LEARNING opencl

#opencl

Table of Contents

About

Chapter 1: Getting started with opencl

Remarks

This section provides an overview of what opencl is, and why a developer might want to use it.

It should also mention any large subjects within opencl, and link out to the related topics. Since the Documentation for opencl is new, you may need to create initial versions of those related topics.

Opencl is an api that puts gpus,cpus and some other accelerators(like a pcie-fpga) into good use of C99-like computations but with a very wide concurrency advantage. Once installation and basic implementation is done, only simple changes in a kernel string(or its file) applies an algorithm to N hardware threads automagically.

A developer might want to use it because it will be much easier to optimize for memory space or speed than doing same thing on opengl or direct-x. Also it is royalty-free. Concurrency within a device is implicit so no need for explicit multi-threading for each device. But for multi-device configurations, a cpu-multi-threading is still needed. For example, when a 1000 threaded job is sent to a cpu, thread synchronization is handled by driver. You just tell it how big a workgroup should be(such as 256 each connected with virtual local memory) and where synchronization points are(only when needed).

Using gpu for general purpose operations is nearly always faster than cpu. You can sort things quicker, multiply matrices 10x faster and left join in-memory sql tables in "no" time. Any 200$ desktop-grade gpu will finish quicker in a physics(finite-element-method fluid) workload than any 200$ cpu. Opencl makes it easier and portable. When you're done working in C#, you can easily move to java-opencl implementation using same kernels and C++ project(ofcourse using JNI with extra C++ compiling).

For the graphics part, you are not always have to send buffers between cpu and gpu. You can work purely on gpu using "interop" option in context creation part. With interop, you can prepare geometries at the limit performance of gpu. No pci-e required for any vertex data. Just a "command" is sent through, and work is done only inside of graphics card. This means no cpu-overhead for data. Opencl prepares geometry data, opengl renders it. CPU becomes released. For example, if a single thread of cpu can build a 32x32 verticed sphere in 10000 cycles, then a gpu with opencl can build 20 spheres in 1000 cycles.

Examples

What is OpenCL?

OpenCL is short for **Open C**omputing **L**anguage. OpenCL is a Framework for parallel programming across heterogeneous platforms, called *compute devices*, ranging from CPUs over GPUs to more special platforms like FPGAs. OpenCL provides a standard interface for parallel computing on these compute devices but also inter-device parallelism. It specifies a programming language, based on C99, and minimum requirements of basic functions implemented on OpenCL capable devices. OpenCL furthermore describes an abstract computing and memory model, being as general as possible to make the reuse of code between different platforms straightforward.

Prerequisites

If you have a modern CPU or graphics card (GPU) inside your machine, chances are you have everything ready for first steps in OpenCL. Finding out if your processor is OpenCL capable can be usually done via the manufacturer's homepage, a good first start is the official documentation at

https://www.khronos.org/conformance/adopters/conformant-products#opencl

What is OpenCL?

Open Computing Language (OpenCL) is a framework for writing programs that execute on CPUs, GPUs, and other parallel processors and accelerators.

OpenCL specifies a programming language (based on C) that provides access to named on-chip memory, a model for executing tasks in parallel, and the ability to synchronize those tasks.

C# implementation of OpenCL 1.2: number of platforms for an AMD system in 64-bit windows

OpenCL is low level api so it must be implemented in "C space" first. For that, one needs to download header files from Khronos' site. My hardware is AMD and capable of version 1.2, downloading

```
opencl.h
cl_platform.h
cl.h
cl_ext.h
cl_egl.h
cl_dx9_media_sharing.h
cl_d3d10.h
cl_d3d11.h
cl_gl.h
cl_gl_ext.h
cl.hpp
```

from this page

should be enough for C++ bindings so after adding these files to your project and setting proper binary(and library) file locations(

> $(AMDAPPSDKROOT)\lib\x86_64 for 64-bit amd library (amd app sdk's libraries are preferred)

,

C:\Windows\SysWOW64 for 64-bit opencl.dll (.so file if ICD is of a Linux system)

for example but different for Intel-Nvidia), you can start querying a list of platforms(amd,intel,xilinx,nvidia) after installing proper drivers(such as crimson for amd). Drivers are for running opencl application(using ICD), libraries and header files are for development to be in short.

