **argv) {
    const size_t kArraySize = 50;

    vector<cl::Platform> platforms;
    cl::Platform::get(&platforms); // get available OpenCL platforms (=SDKs)

    vector<cl::Device> devices;
    platforms[0].getDevices(CL_DEVICE_TYPE_ALL, &devices); // get OpenCL devices from first platform

    cl::Context context(devices); // create a context for these devices

    cl::CommandQueue queue(context, devices[0], 
        CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE); // create an out of order queue for the first device
```
... string programSourceString = "kernel void hello_kernel(global const float *a, 
    global const float *b, global float *result){ \n" "    int gid = get_global_id(0); \n" "    result[gid] = a[gid] + b[gid]; \n" "};

cl::Program::Sources sources; 
sources.push_back(make_pair(programSourceString.c_str(), 0)); // don't need to 
specify length as we used a null-terminated string

cl::Program program(context, sources); //create the OpenCL program 
try{
    program.build(devices); // compile it for every devices
}
catch(const cl::Error &e){ // get compilation log in case of failure 
cerr << e.what() << " returned err: " << e.err() << endl;
    for(size_t i=0;i<devices.size();++i)
        cerr << "build log for device " << i << ":\n" << program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(devices[i]) << endl;
}

cl::Kernel kernel(program, "hello_kernel"); //create a OpenCL kernel object
cl::KernelFunctor hello_kernelCLFunc = kernel.bind(queue, 
cl::NDRange(kArraySize), cl::NullRange); //create a functor bound 
to the OpenCL kernel with a local range automatically set by OpenCL
...
OpenCL Hello World – C++ 3/3

...  
// allocate local memory objects  
float* a = new float[kArraySize];  
float* b = new float[kArraySize];  
float* result = new float[kArraySize];  
for (size_t i=0; i<kArraySize; ++i){   
a[i]=i;   
b[i]=i*2;  }

// Create OpenCL memory objects and copy a and b to their respective buffers  
cl::Buffer buffer_a(context, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,  
sizeof(float)*kArraySize, a);  
cl::Buffer buffer_b(context, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,  
sizeof(float)*kArraySize, b);  
cl::Buffer buffer_result(context, CL_MEM_WRITE_ONLY, sizeof(float)*kArraySize);

cl::Event helloKernelEvt = hello_kernelCLFunc(buffer_a, buffer_b,  
buffer_result); //use the functor to call the kernel using our arguments

vector<cl::Event> eventsToWaitOn(1,helloKernelEvt);  
// explicitly waiting for helloKernelEvt is needed only if CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE is set on queue
queue.enqueueReadBuffer(buffer_result,  /*blocking=*/CL_TRUE, 0, kArraySize*sizeof(float), result, &eventsToWaitOn);

for (size_t i=0; i<kArraySize; ++i) cout << result[i] << " ";  
cout << endl;

delete[] a;  
delete[] b;  
delete[] result;

return EXIT_SUCCESS;
OpenCL Hello World – C 1/4

#include <stdio.h>
#include <string.h>
#include <CL/cl.h>

#define ARRAY_SIZE 50

int main(int argc, char ** argv){
    cl_int errorNum=CL_SUCCESS;
    size_t i=0;

    cl_platform_id platformId = NULL;
    errorNum = clGetPlatformIDs(1, &platformId, NULL); //get first available platform ID

    cl_uint numDevices=0;
    errorNum = clGetDeviceIDs(platformId, CL_DEVICE_TYPE_ALL, 0, NULL, &numDevices); //get the number of devices
    cl_device_id *devices = (cl_device_id*)malloc(sizeof(cl_device_id)*numDevices); //allocate memory for their IDs
    errorNum = clGetDeviceIDs(platformId, CL_DEVICE_TYPE_ALL, numDevices, devices, NULL); //get their IDs

    cl_context_properties props[] = {};
    cl_context context = clCreateContext(props, numDevices, devices, NULL, NULL, &errorNum); //create context using the first device

    cl_command_queue queue = clCreateCommandQueue(context, devices[0], CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &errorNum); //create out-of-order queue
    ...

... 

const char *programSourceString = "kernel void hello_kernel(global const float *a, 
global const float *b, global float *result){ \n" 
" int gid = get_global_id(0); \n" 
" result[gid] = a[gid] + b[gid]; \n" 
"}";

cl_program program = clCreateProgramWithSource(context, 1,(const char**) &programSourceString, NULL, &errorNum); //create program - we don't need to 
specify source string lengths as we used null-terminated strings

errorNum = clBuildProgram(program, 0, NULL, NULL, NULL, NULL); //build program 
if(errorNum!=CL_SUCCESS){
 printf("error while compiling program.\n");
 for(i=0;i<numDevices;++i){
   size_t bufferSize=0;
   clGetProgramBuildInfo(program, devices[0], CL_PROGRAM_BUILD_LOG, 0, NULL, &bufferSize); //get size of build log
   char *buffer = (char*)malloc(bufferSize); // allocate memory for the build
   log
   clGetProgramBuildInfo(program,devices[0], CL_PROGRAM_BUILD_LOG, bufferSize, buffer, NULL); //get the build log
   printf("build log for device %lu : \n%s\n", i, buffer);
 }
 clReleaseProgram(program);
}

cl_kernel kernel = clCreateKernel(program, "hello_kernel", &errorNum); // create kernel object
...
OpenCL Hello World – C 3/4

... allocate local memory objects */
float *a = (float*)malloc(sizeof(float)*ARRAY_SIZE);
float *b = (float*)malloc(sizeof(float)*ARRAY_SIZE);
float *result = (float*)malloc(sizeof(float)*ARRAY_SIZE);

