OpenCL Overview

Ofer Rosenberg

Contributors: Tim Mattson (Intel), Aaftab Munshi (Apple)

Agenda

- OpenCL intro
  - GPGPU in a nutshell
  - OpenCL roots
  - OpenCL status
- OpenCL 1.0 deep dive
  - Programming Model
  - Framework API
  - Language
  - Embedded Profile & Extensions
- Summary

GPGPU in a nutshell

Disclaimer:

1. GPGPU is a lot of things to a lot of people.
   This is my view & vision on GPGPU...

2. GPGPU is a huge subject, and this is a Nutshell.
   I recommend Kayvon's lecture at http://s08.idav.ucdavis.edu/

GPGPU in a nutshell

```
#include <stdio.c>
…

void main (int argc, char* argv[])
{

   …


   for (i=0 ; i < iBuffSize; i++)
         C[i] = A[i] + B[i];

   …

}
```

- On the right there is a very artificial example to explain GPGPU.
- A simple program , with a “for” loop which takes two buffers and adds them into a third buffer

(did we mention the word artificial yet ???)

GPGPU in a nutshell

```
#include <stdio.c>
…

void main (int argc, char* argv[])
{
  …

  _beginthread(vec_add, 0, A, B, C, iBuffSize/2);
  _beginthread(vec_add, 0, A[iBuffSize/2],
                   B[iBuffSize/2], C[iBuffSize/2],
                   iBuffSize/2);
  …

}

void vec_add (const float *A, const float *B, float *C,
                  int iBuffSize)
{
     __m128 vC, vB, vA;
     for (i=0 ; i < iBuffSize/4; i++)
     {
         vA = _mm_load_sp(&A[i*4]);
         vB = _mm_load_sp(&B[i*4]);
         vC = _mm_add_ps (vA, vB);
         _mm_store_ps (&C[i*4], vC);
     }
     _endthread();
}
```

- The example after an expert visit:
  - Dual threaded to support Dual Core
  - SSE2 code is doing a vectorized operation

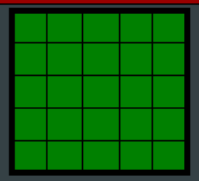
Traditional GPGPU…

- Write in graphics language and use the GPU
- Highly effective, but :
  - The developer needs to learn another (not intuitive) language
  - The developer was limited by the graphics language

GPGPU example – Adding Vectors

- Place arrays into 2D textures
- Convert loop body into a shader
- Loop body = Render a quad
  - Needs to cover all the pixels in the output
  - 1:1 mapping between pixels and texels
- Readback framebuffer into result array

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24



```
float a[5*5];
float b[5*5];
float c[5*5];
//initialize vector a
//initialize vector b
for(int i=0; i<5*5; i++)
{
    c[i] = a[i] + b[i];
}
```

```
!!ARBfp1.0
TEMP R0;
TEMP R1;
TEX R0, fragment.position, texture[0], 2D;
TEX R1, fragment.position, texture[1], 2D;
ADD R0, R0, R1;
MOV fragment.color, R0;
```

Mike Houston - Stanford University Graphics Lab

GPGPU reloaded

```
#include <stdio.c>
…

Void main (int argc, char* argv[])
{

   …


   for (i=0 ; i < iBuffSize; i++)
         C[i] = A[i] + B[i];

   …

}
```

```
__kernel void dot_product (__global const float4 *a,
                           __global const float4 *b,
                           __global float *c)
{
    int gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}
```

OpenCL

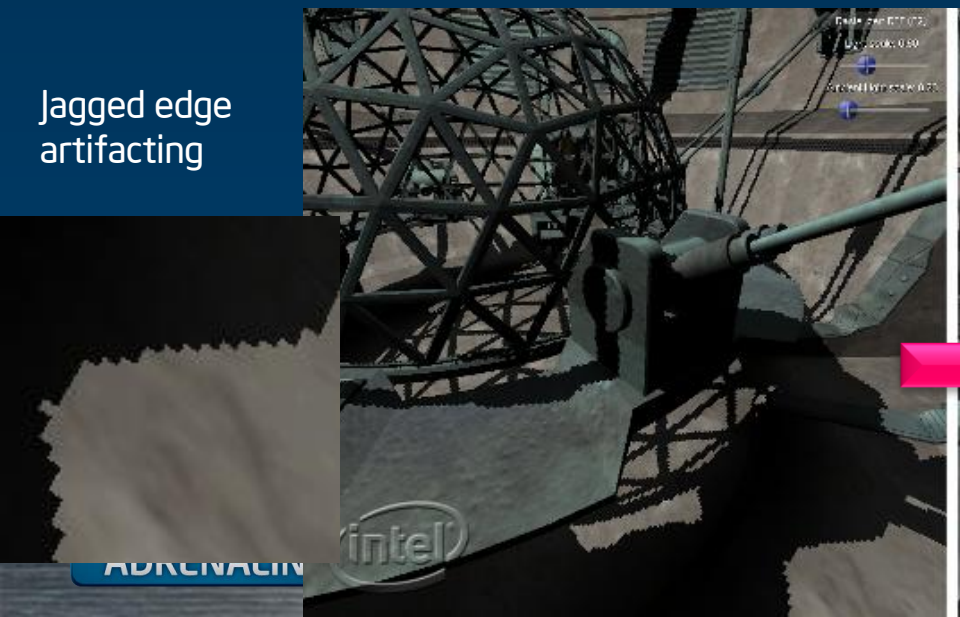
- CUDA was a disruptive technology
  - Write C on the GPU
  - Extend to non-traditional usages
  - Provide synchronization mechnaism
- OpenCL deepens and extends the revolution
- GPGPU now used for games to enhance the standard GFX pipe
  - Physics
  - Advanced Rendering

GPGPU also in Games…

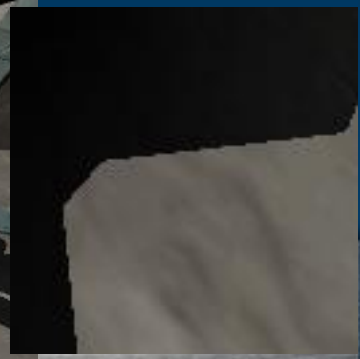
Non-interactive point light

Dynamic light affects character & environment

Jagged edge artifacting



Clean edge details



A new type of programming…

"The way the processor industry is going is to add more and more cores, but nobody knows how to program those things. I mean, two, yeah; four, not really; eight, forget it."

Steve Jobs, NY Times interview, June 10 2008

What About GPU's ?

NVIDIA G80: 16 Cores, 8 HW threads per core

Larrabee: XX Cores, Y HW threads per core

"Basically it lets you use graphics processors to do computation," he said. "It's way beyond what Nvidia or anyone else has, and it's really simple."

Steve Jobs on OpenCL, NY Times interview, June 10 2008

http://bits.blogs.nytimes.com/2008/06/10/apple-in-parallel-turning-the-pc-world-upside-down/