To query platforms:

```
#define __CL_ENABLE_EXCEPTIONS
#include "stdafx.h"
#include <vector>
#include <CL/cl.hpp>

extern "C"
    {
       // when this class is created, it contains a list of platforms in "platforms" field.
       class OpenClPlatformList
       {
           public:
               std::vector<cl::Platform> platforms;
               int platformNum;
               OpenClPlatformList()
               {
                   platforms= std::vector< cl::Platform>();
                   cl::Platform::get(&platforms);
                   platformNum= platforms.size();
               }
        };


        // this is seen from C# when imported. Creates an object in memory.
        __declspec(dllexport)
            OpenClPlatformList * createPlatformList()
        {
            return new OpenClPlatformList();
        }

        __declspec(dllexport)
            int platformNumber(OpenClPlatformList * hList)
        {
            return hList->platformNum;
        }


        __declspec(dllexport)
            void deletePlatformList(OpenClPlatformList * p)
        {
            if (p != NULL)
                delete p;
            p = NULL;
        }


    }
```

could be built into a dll(such as OCLImplementation.dll) and to use it from C# side,

```
using System;
using System.Collections.Generic;
using System.Runtime.InteropServices;


namespace WrapperCSharp
{
    public class WrapperCSharp
    {
        [DllImport("OCLImplementation", CallingConvention = CallingConvention.Cdecl)]
        private static extern IntPtr createPlatformList();

        [DllImport("OCLImplementation", CallingConvention = CallingConvention.Cdecl)]
        private static extern int platformNumber(IntPtr hList);

        [DllImport("OCLImplementation", CallingConvention = CallingConvention.Cdecl)]
        private static extern void deletePlatformList(IntPtr hList);
    }
}
```

ofcourse the dll must be seen by the C# project, simply putting it near executable of project solves it.

Now, if sample computer has at least one opencl-capable platform,

```
IntPtr platformList = createPlatformList(); // just an address in C-space
int totalPlatforms = platformNumber(platformList); // AMD+NVIDIA systems should have "2"
deletePlatformList(platformList); //
```

totalPlatforms variable must have at least "1" value. Then you can use platforms variable in C-space using additional functions to iterate through all platforms to query all devices such as CPU,GPU and special purpose accelerators such as phi or some fpga.

One does not simply write all these C++ to C# wrappers for time-critical projects. There are many wrappers written for C#, Java and other languages. For java, there is "Aparapi" that is the "java bytecode to opencl-c" converter api that takes what you write purely in java to a gpu-parallel version on the fly so it is somewhat portable.

OpenCL and C#

For C# there exist many wrappers that offer an interface to communicate with OpenCL.

- OpenCL.NET: This is one of the most low level wrappers out there. It offers a complete implementation of the OpenCL API for C# without adding any abstraction at all. So C\C++ examples are easily ported for this library. The only project page is currently on codeplex, which shuts down on 15.12.2017 but the package is available on NuGet

https://openclnet.codeplex.com/

- NOpenCL: This library offers an abstract interface between C# and OpenCL.

The short-term goal is providing an easy-to-use abstract layer which provides access to the full capability of OpenCL without sacrificing performance.

https://github.com/tunnelvisionlabs/NOpenCL

- Cloo:

  Cloo is an open source, easy to use, managed library which enables .NET/Mono applications to take full advantage of the OpenCL framework.

https://sourceforge.net/projects/cloo/

Read Getting started with opencl online: https://riptutorial.com/opencl/topic/3694/getting-started-with-opencl