for(i=0; i < ARRAY_SIZE; ++i) { a[i] = i; b[i] = i*2; } //fill it with some sample data

/* Create OpenCL memory objects and copy a and b to their respective buffers */
cl_mem buffer_a = clCreateBuffer(context, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
sizeof(float)*ARRAY_SIZE, a, &errorNum);
cl_mem buffer_b = clCreateBuffer(context, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
sizeof(float)*ARRAY_SIZE, b, &errorNum);
cl_mem buffer_result = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
sizeof(float)*ARRAY_SIZE, NULL, &errorNum);

/* Set Kernel Arguments */
errorNum = clSetKernelArg(kernel, 0, sizeof(cl_mem), &buffer_a);
errorNum = clSetKernelArg(kernel, 1, sizeof(cl_mem), &buffer_b);
errorNum = clSetKernelArg(kernel, 2, sizeof(cl_mem), &buffer_result);

/* enqueue kernel */
size_t globalWorkSize[] = {ARRAY_SIZE};
cl_event helloKernelEvt;
errorNum = clEnqueueNDRangeKernel(queue, Kernel, 1, NULL, globalWorkSize,
NULL, 0, NULL, &helloKernelEvt); //with a local range automatically set by OpenCL

...
errorNum = clEnqueueReadBuffer(queue, buffer_result, /*blocking=*/CL_TRUE, 0, ARRAY_SIZE*sizeof(float), result, 1, &helloKernelEvt, NULL); // read output buffer back - event needed only if using an out of order queue

for(i=0; i<ARRAY_SIZE; ++i) printf("%.0f ",result[i]); // output result
printf("\n");

/* clean environment | Rule: one clRelease for each clCreate */
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(queue);
clReleaseContext(context);
free(devices);
free(a);
free(b);
free(result);

return EXIT_SUCCESS;
}
Using local memory

**local** memory is part of memory shared by all work-items within a work-group.

You can dynamically declare local memory from host program, and pass it as kernel arguments using:

C++: `kernel.setArg(cl_int idx, cl::__local(size_t bytes));`

C: `cl_int clSetKernelArg (cl_kernel kernel, cl_uint idx, size_t bytes, NULL);`

If its size is static, you can otherwise declare it directly from a kernel using the **local** qualifier:

local float[256];
Know your devices

OpenCL code is portable in term of functionality, but that's not the case for its performance!

CPUs:
Best for task parallelism
CPU cores can perform well asynchronously from each other

GPUs:
Best for data parallelism
Performance Tips for CPUs

`local` memory has no hardware meaning, so caching through it is not efficient. (the CL DEVICE LOCAL MEM TYPE info from device is CL_GLOBAL)

`clFinish` is more effective than using the `clWaitForEvents` function, because `clWaitForEvents` blocks the underlying thread.

`Map/unMap` is more efficient than Read/Write.

Recommended work-group size for `kernels` is 64-128 (32-64 if there is a barrier instruction). But you can pass a NULL argument when enqueuing your kernel to let the OpenCL implementation decide.
Advanced topics

Some topics are not yet covered in that presentation, you might want to learn them by yourself:

- Program binary reuse
- Device fission
- Profiling
- OpenGL Interoperability
- Direct3D Interoperability
- OpenCL 1.2
Useful links

Khronos specification:
http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

Khronos reference card:

Intel OpenCL SDK Optimization Guide:

OpenCL sample source codes from Intel:
http://software.intel.com/en-us/articles/vcsource-samples/?filter=oclSDK

OpenCL sample source codes from Apple:

Books


Knowledge is power. Knowledge shared is power multiplied.

— Robert Noyce —
Visit www.Intel-Software-Academic-Program.com to download the editable version of this lecture.
License Creative Commons - By 3.0

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.

The Intel code snippets provided in this presentation are licensed to you under the CC-BY 3.0: http://creativecommons.org/licenses/by/3.0/

You are free:
- to Share — to copy, distribute and transmit the work
- to Remix — to adapt the work
- to make commercial use of the work

Under the following conditions:
- Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

With the understanding that:
- Waiver — Any of the above conditions can be waived if you get permission from the copyright holder.
- Public Domain — Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.
- Other Rights — In no way are any of the following rights affected by the license:
  - Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
  - The author's moral rights;
  - Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.
- Notice — For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to the web page http://creativecommons.org/licenses/by/3.0/
Legal Disclaimer

This presentation also contains code from the Khronos Group Inc. which is licensed to you under the following terms:

Khronos Group Copyright (c) 2011 The Khronos Group Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and /or associated documentation files (the "Materials"), to deal in the Materials without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Materials, and to permit persons to whom the Materials are furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Materials.

THE MATERIALS ARE PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE MATERIALS OR THE USE OR OTHER DEALINGS IN THE MATERIALS.

Copyright © 2012 Intel Corporation. All rights reserved

*Other names and brands may be claimed as the property of others.