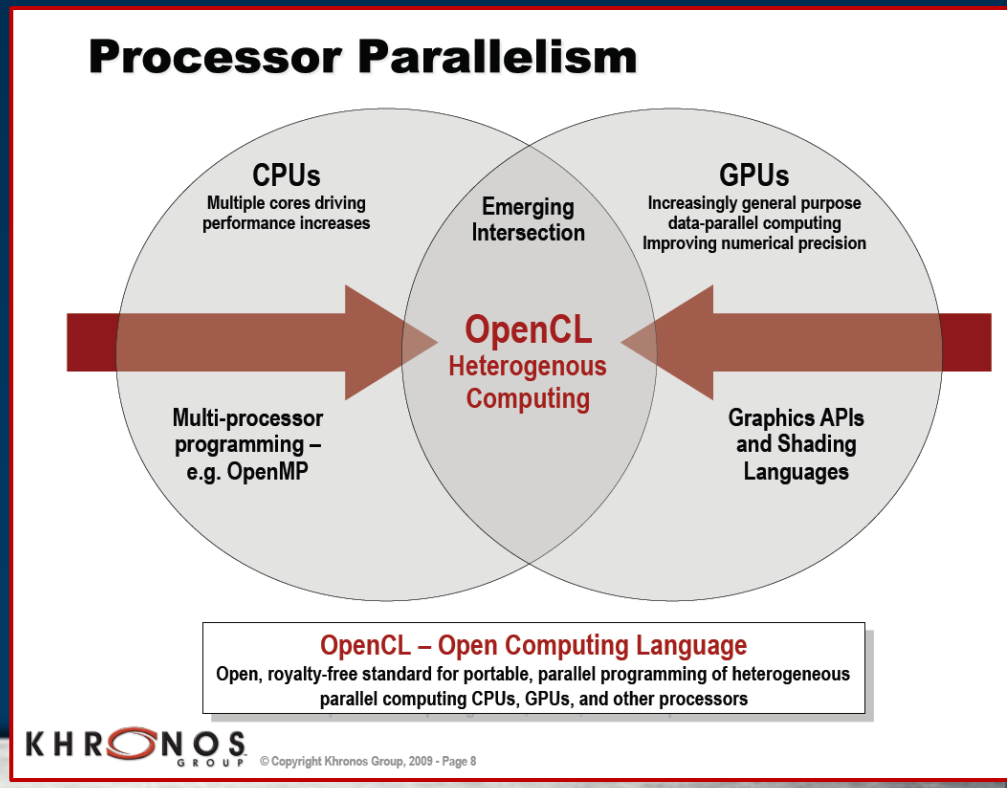
OpenCL in a nutshell

- OpenCL is :
  - An opened Standard managed by the Khronos group (Cross-IHV, Cross-OS)
    - Influenced & guided by Apple
    - Spec 1.0 approved Dec'08
  - A system for executing short "Enhanced C" routines (kernels) across devices
    - All around Heterogeneous Platforms – Host & Devices
    - Devices: CPU, GPU, Accelerator (FPGA)
  - Skewed towards GPU HW
    - Samplers, Vector Types, etc.
  - Offers hybrid execution capability

- OpenCL is not:
  - OpenGL, or any other 3D graphics language
  - Meant to replace C/C++ (don't write the entire application in it...)

Khronos WG Key contributors

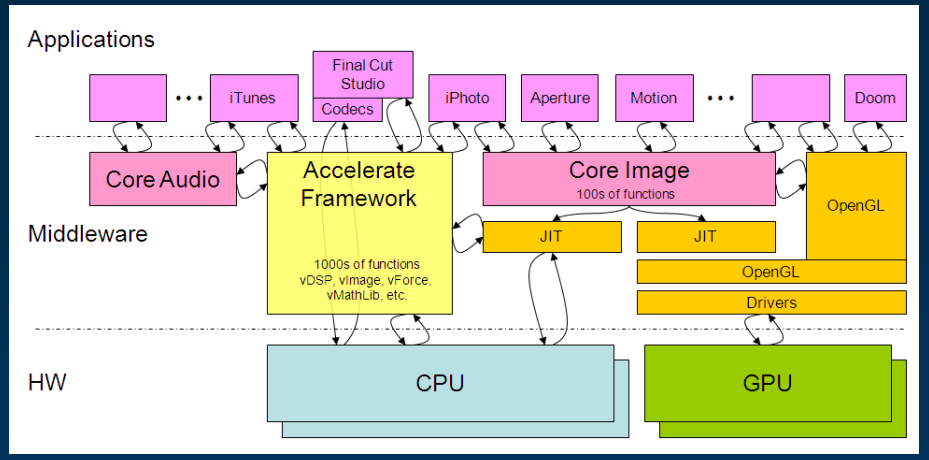
Apple, NVidia, AMD, Intel, IBM, RapidMind, Electronic Arts (EA), 3DLABS, Activision Blizzard, ARM, Barco, Broadcom, Codeplay, Ericsson, Freescale, HI, Imagination Technologies, Kestrel Institute, Motorola, Movidia, Nokia, QNX, Samsung, Seaweed, Takumi, TI and Umeå University.

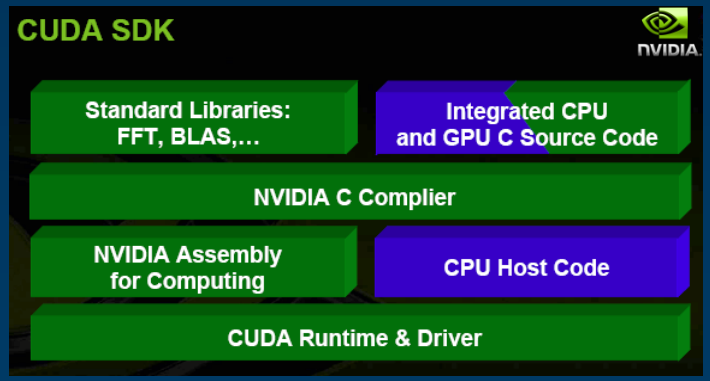


VISUAL ADRENALINE intel

The Roots of OpenCL

- Apple Has History on GPGPU…
  - Developed a Framework called "Core Image" (& Core Video)
  - Based on OpenGL for GPU operation and Optimized SSE code for CPU
- Feb 15th 2007 – NVIDIA introduces CUDA
  - Stream Programming Language – C with extensions for GPU
  - Supported by Any Geforce 8 GPU
  - Works on XP & Vista (CUDA 2.0)
  - Amazing adoption rate
    - 40 university courses worldwide
    - 100+ Applications/Articles
- Apple & NVIDIA cooperate to create OpenCL – Open Compute Language





OpenCL Status

- Apple Submitted the OpenCL 1.0 specification draft to Khronos (owner of OpenGL)
- June 16th 2008 - Khronos established the "Compute Working Group"
  - Members: AMD, Apple, Ardites, ARM, Blizzard, Broadcom, Codeplay, EA, Ericsson, Freescale, Hi Corp., IBM, Imagination Technologies, Intel, Kestrel Institute, Movidia, Nokia, Nvidia, Qualcomm, Rapid Mind, Samsung, Takumi and TI.
- Dec. 1st 2008 – OpenCL 1.0 ratification
- Apple is expected to release "Snow Leopard" Mac OS (containing OpenCL 1.0) by end of 2009
- Apple already began porting code to OpenCL

KHRONOS GROUP
Open Standards for Media Authoring and Acceleration

APIs & Technologies | Developers | News & Events | Adopters | Members | About Khronos

Home > News & Events > Press Releases > Khronos Launches Heterogeneous Computing Initiative

Khronos Press Releases

Khronos Launches Heterogeneous Computing Initiative

Call for industry participation to create open, royalty-free standard for programming parallel computing across GPUs and CPUs

For Information: Elizabeth Riegel, Khronos Group Marketing Director
elizabeth@goldstandardgroup.com
Toll-free US: +1 888 222 1899
From outside US: +1 415 869 8627

日本語 Japanese Translation (pdf)

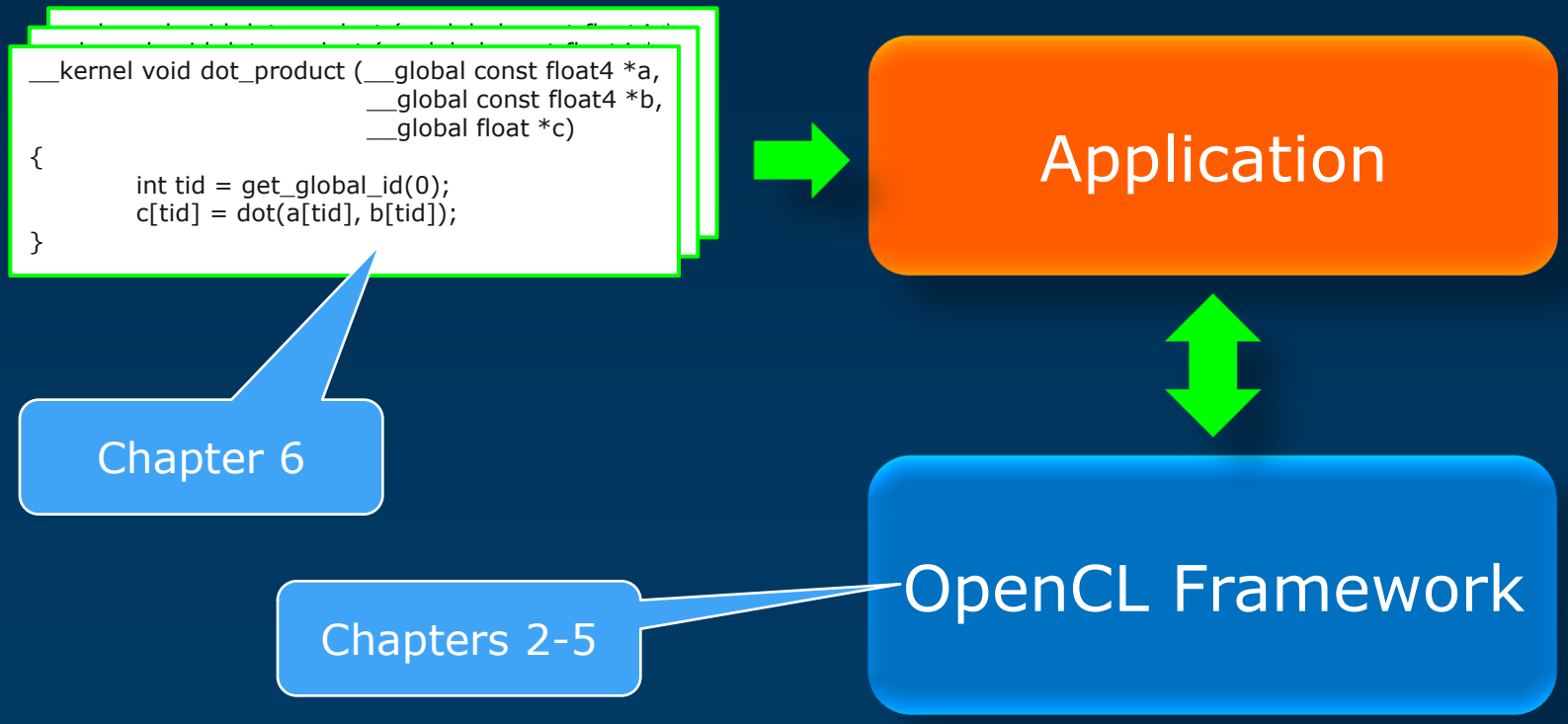
June 16th 2008 – San Francisco, CA – The Khronos™ Group announced today the formation of a new Compute Working Group