Chapter 2: Atomic Operations

Syntax

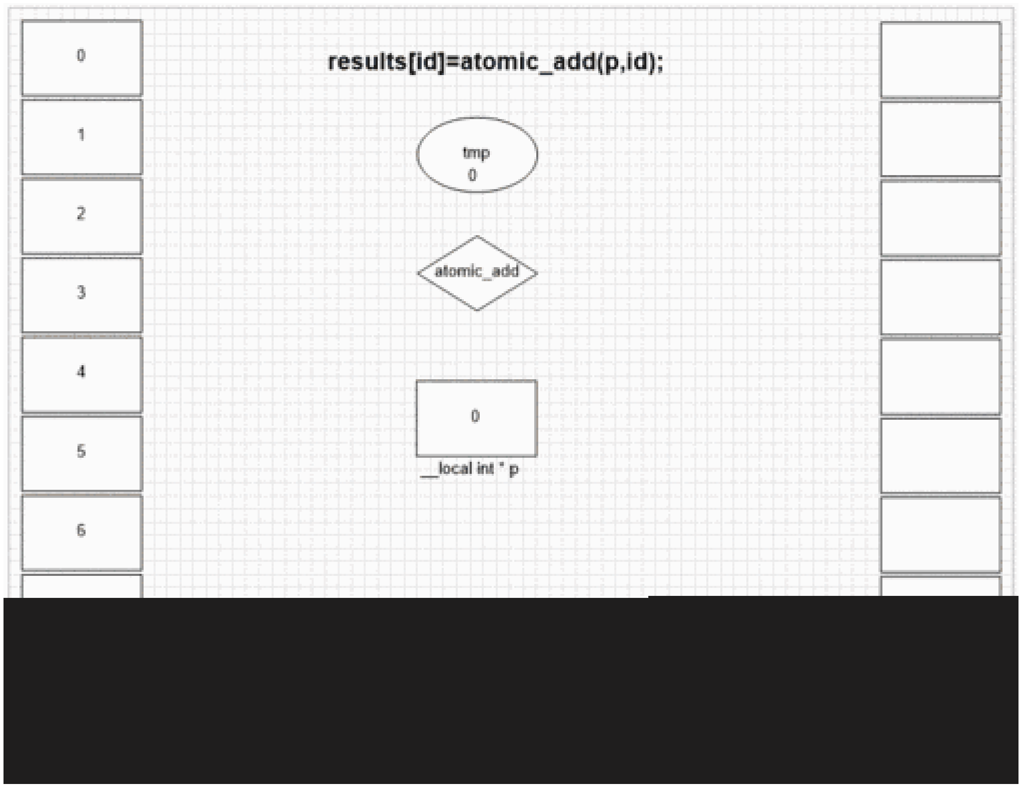
- int atomic_add (volatile __global int *p , int val)
- unsigned int atomic_add (volatile __global unsigned int *p , unsigned int val)
- int atomic_add (volatile __local int *p , int val)
- unsigned int atomic_add (volatile __local unsigned int *p ,unsigned int val)

Parameters

p	val
pointer to cell	added to cell

Remarks

Performance depends on atomic operations number and memory space. Doing serial work almost always slows kernel execution because of gpu being a SIMD array and each unit in an array waits other units if they don't do same type of work.



Examples

Atomic Add Function

```
int fakeMalloc(__local int * addrCounter,int size)
{
    // lock addrCounter
    // adds size to addrCounter's pointed cell value
    // unlock
    // return old value of addrCounter's pointed cell

    // serial between all threads visiting -> slow
    return atomic_add(addrCounter,size);
}

__kernel void vecAdd(__global float* results )
{
   int id = get_global_id(0);
   int lid=get_local_id(0);
   __local float stack[1024];
   __local int ctr;
   if(lid==0)
      ctr=0;
   barrier(CLK_LOCAL_MEM_FENCE);
   stack[lid]=0.0f;                     // parallel operation
   barrier(CLK_LOCAL_MEM_FENCE);
   int fakePointer=fakeMalloc(&ctr,1); // serial operation
   barrier(CLK_LOCAL_MEM_FENCE);
   stack[fakePointer]=lid;              // parallel operation
   barrier(CLK_GLOBAL_MEM_FENCE);
   results[id]=stack[lid];
}
```

Output of first elements:

sometimes

192 193 194 195 196 197 198

sometimes

0 1 2 3 4 5 6

sometimes

128 129 130 131 132 133 134

for a setting with local range=256.

Whatever thread visits fakeMalloc first, it puts its own local thread id in first result cell. Internal SIMD and wavefront structure of the example gpu lets neighboring 64 threads to put their values to neighboring result cells. Other devices may put values more randomly or totally in order depending on the opencl implementation of that devices.

Read Atomic Operations online: https://riptutorial.com/opencl/topic/6758/atomic-operations

Chapter 3: Host memory interaction

Introduction

This topic highlights different ways to put data somewhere where your device can access it.

Examples

Reading an array

To read an array from the device back to the host, one calls

```
clEnqueueReadBuffer($queue, $memobj, $blocking, $offset, $size, $target, 0, null, null);
```

The $queue is the CommandQueue which was used to allocate the memory on the device. The $memobj contains the address to the device memory, $offset and $size further define from where and how much data is copied. The $target is a pointer to the host memory where the data will be stored in. The $target needs to be allocated and have an appropriate size.

Reading a Texture

Reading an image is almost like reading an array. The only difference beeing that the size and offset need to be three-dimensional.

```
clEnqueueReadImage($queue, $memobj, $blocking, $offset, $size, $stride, $slice_pitch, $target,
0, null, null);
```

The $stride defines how many bytes a row has. Normally this is just width * (bytes per pixel), but someone might want to change that to align the data with the memory banks. The same goes for $slice_pitch, only that this value is for the third dimension.

Writing a 2D Texture

To copy a texture to the device there are two steps necessary

1. Allocate the memory on the device
2. Copy the image to the device

```
  _mem = clCreateImage2D($context, $mem_flags, $image_format, $width, $height, $stride,
$source, &err);
```

The $mem_flags define how the memory is allocated. It can be either read only, write only or both. Additionally you can define where and how the memory is allocated. $width, $height and $stride are pretty self explanatory.

If your mem_flags copy the data, you're done. If you want to do that manually at a later point, you'll need to call another function when you are ready.

```
err = clEnqueueWriteImage($queue, _mem, $blocking, $offset, $size, $stride, $slice_pitch,
$source, 0, null, null);
```

The $offset and $size define the image region which you want to copy to the target memory. The $stride defines how many bytes a row has. Normally this is just width * (bytes per pixel), but someone might want to change that to align the data with the memory banks. The same goes for $slice_pitch, only that this value is for the third dimension. Both $stride and $slice_pitch have to match your input data.

Memory flags

When allocating Memory you have the option to choose between different modes:

- Read only memory
- Write only memory
- Read/Write memory

Read-only memory is allocated in the __constant memory region, while the other two are allocated in the normal __global region.

In addition to the accessibility you can define where your memory is allocated.

- Not specified: Your memory is allocated on the device memory as you would expect. The $source pointer can be set to null.
- CL_MEM_USE_HOST_PTR: This tells the device that the data is in the system RAM and should not be moved. Instead the data is manipulated directly in the ram.
- CL_MEM_COPY_HOST_PTR: Tells the device to copy all values at the given address to device memory or, using CL_MEM_ALLOC_HOST_PTR, to a seperate memory region in system ram.
- CL_MEM_ALLOC_HOST_PTR: Tells the device to allocate space at the system ram. If used as the only parameter, the $source pointer can be set to null.

Speed-wise, access to device global memory is the fastest one. But you also need to call it twice to copy data. Using the Host-pointer is the slowest one, while Alloc_host_ptr offers a higher speed.

When using use_host_ptr, the device does exactly that: It uses your data in the system ram, which of course is paged by the os. So every memory call has to go through the cpu to handle potential pagefaults. When the data is available, the cpu copies it into pinned memory and passes it to the DMA controller using precious cpu clock cycles. On the contrary, alloc_host_ptr allocates pinned memory in the system ram. This memory is placed outside of the pageswap mechanism and therefore has a guaranteed availability. Therefore the device can skip the cpu entirely when accessing system ram and utilize DMA to quickly copy data to the device.

Writing an array

Writing an array consists of two steps:

1. Allocating the memory
2. Copying the data

To allocate the memory, a simple call to

```
_mem = clCreateBuffer($queue, $mem_flags, $size, $host_ptr, &err);
```

is enough. If you decided to copy the host pointer via the mem_flags, you are done. Otherwise you can copy the data whenever you like with

```
err = clEnqueueWriteBuffer($queue, _mem, $blocking, $offset, $size, $source, 0, null, null);
```

Read Host memory interaction online: https://riptutorial.com/opencl/topic/10173/host-memory-interaction

Chapter 4: Kernel Basics

Introduction

This topic aims to explain the fundamentals of writing kernels for opencl

Examples

Grayscale kernel

Lets build a kernel to generate a grayscale image. We will use image data which is defined using uints for each component and with order RGBA.

```
__constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
                                CLK_ADDRESS_CLAMP_TO_EDGE |
                                CLK_FILTER_LINEAR;

__kernel void Grayscale(__read_only image2d_t input, __write_only image2d_t output) {
    int2 gid = (int2)(get_global_id(0), get_global_id(1));
    int2 size = get_image_dim(input);

    if(all(gid < size)){
        uint4 pixel = read_imageui(input, sampler, gid);
        float4 color = convert_float4(pixel) / 255;
        color.xyz = 0.2126*color.x + 0.7152*color.y + 0.0722*color.z;
        pixel = convert_uint4_rte(color * 255);
        write_imageui(output, gid, pixel);
    }
}
```

Now lets walk through that code step by step. The first line creates a variable in the __constant memory region of type sampler_t. This sampler is used to further specify the access to our image data. Please refer to the Khronos Docs for a full documentation.

We allocated the input as read_only and the output as write_only before we called our kernel so we add those modifiers here.

image2d and image3d are always allocated on the global memory, therefore we can omit the __global modifier here.

Then we get our thread id which defines the pixel we are going to convert to grayscale. We also query the size to make sure that our thread is not accessing unallocated memory. This will definitley crash your kernel if you forget that.

After we made sure that we are a legitimate thread, we read our pixel out of our input image. We then convert it to float to avoid loss of decimal places, do some calculations, convert it back and write it into the output.