Agenda

- OpenCL intro
  - GPGPU in a nutshell
  - OpenCL roots
  - OpenCL status
- OpenCL 1.0 deep dive
  - Programming Model
  - Framework API
  - Language
  - Embedded Profile & Extensions
- Summary

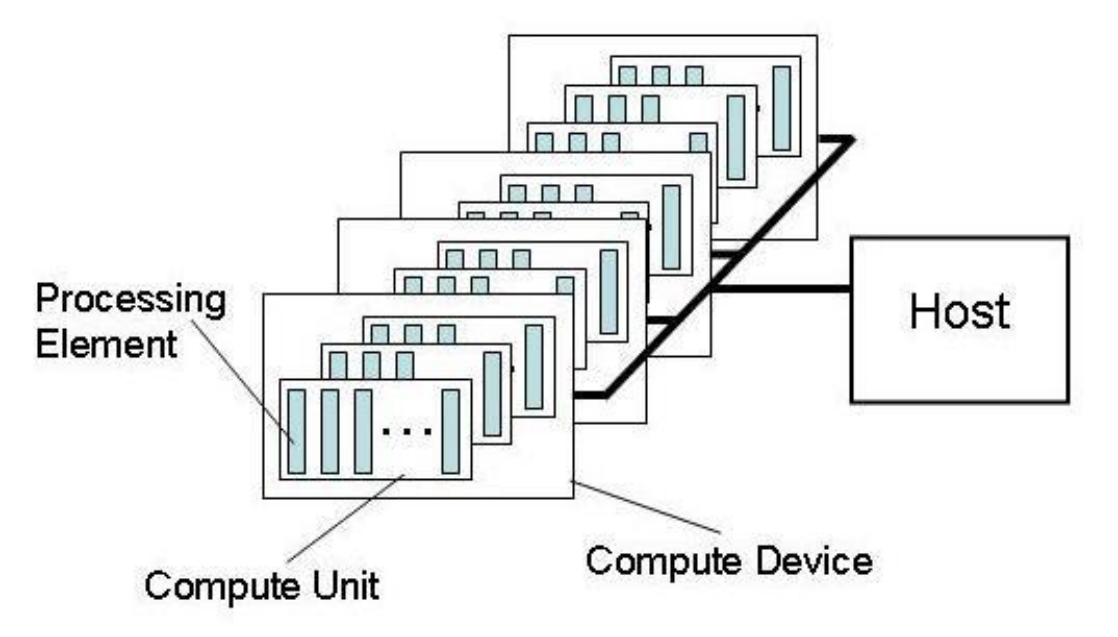
OpenCL from 10,000 feet…

- The Standard Defines two major elements:
  - The Framework/Software Stack
  - The Language



OpenCL Platform Model

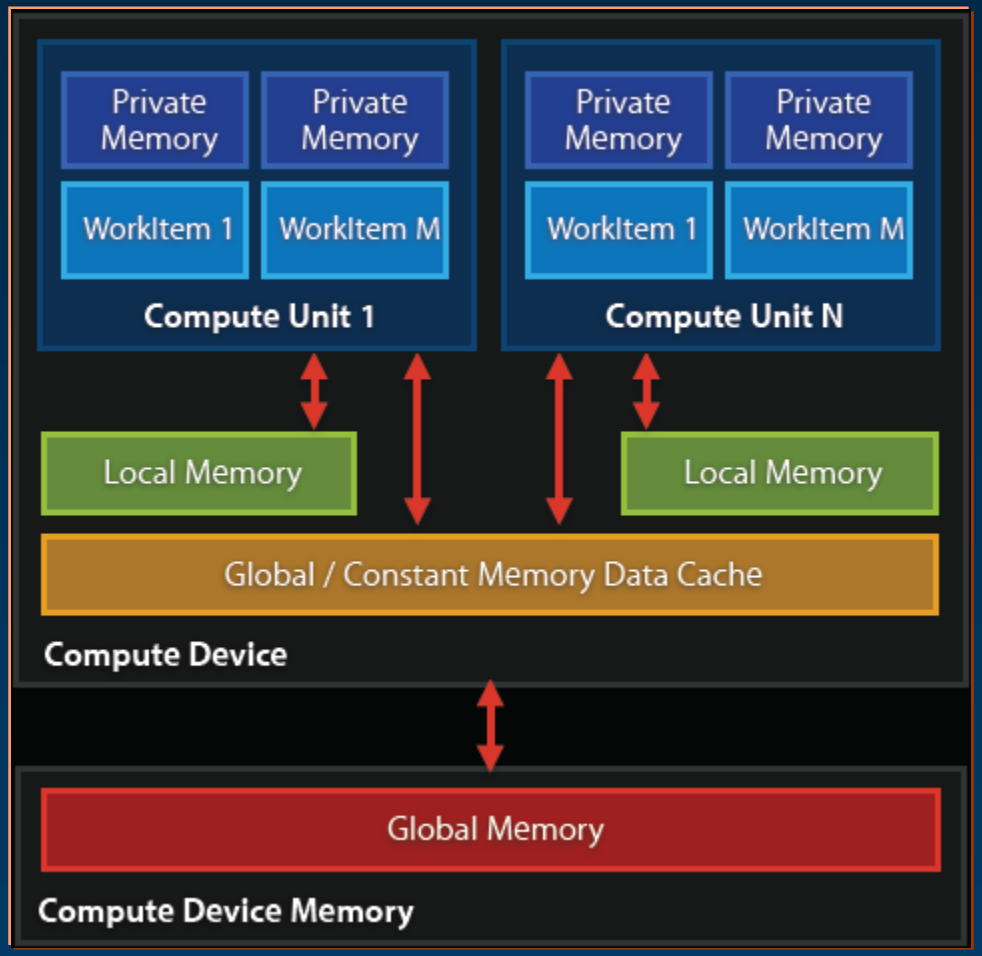
- The basic platform composed of a Host and a few Devices
- Each device is made of a few compute units (well, cores…)
- Each compute unit is made of a few processing elements (virtual scalar processor)