Kernel Skelleton

Lets walk through the most simple kernel there is and some variations of it

```
__kernel void myKernel() {
}
```

A kernel which can be started from your main code is identified by the __kernel keyword. A Kernel function can only have return type void.

```
__kernel void myKernel(float a, uint b, byte c) {

}
```

Of course you can create more functions which are not exposed as kernels. In this case you can just omit the __kernel modifier.

A function may expose variables like any other C/C++ function would. The only difference is when you want to reference memory. This applies to all pointers whether they are arguments or used in code.

```
float*  ptr;
```

is a pointer to a memory region only the executing thread has access to. In fact it is the same as

```
__private float* ptr;
```

There are four different memory region modifiers available. Inside the kernel you usually don't have to worry about it, but for arguments this is essential.

- __global: This modifier refers to a pointer which is placed in the global memory
- __constant: refers to a constant memory pointer
- __local: refers to a shared memory pointer
- __private: refers to a local memory pointer

In addition we can define how we want to access the memory

- no modifier: read and write
- __read_only
- __write_only

Those flags have to match the way we allocated the memory buffer back on our host.

Kernel ID

To properly work with the data each thread needs to know its position in the threadblock/global thread pool. This can be archieved with

```
get_local_id($dim);
get_global_id($dim);
```

Those two functions return the position of the thread relative to the threadblock or all threads.

```
get_working_dim();
```

Gets the total number of dimensions the kernel was launched with.

```
get_local_size($dim);
get_global_size($dim);
```

Gets the total number of threads in the threadblock or in total for a given dimension.

Caveat: make always sure that your thread is not exceeding your data size. This is very likely to happen and should always be checked for.

Vectors in OpenCL

Each fundamental opencl type has a vector version. You can use the vector type by appending the number of desired components after the type. Supported number of components are 2,3,4,8 and 16. OpenCL 1.0 does not offer three components.

You can initialize any vector using two ways:

- Provide a single scalar
- Satisfy all components

```
float4 a = (float4)(1); //a = (1, 1, 1, 1)
```

or

```
float4 b = (float4)(1, 2, 3, 4);
float4 c = (float4)(1, (float3)(2));
```

or any other combination of vectors which satisfy the number of components. To access the elements of a vector you can use different methods. You can either use indexing:

```
a[0] = 2;
```

or use literals. The advantage of literals is that you can combine them any way you want, well do that in a moment. You can access all vector components with

```
a.s0 = 2; // same as a[0] = 2
```

you can also combine multiple components into a new vector

```
a.s02 = (float2)(0, 0); // same as  a[0] = 0; a[2] = 0; or even a.s20 = (float2)(0, 0)
```

you can change the order of the components in any way you want.

```
a.s1423 = a.s4132; // flip the vector
```

but you cannot do something like

```
a.s11 = ... // twice the same component is not possible
```

There are some convenient shorthands for accessing vector components. The following shorthands only apply to sizes 2, 4, 8 and 16

```
a.hi //=a.s23 for vectors of size 4, a.4567 for size 8 and so on.
a.lo //=a.s01
a.even //=a.s02
a.odd //=a.13
```

For vector sizes 2,3 and 4 there are some additional shorthands

```
a.x //=a.s0
a.y //=a.s1
a.z //=a.s2
a.w //=a.s3
```

Gamma Correction kernel

Lets look at a gamma correction kernel

```
__constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
                                CLK_ADDRESS_CLAMP_TO_EDGE |
                                CLK_FILTER_LINEAR;

__kernel void Gamma(__read_only image2d_t input, __write_only image2d_t output, __constant
float gamma) {
    int2 gid = (int2)(get_global_id(0), get_global_id(1));
    int2 size = get_image_dim(input);

    if(all(gid < size)){
        uint4 pixel = read_imageui(input, sampler, gid);
        float4 color = convert_float4(pixel) / 255;
        color = pow(color, (float4)(gamma));
        pixel = convert_uint4_rte(color * 255);
        write_imageui(output, gid, pixel);
    }
}
```

Now lets walk through that code step by step. The first line creates a variable in the __constant memory region of type sampler_t. This sampler is used to further specify the access to our image data. Please refer to the Khronos Docs for a full documentation.

We allocated the input as read_only and the output as write_only before we called our kernel so we add those modifiers here.

image2d and image3d are always allocated on the global memory, therefore we can omit the __global modifier here. Our gamma value is located in __constant memory, so we specify that too.

Then we get our thread id which defines the pixel we are going to gamma correct. We also query the size to make sure that our thread is not accessing unallocated memory. This will definitley crash your kernel if you forget that.

After we made sure that we are a legitimate thread, we read our pixel out of our input image. We then convert it to float to avoid loss of decimal places, do some calculations, convert it back and write it into the output.

Read Kernel Basics online: https://riptutorial.com/opencl/topic/10174/kernel-basics

Chapter 5: OpenCL basic setup

Introduction

Before utilizing OpenCL, one has to set up their code to use it. This topic focuses on how to get opencl up and running in your project and execute a basic kernel. The examples are based on the C# wrapper OpenCL.NET but as the wrapper adds no abstraction to OpenCL the code will probably run with very few changes on C/C++ aswell.

Calls in C# may look as follows: 'Cl.GetPlatformIDs'. For the C-Style OpenCL Api you would call 'clGetPlatformIDs' and for the C++ style one 'cl::GetPlatformIDs'

Remarks

- NVidia, AMD and Intel have slightly different implementations of OpenCL but the known differences are (for my experience) limited to parenthesis requirements and implicit casts. Sometimes NVidia will crash your kernel while trying to figure out the correct overload for a method. In this case it helps to offer an explicit cast to aid the GPU. The problem was observed for runtime-compiled kernels.
- To get more information on the used calls in this topic, it is enough to google 'OpenCL ' followed by the function name. The Khronos group has a complete documentation on all parameters and datatypes available on their website.

Examples

Initializing the target Device

OpenCL Kernels can be either executed on the GPU or the CPU. This allows for fallback solutions, where the customer may have a very outdated system. The programmer can also choose to limit their functionality to either the CPU or GPU.

To get started using OpenCL, you'll need a 'Context' and a 'Device'. Both are structures defined by the OpenCL API (also known as cl::Context or clContext & ~Device) and define the used target processor.

To get your device and context, you need to query a list of available platforms, which each can host multiple devices. A Platform represents your physical GPU and CPU, while a device can further distinguish the contained computing units. For GPUs, most platforms will only have one device. But a CPU may offer an additional integrated GPU beside its CPU capabilities.

The context manages memory, command queues, the different kernels and programs. A context can either be limited to a single device but also reference multiple devices.

A quick API note before we start coding: Almost every call to OpenCL gives you an error value, either as return value or via a ref-value (pointer in C). Now lets get started.

```
ErrorCode err;
var platforms = Cl.GetPlatformIDs(out err);
if(!CheckError(err, "Cl.GetPlatformIDs")) return;
foreach (var platform in platforms) {
    foreach (var device in Cl.GetDeviceIDs(platform, DeviceType.Gpu, out err)) {
        if(!CheckError(err, "Cl.GetDeviceIDs")) continue;
        [...]
    }
}
```

This code snippet queries all available GPU devices on the system. You can now add them to a list or start your context directly with the first match. The 'CheckError(...)' function is a simple utility, that checks whether the error code has the success-value or a different one and can offer you some logging. It is recommended to use a seperate function or macro, because you will call that a lot.

ErrorCode is just an enum on the datatype cl_int for C#, C/C++ can compare the int value with predefined error constants as listed here:
https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/errors.html

You also might want to check whether the device supports all needed features, otherwise your kernels might crash at runtime. You can query a device capability with

```
Cl.GetDeviceInfo(_device, DeviceInfo.ImageSupport, out err)
```

This example asks the device whether it can execute image functions. For the next and final step we need to construct our context out of the collected devices.

```
_context = Cl.CreateContext(null, 1, new[] { _device }, ContextNotify, IntPtr.Zero, out err);
```

Some stuff is going on here. For C/C++ folks, IntPtr is a pointer address in C#. I will concentrate on the important parts here.

- The second parameter defines the number of devices you want to use
- The third parameter is an array of those devices (or a pointer in C/C++)
- And the third parameter is a function pointer for a callback function. This function will be used whenever errors happen inside the context.

For futher usage, you'll need to preserve your used devices and the context somewhere.

When you have finished all your OpenCL interaction you'll need to release the context again with

```
Cl.ReleaseContext(_context);
```

Compiling your Kernel

Kernels can be compiled at runtime on the target device. To do so, you need

- the kernel source code
- the target device on which to compile
- a context built with the target device

A quick terminology update: A program contains a collection of kernels. You can think of a program as a complete C/C++/C# source file, while kernels are the different function members of that file.

First you'll need to create a program out of your source code.

```
var program = Cl.CreateProgramWithSource(_context, 1, new[] { source }, null, out err);
```

You can combine multiple source files into one program and compile them together, which allows you to have kernels in different files and compile them together in one go.