Under OpenCL the CPU is also compute device

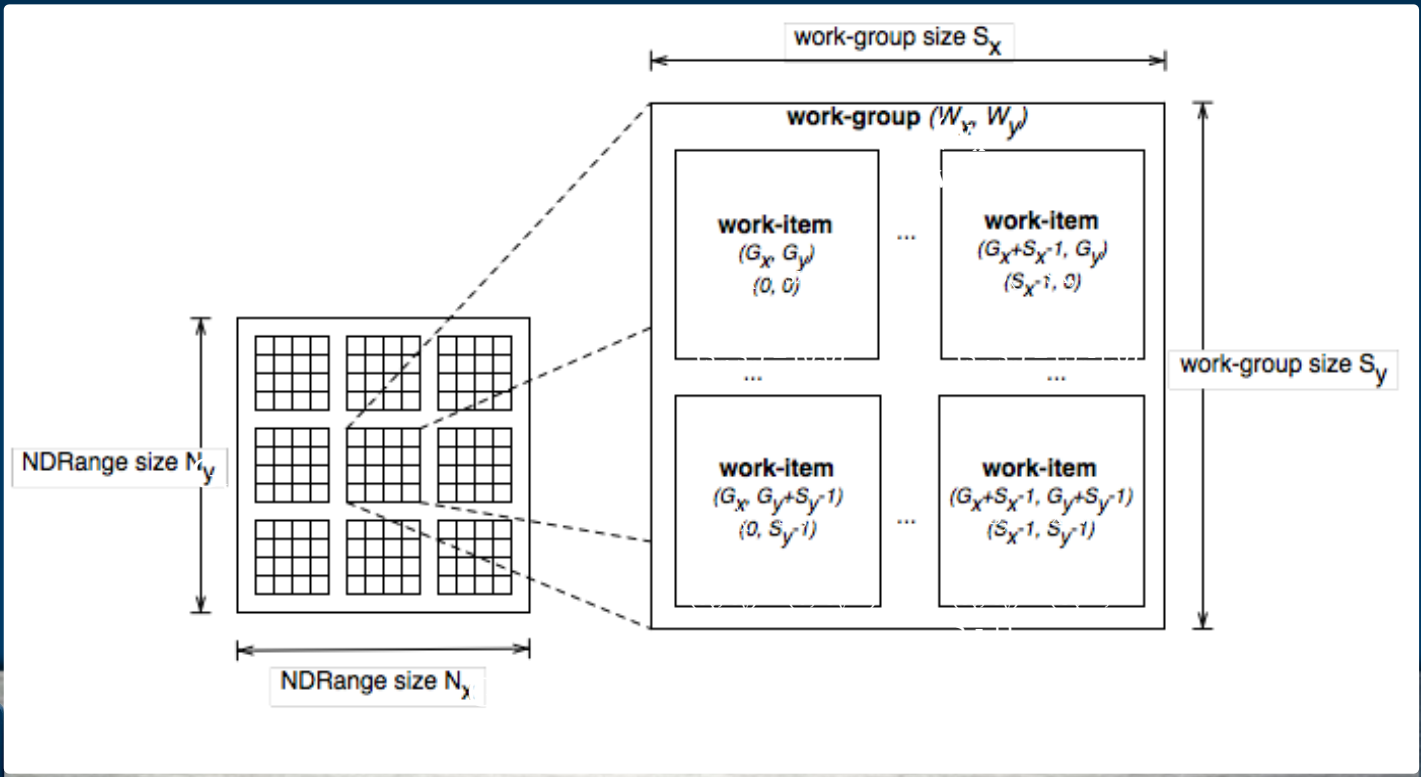
Compute Device Memory Model

- Compute Device – CPU or GPU
- Compute Unit = Core
- Compute Kernel
  - A function written in OpenCL C
  - Mapped to Work Item(s)
- Work-item
  - A single copy of the compute kernel, running on one data element
  - In Data Parallel mode, kernel execution contains multiple work-items
  - In Task Parallel mode, kernel execution contains a single work-item
- Four Memory Types:
  - Global : default for images/buffers
  - Constant : global const variables
  - Local : shared between work-items
  - Private : kernel internal variables



Execution Model

- Host defines a command queue and associates it with a context (devices, kernels, memory, etc).
- Host enqueues commands to the command queue
- Kernel execution commands launch work-items: i.e. a kernel for each point in an abstract Index Space
- Work items execute together as a work-group.



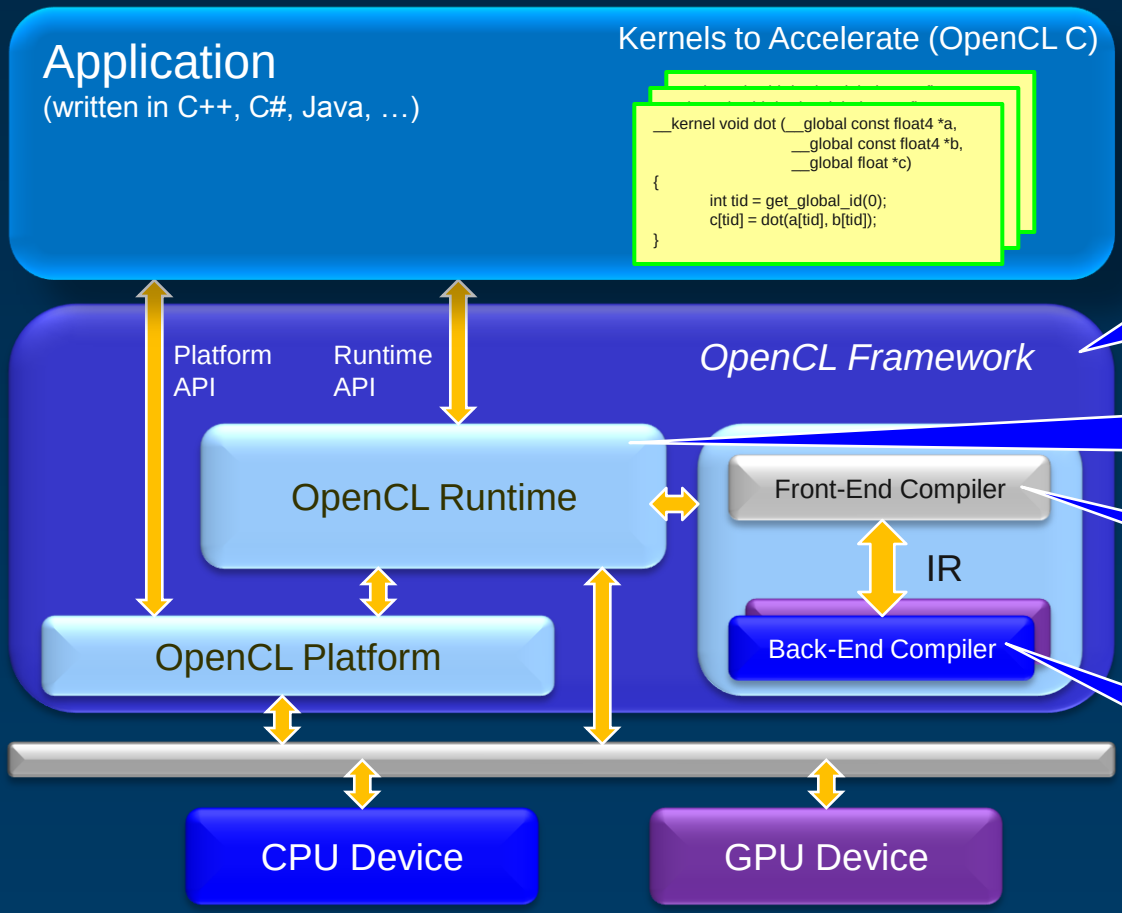
Programming Model

- Data Parallel, SPMD
  - Work-items in a work-group run the same program
  - Update data structures in parallel using the work-item ID to select data and guide execution.
- Task Parallel
  - One work-item per work group … for coarse grained task-level parallelism.
  - Native function interface: trap-door to run arbitrary code from an OpenCL command-queue.

Compilation Model

- OpenCL uses dynamic (runtime) 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 (when the developer complied it using the IDE)
- Dynamic compilation:
  - Also known as runtime compilation
  - Step 1 : The code is complied to an Intermediate Representation (IR), which is usually an assembler of a virtual machine. This step is known as offline compilation, and it's done by the Front-End compiler
  - Step 2: The IR is compiled to a machine code for execution. This step is much shorter. It is known as online compilation, and it's done by the Back-end compiler
- In dynamic compilation, step 1 is done usually only once, and the IR is stored. The App loads the IR and performs step 2 during the App's runtime (hence the term...)

OpenCL Framework overview



Open CL Framework
Allows applications to use a host and one or more OpenCL devices as a single heterogeneous parallel computer system.

OpenCL Runtime:
Allows the host program to manipulate contexts once they have been created.

Front End Compiler:
Compile from source into common binary intermediate that contain OpenCL kernels.