In the next step you'll need to compile the program on your target device.

```
err = Cl.BuildProgram(program, 1, new[] { _device }, string.Empty, null, IntPtr.Zero);
```

Now here comes a little caveat: The error code only tells you, whether the function call itself was successfull but not whether your code did actually compile. To verify that, we have to query some additional information

```
BuildStatus status;
status = Cl.GetProgramBuildInfo(program, _device, ProgramBuildInfo.Status, out
err).CastTo<BuildStatus>();
if (status != BuildStatus.Success) {
    var log = Cl.GetProgramBuildInfo(program, _device, ProgramBuildInfo.Log, out err);
}
```

C/C++ people can ignore the cast at the end and just compare the returned integer with the corresponding constant.

The first call checks whether our build was actually successfull. If not we can retreive a log and see exactly where things went wrong. See the remarks for some common pitfals regarding different platforms.

Once the program is built, you need to extract your different kernels out of the compiled program. To do so you create your kernels with

```
_kernel = Cl.CreateKernel(_program, kernel, out err);
```

where 'kernel' is a string of the kernel name. When you are finished with your kernel, you need to release it with

```
Cl.ReleaseKernel(_kernel);
```

Creating a command queue

To initiate any operation on your devices, you'll need a command queue for each device. The Queue keeps track of different calls you did to the target device and keeps them in order. Most commands can also be executed either in blocking or non-blocking mode.

Creating a queue is pretty straightforward:

```
_queue = Cl.CreateCommandQueue(_context, _device, CommandQueueProperties.None, out err);
```

The basic interaction with your command queue is to enqueue different operations you want to perform, e.g. copy data to and from your device and launch a kernel.

When you have finished using the command queue you need to release the queue with a call to

```
Cl.ReleaseCommandQueue(_queue);
```

Executing the Kernel

So now we come down to the real stuff, executing your kernels on the parallel device. Please read about the hardware basics to fully understand the kernel dispatching.

First you'll need to set the kernel arguments before actually calling the kernel. This is done via

```
err = Cl.SetKernelArg(_kernel, $argumentIndex, $argument);
```

If you don't set every argument before launching the kernel, the kernel will fail.

Before we actually launch our kernel, we need to calculate the 'global work size' and the 'local work size'.

the global work size is the total number of threads that will be launched on your GPU. The local work size is the number of threads inside each thread block. The local work size can be omitted if the kernel does not need any special requirements. But if the local work size is given, the global work size has to be a multiple of the local work size.

The work sizes can either be one-dimensional, two dimensional or three dimensional. The choice on how many dimensions you want is entirely up to you and you can pick whatever suits your algorithm best.

Now that we decided on our work sizes we can call the kernel.

```
Event clevent;
err = Cl.EnqueueNDRangeKernel(_queue, _kernel, $dimensions, null, $globalWorkSize,
$localWorkSize, 0, null, out clevent);
```

The $dimensions define our desired number of dimensions, $globalWorkSize is an array of size $dimensions with the global Work size and the same for $localWorkSize. The last argument gives you an object which represents your currently executed kernel.

Read OpenCL basic setup online: https://riptutorial.com/opencl/topic/10169/opencl-basic-setup

Chapter 6: OpenCL hardware basics

Introduction

This topic introduces some of the underlying core mechanics of parallel computing which are needed to fully understand and utilize OpenCL.

Examples

Threads and Execution

The key of parallelism is to use multiple threads to solve a problem (duh.) but there are some differences to classical multithreaded programming in how threads are organized.

First lets talk about your typical GPU, for simplicities sake I'll focus on

A GPU has many processing cores, which make it ideal to execute many threads in parallel. Those cores are organized in Streaming Processors (SM, NVidia term), of which a GPU has a given number.

All threads running inside a SM are called a 'thread block'. There can be more threads on an SM than it has cores. The number of cores defines the so called 'Warp size' (NVidia term). Threads inside a thread block are sheduled in so called 'warps'.

A quick example to follow up: A typical NVidia SM has 32 processing cores, thus its warp size is 32. If my thread block now has 128 threads to run, they will be shedulled in 4 warps (4 warps * 32 warp size = 128 threads).

The warp size is rather important when choosing the number of threads later on.

All threads inside a single warp share a single instruction counter. That means those 32 threads are truly synchronized in that every thread executes every command at the same time. Here lies a performance pitfall: This also applies to branching statements in your kernel!

Example: I have a kernel that has an if statement and two branches. 16 of my threads inside a warp will execute branch one, the other 16 branch two. Up until the if statement, all threads inside the warp are in sync. Now half of them choose a different branch. What happens is that the other half will lay dormant until the wrong statement has finished executing on the first 16 threads. Then those threads will be dormant until the other 16 threads finished their branch.