Backend compiler:
Compile from general intermediate binary into a device specific binary with device specific optimizations

Note:
The CPU is both the Host and a compute device

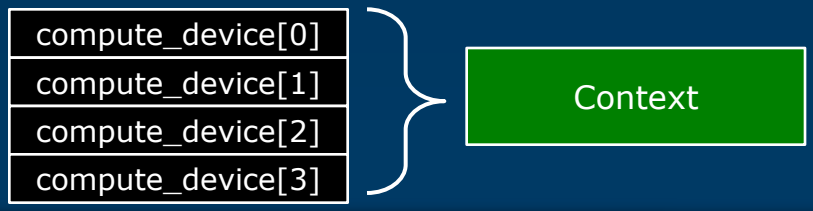
Agenda

- OpenCL intro
  - GPGPU in a nutshell
  - OpenCL roots
  - OpenCL status
- OpenCL 1.0 deep dive
  - Programming Model
  - Framework API
  - Language
  - Embedded Profile & Extensions
- Summary

The Platform Layer

- Query the Platform Layer
  - *clGetPlatformInfo*
- Query Devices (by type)
  - *clGetDeviceIDs*
- For each device, Query Device Configuration
  - *clGetDeviceConfigInfo*
  - *clGetDeviceConfigString*
- Create Contexts using the devices found by the "get" function
- Context is the central element used by the runtime layer to manage:
  - Command Queque
  - Memory objects
  - Programs
  - Kernels

clCreateContext



clGetDeviceIDs (cl_device_type device_type ...

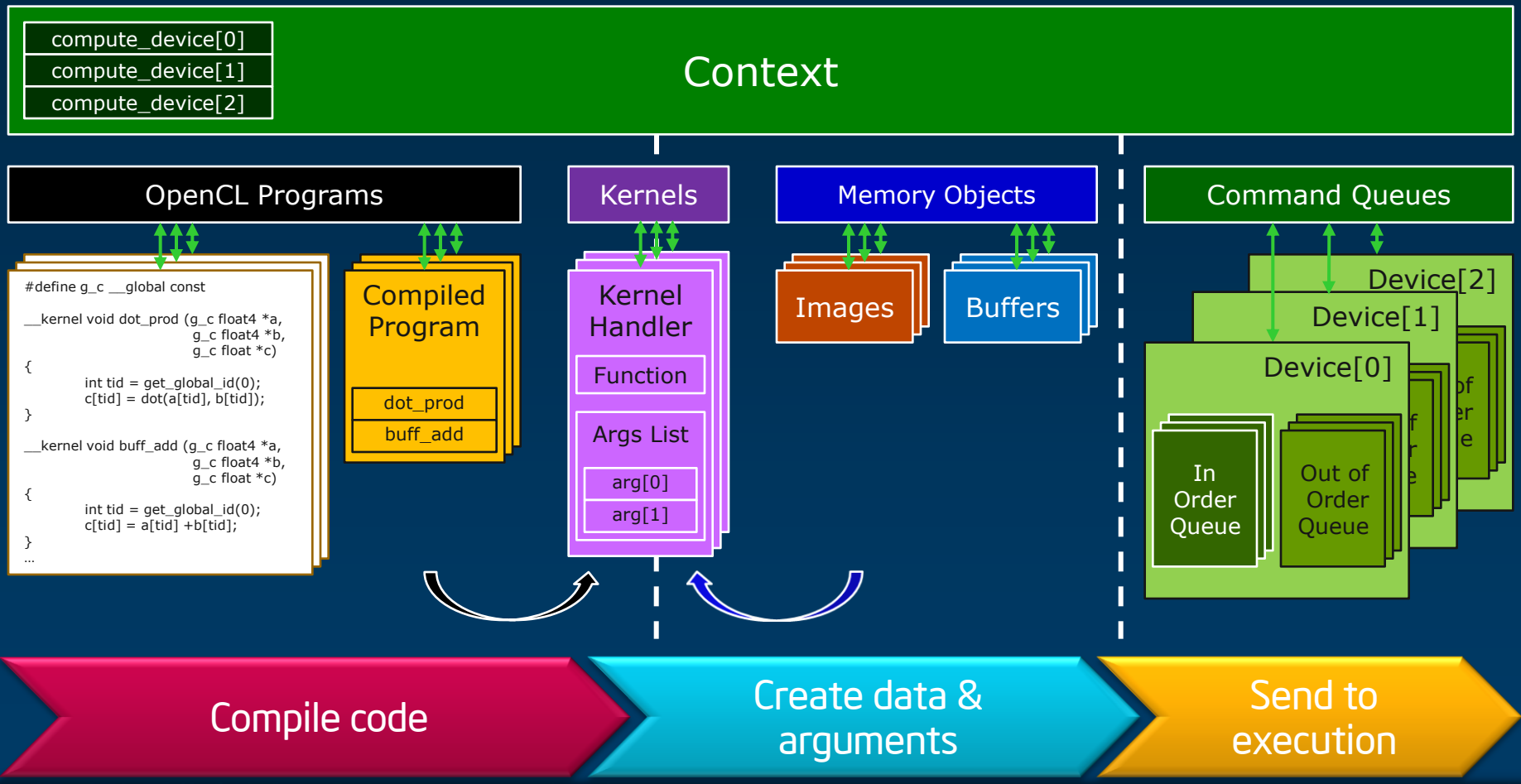
cl_device_type	Description
CL_DEVICE_TYPE_CPU	An OpenCL device that is the host processor. The host processor runs the OpenCL implementations and is a single or multi-core CPU.
CL_DEVICE_TYPE_GPU	An OpenCL device that is a GPU. By this we mean that the device can also be used to accelerate a 3D API such as OpenGL or DirectX.
CL_DEVICE_TYPE_ACCELERATOR	Dedicated OpenCL accelerators (for example the IBM CELL Blade). These devices communicate with the host processor using a peripheral interconnect such as PCIe.
CL_DEVICE_TYPE_DEFAULT	The default OpenCL device in the system.
CL_DEVICE_TYPE_ALL	All OpenCL devices available in the system.

clGetDeviceInfo (cl_device_id device,
cl_device_info param_name,

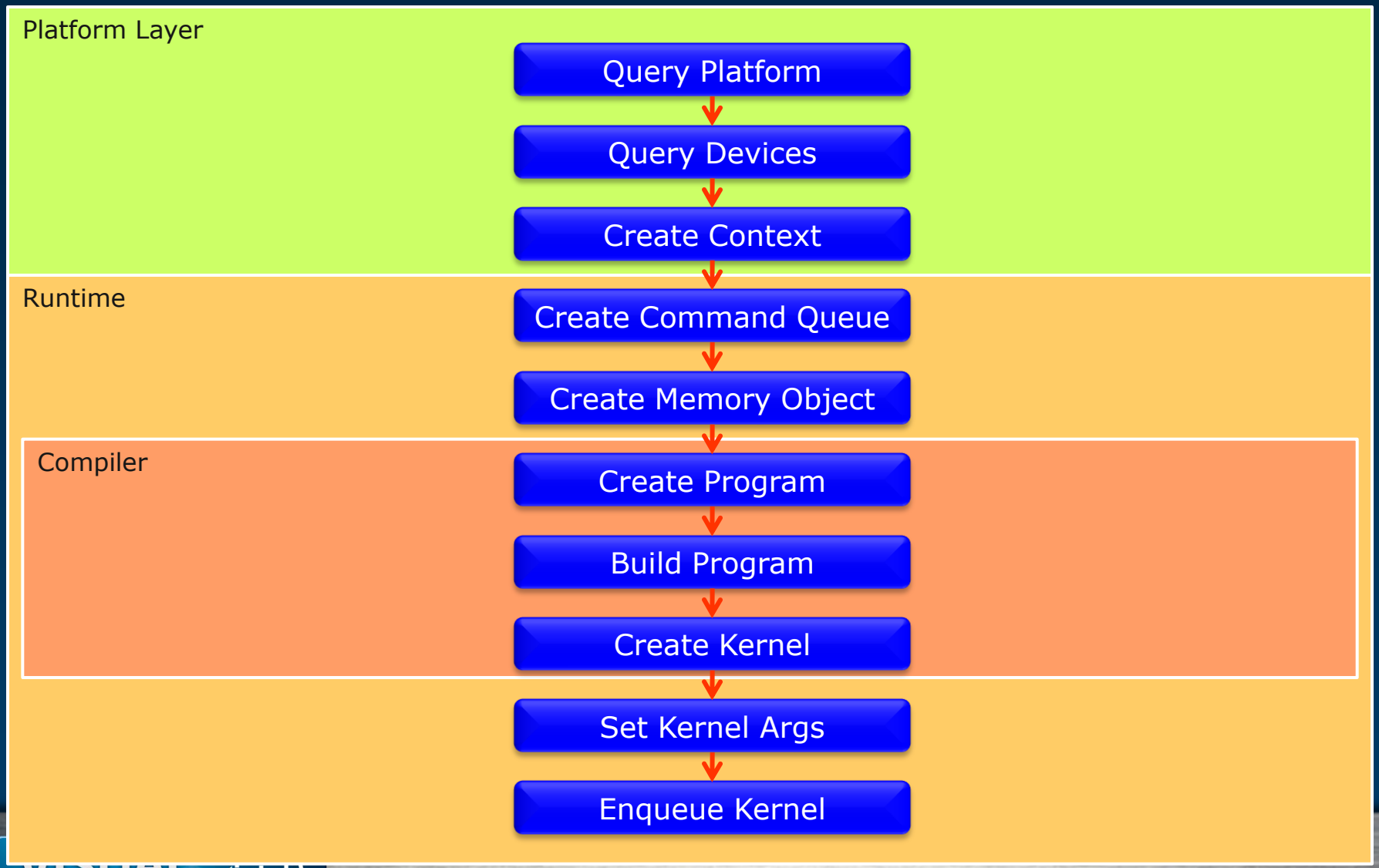
cl_device_info	Description
CL_DEVICE_TYPE	The OpenCL device type. Currently supported values are: CL_DEVICE_TYPE_CPU, CL_DEVICE_TYPE_GPU, CL_DEVICE_TYPE_ACCELERATOR, CL_DEVICE_TYPE_DEFAULT or a combination of the above.
CL_DEVICE_MAX_COMPUTE_UNITS	The number of parallel compute cores on the OpenCL device. The minimum value is one.
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS	Maximum dimensions that specify the global and local work-item IDs used by the data-parallel execution model. (Refer to clEnqueueNDRangeKernel). The minimum value is 3.
CL_DEVICE_MAX_WORK_GROUP_SIZE	Maximum number of work-items in a work-group executing a kernel using the data-parallel execution model. (Refer to clEnqueueNDRangeKernel).
CL_DEVICE_MAX_CLOCK_FREQUENCY	Maximum configured clock frequency of the device in MHz.
CL_DEVICE_ADDRESS_BITS	Device address space size specified as an unsigned integer value in bits. Currently supported values are 32 or 64 bits.
CL_DEVICE_MAX_MEM_ALLOC_SIZE	Max size of memory object allocation in bytes. The minimum value is max (1/4th of CL_DEVICE_GLOBAL_MEM_SIZE, 128*1024*1024)
And many more : 40+ parameters	

OpenCL Runtime

Everything in OpenCL Runtime is happening within a context



OpenCL "boot" process



OpenCL C Programming Language in a Nutshell

- Derived from ISO C99
- A few restrictions:
  - Recursion
  - Function pointers
  - Functions in C99 standard headers
- New Data Types
  - New Scalar types
  - Vector Types
  - Image types
- Address Space Qualifiers
- Synchronization objects
  - Barrier
- Built-in functions
- IEEE 754 compliant with a few exceptions

Type	Description
char*n*	A 8-bit signed two's complement integer vector.
uchar*n*	A 8-bit unsigned integer vector.
short*n*	A 16-bit signed two's complement integer vector.
ushort*n*	A 16-bit unsigned integer vector.
int*n*	A 32-bit signed two's complement integer vector.
uint*n*	A 32-bit unsigned integer vector.
long*n*	A 64-bit signed two's complement integer vector.
ulong*n*	A 64-bit unsigned integer vector.
float*n*	A float vector.

```
__global float4 *color; // An array of float4
typedef struct {
     float a[3];
     int b[2];
} foo_t;
__global image2d_t texture; // A 2D texture image

__kernel void stam_dugma(
      __global float *output,
      __global float *input,
      __local float *tile)
{
      // private variables
      int in_x, in_y ;
      const unsigned int lid = get_local_id(0));

      // declares a pointer p in the __private
      // that points to an int object in __global
      __global int *p;
```

Agenda

- OpenCL intro
  - GPGPU in a nutshell
  - OpenCL roots
  - OpenCL status
- OpenCL 1.0 deep dive
  - Programming Model
  - Framework API
  - Language
  - Embedded Profile & Extensions
- Summary

OpenCL Extensions

- As in OpenGL, OpenCL supports Specification Extensions
- Extension is an optional feature, which might be supported by a device, but not part of the "Core features" (Khronos term)
  - Application is required to query the device using CL_DEVICE_EXTENSIONS parameter
- Two types of extensions:
  - Extensions approved by Khronos OpenCL working group
    - Uses the "KHR" in functions/enums/etc.
    - Might be promoted to required Core feature on next versions of OpenCL
  - Extensions which are Vendor Specific
- The specification already provides some KHR extensions

OpenCL 1.0 KHR Extensions

- Double Precision Floating Point
  - Support Double as data type and extend built-in functions to support it
- Selecting Rounding Mode
  - Add to the mandatory “round to nearest” : “round to nearest even”, “round to zero”, “round to positive infinity”, “round to negative infinity”
- Atomic Functions (for 32-bit integers, for Local memory, for 64-bit)
- Writing to 3D image memory objects
  - OpenCL mandates only read.
- Byte addressable stores
  - In OpenCL core, Write to Pointers is limited to 32bit granularity
- Half floating point
  - Add 16bit Floating point type

OpenCL 1.0 Embedded Profile

- A "relaxed" version for embedded devices (as in OpenGL ES)
- No 64bit integers
- No 3D images
- Reduced requirements on Samplers
  - No CL_FILTER_LINEAR for Float/Half
  - Less addressing modes
- Not IEEE 754 compliant on some functions
  - Example: Min Accuracy for atan() >= 5 ULP
- Reduced set of minimal device requirements
  - Image height/width : 2048 instead of 8192
  - Number of samplers : 8 instead of 16
  - Local memory size : 1K instead of 16K
  - More...

Agenda

- OpenCL intro
  - GPGPU in a nutshell
  - OpenCL roots
  - OpenCL status
- OpenCL 1.0 deep dive
  - Programming Model
  - Framework API
  - Language
  - Embedded Profile & Extensions
- Summary

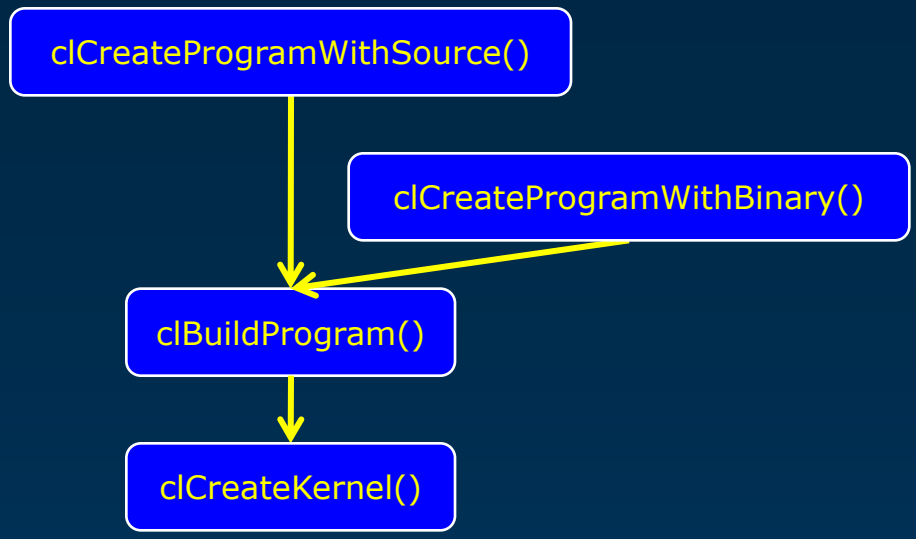
OpenCL Unique Features

- As a Summary, here are some unique features of OpenCL :
- An Opened Standard for Cross-OS, Cross-Platform, heterogeneous processing
  - Khronos owned
- Creates a unified, flat, system model where the GPU, CPU and other devices are treated (almost) the same
- Includes Data & Task Parallelism
  - Extend GPGPU beyond the traditional usages
- Supports Native functions (C++ interop)
- Derived from ISO C99 with additional types, functions, etc. (and some restrictions)
- IEEE 754 compliant