As you can see, bad branching habits can severely slow down your parallel code, because both statements get executed in the worst case. If all threads inside a warp decide they only need one of the statements, the other one is completely skipped and no delay occurs.

Syncing threads is also not a simple matter. You can **only** sync threads withing a single SM. Everything outside the SM is unsyncable from inside the kernel. You'll have to write seperate kernels and launch them one after the other.

GPU Memory

The GPU offers six different memory regions. They differ in their latency, size and accessibility from different threads.

- Global Memory: The largest memory available and one of the few ones to exchange data with the host. This memory has the highest latency and is available for all threads.
- Constant Memory: A read only part of the global memory, which can only be read by other threads. Its advantage is the lower latency compared to the global memory
- Texture Memory: Also a part of constant memory, specifically designed for textures
- Shared Memory: This memory region is placed close to the SM and can only accessed by a single thread block. It offers way lower latency than the global memory and a bit less latency than the constant memory.
- Registers: Only accessible by a single thread and the fastest memory of them all. But if the compiler detects that there are not enough Registers for the kernel needs, it will outsource variables to local memory.
- Local Memory: A thread-only accessible part of memory in the global memory region. Used as a backup for registers, to be avoided if possible.

Memory access

The typical scenario for your memory usage is to store the source data and the processed data in the global memory. When a threadblock starts, it first copies all relevant parts into the shared memory before getting their parts into the registers.

Memory access latency also depends on your memory strategy. If you blindly access data you will get the worst performance possible.

The different memories are organized in so-called 'banks'. Each memory request for a bank can be handled in a single clock cycle. The number of banks in the shared memory equals the warp size. The memory speed can be increased by avoiding conflicting bank access inside a single warp.

To copy shared memory from or to global memory the fastest way is to 'align' your memory calls. This means that the first thread in a warp should access the first element in the bank of both the shared and global memory. The second thread the second element and so on. This call will be optimized into a single memory transfer instruction which copies the whole bank to the target memory in one go.

Read OpenCL hardware basics online: https://riptutorial.com/opencl/topic/10168/opencl-hardware-basics

Chapter 7: Pseudo-Random Number Generator Kernel Example

Parameters

Parameter	Details
__global unsigned int * rnd_buffer	unsigned int is standardised by the OpenCL standard as being 32-bit
*	__global means device's main memory for read/write access
*	rnd_buffer is just a name in scope of "opencl program"(not host but device)

Examples

Using Thomas Wang's integer hash function

Auxilliary function that takes a seed and evaluates:

```
uint wang_hash(uint seed)
{
        seed = (seed ^ 61) ^ (seed >> 16);
        seed *= 9;
        seed = seed ^ (seed >> 4);
        seed *= 0x27d4eb2d;
        seed = seed ^ (seed >> 15);
        return seed;
 }
```

another auxilliary function that uses it to initialize a buffer location shown by "id":

```
 void wang_rnd_0(__global unsigned int * rnd_buffer,int id)
 {
     uint maxint=0;
     maxint--;
     uint rndint=wang_hash(id);
     rnd_buffer[id]=rndint;
 }
```

and another doing extra float output between 0 and 1

```
 float wang_rnd(__global unsigned int * rnd_buffer,int id)
 {
     uint maxint=0;
     maxint--; // not ok but works
```

```
    uint rndint=wang_hash(rnd_buffer[id]);
    rnd_buffer[id]=rndint;
    return ((float)rndint)/(float)maxint;
}
```

Initializer kernel:

```
__kernel void rnd_init(__global unsigned int * rnd_buffer)
{
      int id=get_global_id(0);
      wang_rnd_0(rnd_buffer,id);  // each (id) thread has its own random seed now
}
```

Single iteration kernel:

```
__kernel void rnd_1(__global unsigned int * rnd_buffer)
{
     int id=get_global_id(0);

     // can use this to populate a buffer with random numbers
     // concurrently on all cores of a gpu
     float thread_private_random_number=wang_rnd(rnd_buffer,id);
}
```

Read Pseudo-Random Number Generator Kernel Example online:
https://riptutorial.com/opencl/topic/5893/pseudo-random-number-generator-kernel-example

Credits

S. No	Chapters	Contributors
1	Getting started with opencl	benshope, Community, CShark, Dithermaster, Dschoni, huseyin tugrul buyukisik, Umar Arshad
2	Atomic Operations	huseyin tugrul buyukisik
3	Host memory interaction	CShark
4	Kernel Basics	CShark
5	OpenCL basic setup	CShark
6	OpenCL hardware basics	CShark
7	Pseudo-Random Number Generator Kernel Example	huseyin tugrul buyukisik, silverclaw