Backups

Building OpenCL Code

1. Creating OpenCL Programs
   - From Source : receive array of strings
   - From Binaries : receive array of binaries
     - Intermediate Representation
     - Device specific executable
2. Building the programs
   - The developer can define a subgroup of devices to build on
3. Creating Kernel objects
   - Single kernel : according to kernel name
   - All kernels in program

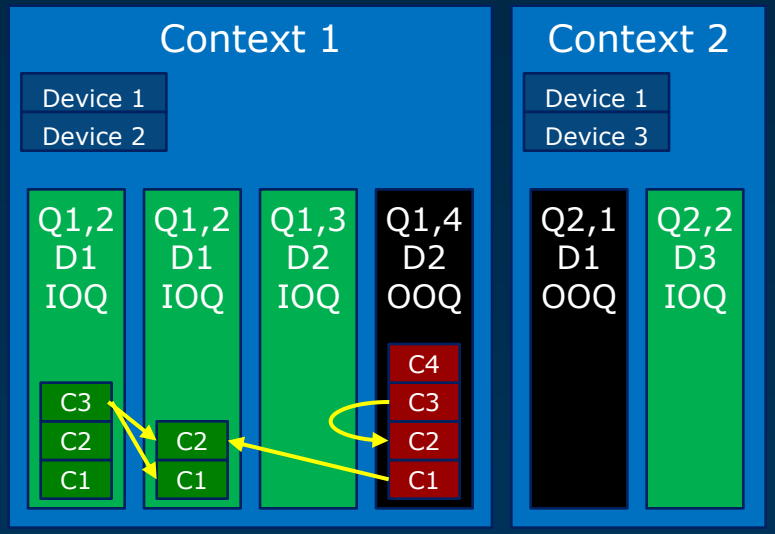


OpenCL supports dynamic compilation scheme – the Application uses "create from source" on the first time and then use "create from binaries" on next times

```
cl_program clCreateProgramWithSource (cl_context context,
                                      cl_uint count,
                                      const char **strings,
                                      const size_t *lengths,
                                      cl_int *errcode_ret)

cl_program clCreateProgramWithBinary (cl_context context,
                                      cl_uint num_devices,
                                      const cl_device_id *device_list,
                                      const size_t *lengths,
                                      const void **binaries,
                                      cl_int *binary_status,
                                      cl_int *errcode_ret)

cl_int clBuildProgram (cl_program program,
                       cl_uint num_devices,
                       const cl_device_id *device_list,
                       const char *options,
                       void (*pfn_notify)(cl_program, void *user_data),
                       void *user_data)

cl_kernel clCreateKernel (cl_program program,
                          const char *kernel_name,
                          cl_int *errcode_ret)
```

Some order here, please…

- OpenCL defines a command queue, which is created on a single device
  - within the scope of a context, of course…
- Commands are enqueued to a specific queue
  - Kernels Execution
  - Memory Operations
- Events
  - Each command can be created with an event associated to it
  - Each command execution can be dependent in a list of pre-created events
- Two types of queues
  - In order queue : commands are executed in the order of issuing
  - Out of order queue : command execution is dependent only on its event list completion



- A few queues can be created on the same device
- Commands can dependent on events created on other queues/contexts
- In the example above :
  - C3 from Q1,2 depends on C1 & C2 from Q1,2
  - C1 from Q1,4 depends on C2 from Q1,2
  - In Q1,4, C3 depends on C2

Memory Objects

- OpenCL defines Memory Objects (Buffers/Images)
  - Reside in Global Memory
  - Object is defined in the scope of a context
  - Memory Objects are the only way to pass a large amount of data between the Host & the devices.
- Two mechanisms to sync Memory Objects:
- Transactions - Read/Write
  - Read - take a "snapshot" of the buffer/image to Host memory
  - Write – overwrites the buffer/image with data from the Host
  - Can be blocking or non-blocking (app needs to sync on event)
- Mapping
  - Similar to DX lock/map and command
  - Passes ownership of the buffer/image to the host, and back to the device

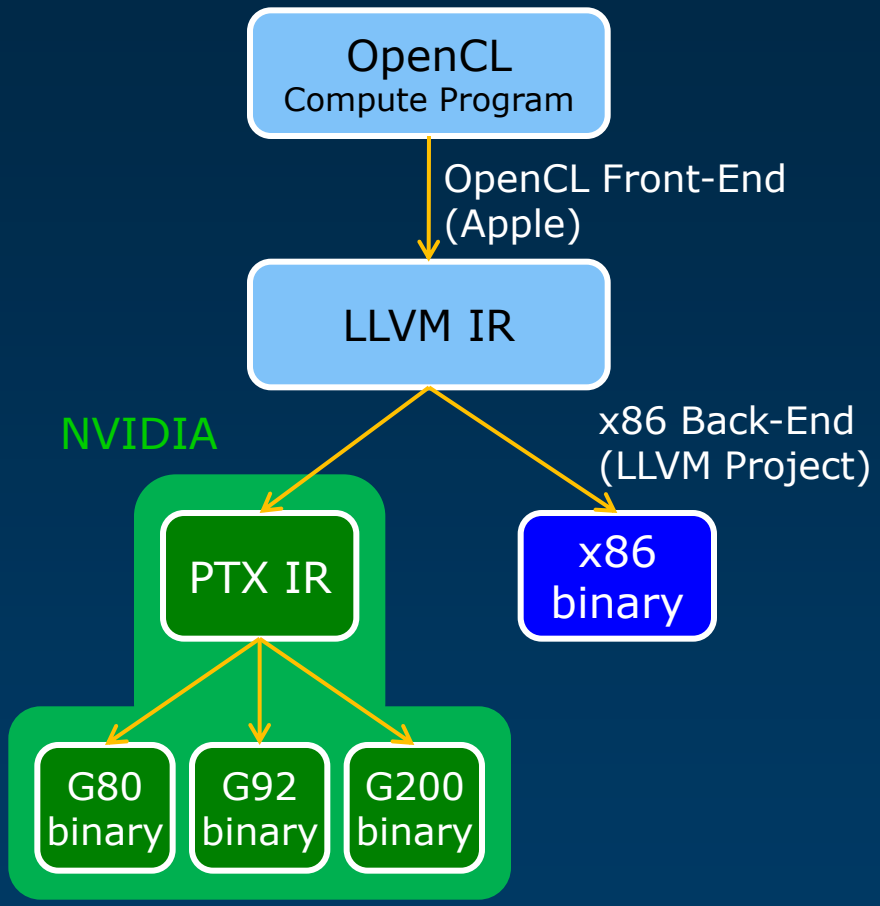
Executing Kernels

- A Kernel is executed by enqueueing it to a Specific command queue
- The App must set the Kernel Arguments before enqueueing
  - Setting the arguments is done one-by-one
  - The kernel's arguments list is preserved after enqueueing
  - This enables changing only the required arguments before enqueueing again
- There are two separate enqueueing API's – Data Parallel & Task Parallel
- In Data Parallel enqueueing, the App specifies the global & local work size
  - Global : the overall work-items to be executed, described by an N dimensional matrix
  - Local : the breakdown of the global to fit the specific device (can be left NULL)

```
cl_int clSetKernelArg (cl_kernel kernel,
                       cl_uint arg_index,
                       size_t arg_size,
                       const void *arg_value)

cl_int clEnqueueNDRangeKernel (cl_command_queue command_queue,
                               cl_kernel kernel,
                               cl_uint work_dim,
                               const size_t *global_work_offset,
                               const size_t *global_work_size,
                               const size_t *local_work_size,
                               cl_uint num_events_in_wait_list,
                               const cl_event *event_wait_list,
                               cl_event *event)

cl_int clEnqueueTask (cl_command_queue command_queue,
                      cl_kernel kernel,
                      cl_uint num_events_in_wait_list,
                      const cl_event *event_wait_list,
                      cl_event *event)
```

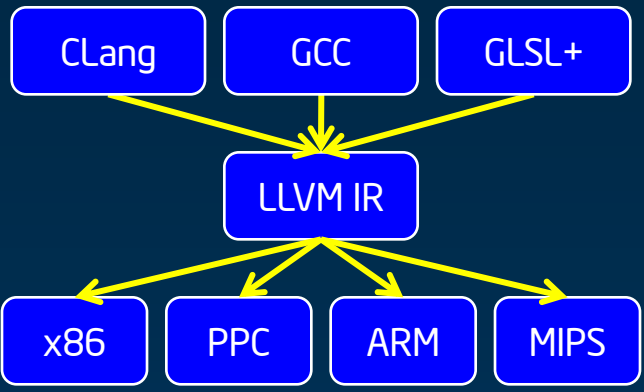
Reality Check – Apple compilation scheme

- OCL compilation process on SnowLeopard (OSX 10.6)
  - Step1: Compile OCL to LLVM IR (Intermediate Representation)
  - Step2: Compile to target device
- NVIDIA GPU device compiles the LLVM IR in two steps:
  - LLVM IR to PTX (CUDA IR)
  - PTX to target GPU
- CPU device uses LLVM x86 BE to compile directly to x86 Binary code.
- So what is LLVM ? Next slide...



The LLVM Project

- LLVM - Low Level Virtual Machine
- Open Source Compiler for
  - Multi Language
  - Cross Platform/Architecture
  - Cross OS



Try out LLVM in your browser!

Bitter Melon the cat says, paste a C/C++ program in the text box or upload one from your computer, and you can see LLVM compile it, meow!!

Type your source code in below: (hints and advice)

```
#include <stdio.h>
#include <stdlib.h>

int factorial(int X) {
  if (X == 0) return 1;
  return X*factorial(X-1);
}

int main(int argc, char **argv) {
  printf("%d\n", factorial(atoi(argv[0])));
}
```

Or upload a file: Browse...

Bytecode size is 760 bytes.

Output from LLVM disassembler

```
; ModuleID = '/tmp/webcompile/_29556_0.bc'
target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:3
target triple = "i386-pc-linux-gnu"
@.str = internal constant [4 x i8] c"%d\0A\00"          ; <[4 x

define i32 @main(i32 %argc, i8** %argv) nounwind  {
entry:
        %tmp7 = load i8** %argv, align 4                ; <i8*>
        %tmp12 = tail call i32 @__strtol_internal( i8* noalias
        %tmp215.i = icmp eq i32 %tmp12, 0               ; <i1>
        br i1 %tmp215.i, label %factorial.exit, label %bb4.i

bb4.i:          ; preds = %bb4.i, %entry
        %indvar.i = phi i32 [ %indvar.next.i, %bb4.i ], [ 0, %e
        %accumulator.tr13.i = phi i32 [ %tmp9.i, %bb4.i ], [ 1,
        %X.tr14.i = sub i32 %tmp12, %indvar.i           ; <i32>
```

OCL C Data Types

- OpenCL C Programming language supports all ANSI C data types
- In addition, the following Scalar types are supported:

Type	Description
half	A 16-bit float. The half data type must conform to the IEEE 754-2008 half precision storage format.
size_t	The unsigned integer type of the result of the sizeof operator (32bit or 64bit).
ptrdiff_t	A signed integer type that is the result of subtracting two pointers (32bit or 64bit).
intptr_t	A signed integer type with the property that any valid pointer to void can be converted to this type, then converted back to pointer to void, and the result will compare equal to the original pointer.
uintptr_t	An unsigned integer type with the property that any valid pointer to void can be converted to this type, then converted back to pointer to void, and the result will compare equal to the original pointer.

Type	Description
char*n*	A 8-bit signed two's complement integer vector.
uchar*n*	A 8-bit unsigned integer vector.
short*n*	A 16-bit signed two's complement integer vector.
ushort*n*	A 16-bit unsigned integer vector.
int*n*	A 32-bit signed two's complement integer vector.
uint*n*	A 32-bit unsigned integer vector.
long*n*	A 64-bit signed two's complement integer vector.
ulong*n*	A 64-bit unsigned integer vector.
float*n*	A float vector.

Address Space Qualifiers

- OpenCL Memory model defined 4 memory spaces
- Accordingly, the OCL C defines 4 qualifiers:
  - `__global, __local, __constant, __private`
- Best to explain on a piece of code:

```
__global float4 *color; // An array of float4 elements
typedef struct {
     float a[3];
     int b[2];
} foo_t;
__global image2d_t texture; // A 2D texture image

__kernel void stam_dugma(
      __global float *output,
      __global float *input,
      __local float *tile)
{
      // private variables
      int in_x, in_y ;
      const unsigned int lid = get_local_id(0));

      // declares a pointer p in the __private address space
      // that points to an int object in address space __global
      __global int *p;
```

Variables outside the scope of kernels must be global

Variables passed to the kernel can be of any type

Internal Variables are private unless specified otherwise

And here's an example to the "specified otherwise"…

Built-in functions

- The Spec specified over 80 built-in functions which must be supported
- The built-in functions are divided to the following types:
  - Work-item functions
    - get_work_dim, get_global_id, etc…
  - Math functions
    - acos, asin, atan, ceil, hypot, ilogb
  - Integer functions
    - abs, add_sat, mad_hi, max, mad24
  - Common functions (float only)
    - Clamp, min, max, radians, step
  - Geometric functions
    - cross, dot, distance, normalize
  - Relational functions
    - isequal, isgreater, isfinite
  - Vector data load & store functions
    - vload*n*, vstore*n*,
  - Image Read & Write Functions
    - read_imagef, read_imagei,
  - Synchronization Functions
    - barrier
  - Memory Fence Functions
    - Read_mem_fence

Vector Addition – Kernel Code

```
__kernel void
dot_product (__global const float4 *a,
             __global const float4 *b, __global float *c)
{
     int gid = get_global_id(0);
     c[gid] = a[gid] + b[gid];
}
```

Vector Addition – Host Code

```
void delete_memobjs(cl_mem *memobjs, int n)
{
    int i;
    for (i=0; i<n; i++)
    clReleaseMemObject(memobjs[i]);
}

int exec_dot_product_kernel(const char *program_source, int n, void *srcA,
                            void *srcB, void *dst)
{
    cl_context context;
    cl_command_queue cmd_queue;
    cl_device_id *devices;
    cl_program program;
    cl_kernel kernel;
    cl_mem memobjs[3];
    size_t global_work_size[1], size_t local_work_size[1], cb;
    cl_int err;
    // create the OpenCL context on a GPU device
    context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU,
                                      NULL, NULL, NULL);
    // get the list of GPU devices associated with context
    clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
```

Vector Addition – Host Code

```
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);
// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0], 0, NULL);
free(devices);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context,
                            CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                            sizeof(cl_float4) * n, srcA, NULL);
memobjs[1] = clCreateBuffer(context,
                            CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                            sizeof(cl_float4) * n, srcB, NULL);
memobjs[2] = clCreateBuffer(context,
                            CL_MEM_READ_WRITE,
                            sizeof(cl_float) * n, NULL, NULL);

// create the program
program = clCreateProgramWithSource(context,
                           1, (const char**)&program_source, NULL, NULL);
// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
// create the kernel
kernel = clCreateKernel(program, "dot_product", NULL);
```

Vector Addition – Host Code

```
    // set the args values
    err = clSetKernelArg(kernel, 0,
                         sizeof(cl_mem), (void *) &memobjs[0]);
    err |= clSetKernelArg(kernel, 1,
                         sizeof(cl_mem), (void *) &memobjs[1]);
    err |= clSetKernelArg(kernel, 2,
                         sizeof(cl_mem), (void *) &memobjs[2]);

    // set work-item dimensions
    global_work_size[0] = n;
    local_work_size[0]= 1;

    // execute kernel
    err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL,
                                 global_work_size, local_work_size,
                                 0, NULL, NULL);
    // read output image
    err = clEnqueueReadBuffer(cmd_queue, memobjs[2], CL_TRUE,
                              0, n * sizeof(cl_float), dst,
                              0, NULL, NULL);

    // release kernel, program, and memory objects
    delete_memobjs(memobjs, 3); clReleaseKernel(kernel);
    clReleaseProgram(program);  clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context); return 0;
}
```

Sources

- OpenCL at Khronos
  - http://www.khronos.org/opencl/
- "Beyond Programmable Shading" workshop at SIGGRAPH 2008
  - http://s08.idav.ucdavis.edu/
- Same workshop at SIGGRAPH Asia 2008
  - http://sa08.idav.ucdavis.edu/